TÉCNICO LISBOA

Exploring the Limits of Cross-Platform Sparse Tensor Processing

Filipe dos Santos Borralho

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Aleksandar Ilic
Prof. Leonel Augusto Pires Seabra de Sousa

Examination Committee

Chairperson: Prof. Pedro Filipe Zeferino Aidos Tomás
Supervisor: Prof. Aleksandar Ilic
Member of the Committee: Prof. Luís Manuel Silveira Russo

June 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Declaro que o presente documento é um trabalho original da minha autoria e que cumpre todos os requisitos do Código de Conduta e Boas Práticas da Universidade de Lisboa.

Abstract

Tensors are the natural way to store multi-dimensional data. Therefore the efficient computation of tensor methods is an important challenge. Some tensor methods, like Matricised Tensor Times Khatri-Rao Product (MTTKRP) and Tensor Times Matrix (TTM), pose as major performance bottlenecks for algorithms commonly used in several areas of research. State of the art optimisations tend to focus on single-device implementations, however modern systems are evolving to become more heterogeneous by combining several acceleration devices, such as multi-core Central Processing Unit (CPU), Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA). In this Thesis, the most prominent tensor methods are analysed as well as the characteristics of the most commonly used architectures when it comes to sparse tensor methods optimisation and acceleration. Implementations for TTM and MTTKRP are developed for all the aforementioned architectures, achieving up to $7\times$ speed up against the state of the art and with the advantage of not being device nor vendor specific. An heterogeneous approach for a system with CPU and GPU is also developed to demonstrate the capabilities of SYCL as a potential standard in heterogeneous computing.

Keywords

Sparse Tensors, TTM, MTTKRP, SYCL, Heterogeneous Systems, FPGA

Resumo

Os tensores são a maneira natural de armazenar dados multidimensionais. Portanto, a computação eficiente de métodos que envolvem tensores é um desafio importante. Alguns destes métodos, como Matricised Tensor Times Khatri-Rao Product (MTTKRP) e Tensor Times Matrix (TTM), apresentam-se como grandes obstáculos ao desempenho de algoritmos frequentemente usados em diversas áreas de investigação. As mais recentes optimizações tendem a concentrar-se em implementações para uma única arquitectura de computação, no entanto, os sistemas modernos estão a evoluir no sentido de se tornarem mais heterogéneos, combinando vários aceleradores, como Central Processing Unit (CPU) multi-core, Graphics Processing Unit (GPU) e Field Programmable Gate Array (FPGA). Nesta Tese, são analisados os mais proeminentes métodos, bem como as características das arquiteturas mais utilizadas no que toca a otimização e aceleração de métodos que envolvem tensores esparsos. Implementações para TTM e MTTKRP são desenvolvidas para todas as arquiteturas mencionadas, alcançando uma melhoria de até $7\times$ em relação às implementações atuais e com a vantagem de não serem específicas para uma arquitectura ou marca. Uma implementação heterogénea, para um sistema com CPU e GPU, também é desenvolvida com o intuito de demonstrar as capacidades do SYCL como uma potencial referência em computação heterogénea.

Palavras Chave

Tensores Esparsos, TTM, MTTKRP, SYCL, Sistemas Heterogéneos, FPGA

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AI	Arithmetic Intensity
ALM	Adaptive Logic Module
ALTO	Adaptative Linearised Tensor Order
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BLCO	Blocked Linearised Coordinate
CARM	Cache Aware Roofline Model
CISS	Compressed Interleaved Sparse Slice
COO	Coordinate
CPD	Canonical Polyadic Decomposition
CPU	Central Processing Unit
CSF	Compressed Sparse Fiber
DPC++	Data Parallel C++
DRAM	Dynamic RAM
DSP	Digital Signal Processing
FLOPs	Floating-Point Operations
FPGA	Field Programmable Gate Array
GEMM	General Matrix Multiplication
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HiCOO	Hierarchical Coordinate
ILP	Instruction-Level Parallelism
MTTKRP	Matricised Tensor Times Khatri-Rao Product

MAC	Multiply-Accumulate
PE	Processing Element
RTL	Register-Transfer Level
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SPMD	Single Program Multiple Data
TC	Tensor Contraction
TEW	Tensor Element-Wise
TS	Tensor-Scalar
TTM	Tensor Times Matrix
TTMc	Tensor Times Matrix chain
TTV	Tensor Times Vector

Chapter 1

Introduction

With the exponential increase in data consumption worldwide, most major research areas are applying complex algorithms to compute and extract information from this data. A diverse range of disciplines such as healthcare [1,2], machine learning [3–7], quantum chemistry [8,9], social network analytics [10], deep learning [11–16] and cybersecurity [17] show how spread the need is. However, attaining efficient processing of this data is far from a trivial task, since it tends to be highly irregular and multi-dimensional. For example, machine learning methods are often applied to images, represented by three dimensions, and videos, represented by four dimensions. With the increase of depth of neural networks, arguably the most prominent computation for this kind of input, the accuracy of the systems improved drastically, however the amount of computation required also increased [18].

Originally proposed in the 19th century by G. Ricci, a tensor is a generalization of vectors and matrices in higher orders [19] and are the de-facto representation of high-dimensional data. Since real-world data is usually multi-dimensional, tensors are of major importance as they are used to represent it. With the need for efficient and performant computation, optimisation of tensor operations has gained a lot of attention over the years.

Tensors have huge storage requirements, some can take up to petabytes [20]. However, tensors that represent real-world data have a considerably high sparsity, which lead to the development of a plethora of data storage formats to reduce memory space. Due to this sparsity, data access patterns are highly irregular, making it a performance bottleneck for computation but also opening a window for further optimisation. A massive amount of software solutions have been developed with attempts to improve the performance of such computations. From efficient storage formats to novel data-parallel approaches aiming at exploiting the capabilities of modern massively parallel architectures and even data reordering algorithms. Data reordering is important for improving spatial and temporal locality of memory accesses, which in memory bound algorithms are the major bottleneck [21]. The design of specialized hardware accelerators for tensor processing is also an active area of research, with new architectures for loading,

storing, reordering and computing tensors being developed [22, 23].

1.1 Motivation

There are many tensor operations, from operations between two or more tensors to operations with scalars [24]. However, Matricised Tensor Times Khatri-Rao Product (MTTKRP) and Tensor Times Matrix (TTM) stand out as subject of intensive research, because they are the central kernels of the most used tensor decomposition algorithms [25–31]. This Thesis will focus on these two operations. MTTKRP consists of a matrix product between the matricized tensor and the Khatri-Rao product of dense matrices, one for each of the flattened dimensions of the tensor. TTM consists of the product between a tensor and a matrix, it works the same way as a matrix multiplication, except the tensor's equivalent to rows is spread along more than one dimension. The methods mentioned are not only difficult to compute efficiently because of the sparsity and data irregularity but also because of their complexity. Sparse MTTKRP is an example of a computation whose complexity is not to be dismissed, with it being $\mathcal{O}(N^{\epsilon}mR)$, where N is the number of dimensions of the tensor, R is the number of columns of the matrices used in the Khatri-Rao product, m is the number of nonzero elements and ϵ varies from zero to one depending on the level of optimisation of the implementation [26].

Storing and computing over zeros is highly inefficient and redundant. On the other hand, simply storing and computing the nonzero elements makes data accesses irregular and non coalesced. Sparsity and data irregularity are tied together. Therefore, storage of sparse tensors is also a very active area of research. As mentioned, storing zeros is not beneficial, however not storing nor computing them causes irregularity which is undesired, so several implementations and optimisations exist for different scenarios, for example the simple Coordinate (COO) storage format, which stores all nonzero elements alongside their indexes across all the tensor's dimensions, is superior for uniformly distributed sparse tensors.

From the vast amount of available computational architectures, it is non-trivial to find the one that best suites the specific needs of a certain method applied to a certain tensor. State-of-the-art approaches typically focus on single device solutions. However none was found, at the time of this work, that tried to exploit even further the heterogeneity of modern systems and with the emerging systems leaning more and more towards heterogeneity, there is an opportunity to be explored.

In this Thesis, the exploitation of said systems will be tackled by developing optimised implementations of the most prominent tensor methods for each of the architectures available. Then, with resort to SYCL, an heterogeneous framework, that distributes data and offloads computation between the architectures, is developed. Such framework allows for an efficient use of the system hence increasing the performance when compared to single device implementations.

SYCL is attracting attention precisely for the simplicity offered when interacting with heterogeneous systems. Modern systems have access to different kinds of accelerators, with SYCL offloading computation to these can be done independently of which accelerator is being targeted, enabling the possibility of utilising a single source code in order to target multiple accelerators. Naturally, for better performance and efficiency, each algorithm should have in mind the characteristics of its respective target device.

1.2 Objectives

To address the increase in heterogeneity and availability of solutions, an unified framework for tensor computation is lacking. Therefore, this Thesis has the following objectives:

- Compare the most prominent tensor methods, namely TTM and MTTKRP, by exploring their existing and deriving novel approaches targeting efficient execution on different architectures, e.g. Central Processing Unit (CPU), Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA).
- Development of a framework that exploits heterogeneous systems by developing specialised algorithms for each architecture and distributing the workload between them.

1.3 Outline

This Thesis is structured as follows:

- **Chapter 2 - Background and State of the Art:** This chapter presents the fundamentals behind tensors, their representations and their most relevant methods. The main hardware architectures used as well as the fundamentals of heterogeneous computing are discussed. An introduction to roofline models and their importance in the scope of this Thesis is also provided. State-of-the-art implementations are thoroughly described.
- **Chapter 3 - Analysis for General-Purpose Architectures:** This chapter unveils our TTM and MTTKRP implementations for general-purpose architectures, such as the CPU and GPU. A theoretical analysis of said implementations is presented, with derivations of the kernels' Arithmetic Intensity (AI) being included. The performance limits for each of the architectures are also tested.
- **Chapter 4 - Analysis for Specialised Architectures:** This chapter unveils our TTM and MTTKRP designs for specialised architectures, such as the FPGA. A theoretical analysis of said designs is provided, with an look over the resource utilisation and derivations of the kernels' potential peak performance as well as AI being included.

- **Chapter 5 - Experimental Results:** On this chapter, the results of our implementations are revealed and analysed, as well as compared against some of the current state-of-the-art implementations.
- **Chapter 6 - Conclusion:** We conclude by analysing the work developed throughout this Thesis and suggesting future improvements and points of research.

Chapter 2

Sparse Tensors: Background and State of the Art

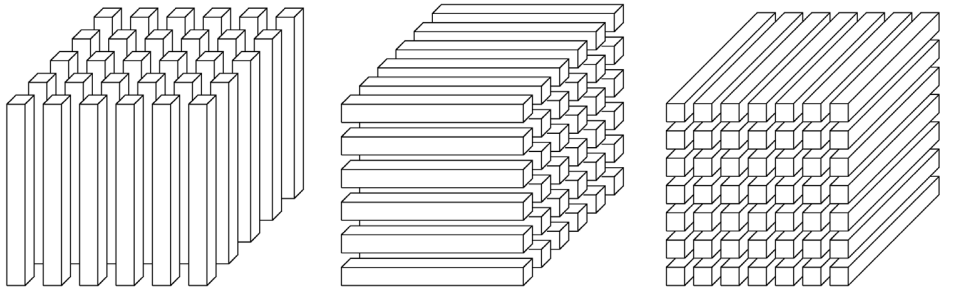
Tensors and their methods have a rather complex mathematical background behind them, even when considering their traditional dense formulation. However the concept of sparsity and its integration in the tensor domain brings additional challenges and optimisation opportunities. To better understand the problem, this chapter provides a brief introduction to the fundamentals of tensor nomenclature, followed by a formal definition of sparsity in tensor notation. The most relevant storage formats for sparse tensors are examined, while an overview of the tensor methods and their application to sparse tensors is also presented. The range modern device architectures available for these computations and their particularities are thoroughly described, including multi-core CPU, GPU and FPGA devices. Since modern systems typically combine all these devices, heterogeneous programming models and open challenges within this research domain are also discussed. State-of-the-art storage formats and algorithms for MTTKRP and TTM operations on GPUs are thoroughly analysed.

2.1 Tensor Nomenclature

Tensors are multidimensional arrays which represent high dimensional data. A tensor's order is the dimensionality of the array, e.g. a first-order tensor is a vector while a second-order tensor is a matrix. Tensors of order greater or equal than three are typically referred to as high-order tensors. For simplicity, it is common to apply the term tensor only to high-order tensors, while matrices and vectors are referred to as low-order tensors [32].

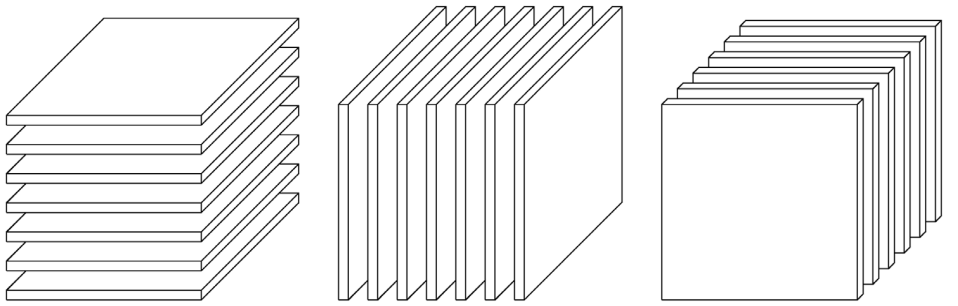
Dimensions are typically referred to as modes, so a tensor has as many modes as its order, e.g. a third-order tensor has mode-zero, mode-one and mode-two. A fiber is a vector formed by fixing all

modes of the tensor but one. As an example, the fibers of a matrix are either its rows or its columns depending on the mode fixed. Similarly, a slice is a matrix formed by fixing all modes of the tensor but two. Figure 2.1 provides a representation of the fibers of a third-order tensor, while Figure 2.2 provides a representation of the slices of a third-order tensor.



(a) Mode-1 (column) fibers: $\mathbf{x}_{:jk}$ (b) Mode-2 (row) fibers: $\mathbf{x}_{i:k}$ (c) Mode-3 (tube) fibers: $\mathbf{x}_{ij:}$

Figure 2.1: Fibers of a third-order tensor [33]



(a) Horizontal slices: $\mathbf{X}_{i::}$ (b) Lateral slices: $\mathbf{X}_{:j:}$ (c) Frontal slices: $\mathbf{X}_{::k}$ (or $\mathbf{X}_k$)

Figure 2.2: Slices of a third-order tensor [33]

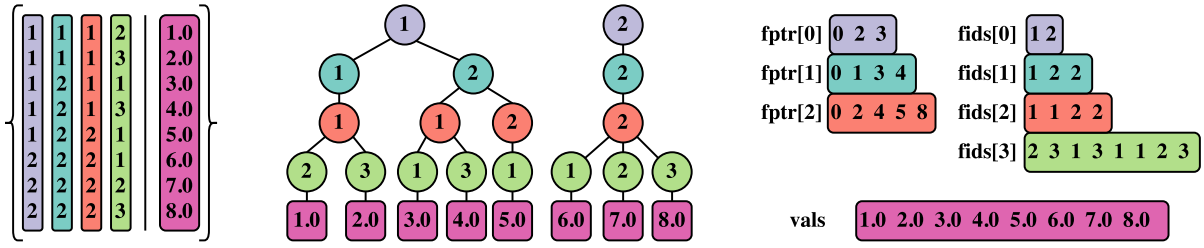
High-order tensors have huge storage requirements, since they tend to have several millions of elements. However, real-world tensors are sparse, meaning that most of the tensor's elements are zero and, therefore, need not be stored nor explicitly computed upon. Allowing for further optimisation but raising additional challenges because of highly irregular data accesses.

2.2 Storage Formats

The storage of tensors is a research area of its own, in particular when it comes to sparse tensors [34]. Several formats for storing the relevant information from these datasets have been developed. The most straight forward approach is only storing the nonzero elements and their mode indexes, it is called the COO format. Although being a baseline format it is widely used due to its simplicity. Figure 2.3(a) presents a COO representation for a fourth-order tensor.

The main disadvantage of this format is its inability to avoid storing redundant information for certain sparse data. For example, all elements in the same slice will by definition have the same indexes for all modes except two, the same applies to elements in same fiber where they all have the same indexes for all modes but one. Therefore, storing the same index for several elements gives no additional information while using extra storage.

Other formats try to solve this problem, with Compressed Sparse Fiber (CSF) [35] being the most representative one. The CSF format takes into consideration implementing the idea of a tree like structure, where each mode is a level and paths from root to leaf encode a nonzero coordinate. In Figure 2.3, it is possible to observe an example of a fourth-order tensor represented in CSF. Figure 2.3(b) brings the conceptual idea of the tree structure, while the actual implementation are exemplified in Figure 2.3(c). When relating Figures 2.3(b) and 2.3(c), it is possible to observe that the tree structure is implemented with resort to pointer arrays. Each dimension adds two arrays – one to indicate what indices are present in that dimension ($fids$), and one to show how to slice the next dimension ($fptr$).



(a) Coordinate format. (b) CSF: conceptual. (c) CSF: implementation.

Figure 2.3: Fourth-order tensor example in COO and CSF [25]

The practical implementation diverges slightly from the conceptual idea, because of the need of extra arrays for bookkeeping, leading to a solution that under certain circumstances requires more storage then COO. Another disadvantage of this format is that it is mode agnostic, meaning that if an operation is to be performed across a mode that is not the leaf mode, either there will be a performance loss or there will be the need to reorder the tensor. Despite these drawbacks, CSF is still widely used due to its performance, since the hierarchical way in which data is stored allows for better locality and less memory accesses in general.

State-of-the-art approaches also propose many variations of these formats in order to improve them for a certain framework and/or architecture [36]. Also, there is a completely different approach that attempts to further reduce the storage requirements of the tensor by linearising the mode indexes, the linearised formats. Both these variations and approach will be further discussed in Section 2.5.

2.3 Tensor Methods

Tensor methods can be classified in four categories according to their behaviour. These categories consist in Tensor Element-Wise, Tensor-Scalar, Tensor Contraction and Sequence Methods. In this section, a general description for each of the categories will be provided, also the several distinct methods/operations that form each category will be described in the text following the general description of the category itself.

2.3.1 Tensor Element-Wise Methods

Tensor Element-Wise (TEW) [24], as the name suggests, are methods where an operation is applied to every corresponding pair of elements from two tensors. This type of operations can be addition, subtraction, multiplication or division. Naturally, in order to be possible to perform this operation, both tensors must have the same order and shape, meaning the same size across all modes. An addition between two dense tensors, $Z = X + Y$, is expressed as follows:

$$Z(m_0, \ldots, m_K) = X(m_0, \ldots, m_K) + Y(m_0, \ldots, m_K), \quad \forall_{elements} \tag{2.1}$$

Where X and Y are input tensors, Z the resulting tensor and m_0 to m_K are the tensor modes. For sparse tensors the reasoning is the same but only applied to the nonzero elements. However, some of the nonzero elements of one of the tensors may match a zero of the other tensor. Since the sparse formats only store the nonzero elements, the only way to know whether an entry has a match in the other data structure is to check all the entries. This leads to complexity $\mathcal{O}(N^2)$, where N is the number of entries in a data structure. As an optimisation, it is common to sort the entries of both tensors in the same mode order, allowing the complexity to become $\mathcal{O}(N \log N)$. Algorithm 1 implements TEW multiplication by iterating over all nonzero elements and if it finds a corresponding pair of elements in both tensors, it creates a new entry in the output tensor with the mode indexes of the corresponding the pair and the product of both values.

Algorithm 1 Pseudo-code for a Sparse TEW multiplication

Input: Sorted sparse tensors X and Y;
Output: Sorted sparse tensor Z;

1: **while** nonzeros left **do**
2: **if** $m_0^X \ldots m_K^X$ == $m_0^Y \ldots m_K^Y$ **then**
3: val^Z = $val^X \times val^Y$;
4: $(m_0^Z \ldots m_K^Z)$ = $(m_0^X \ldots m_K^X)$;
5: **end if**
6: **end while**
7: **return** Z

2.3.2 Tensor-Scalar Operations

Tensor-Scalar (TS) [24] bare several resemblances to the previous, TEW methods, with the major difference being the use of a scalar instead of a second tensor. Since the negative and the inverse of a scalar are also scalars (assuming the scalar is not zero), only addition and multiplication require implementing. Also, as before, for sparse tensors the operation is only applied to the nonzero elements, resulting in a complexity of $\mathcal{O}(N)$ for the algorithm. Algorithm 2 implements TS multiplication by iterating over all nonzero elements, creating a new entry in the output tensor with the mode indexes of the input tensor and the value of the input tensor times the scalar.

Algorithm 2 Pseudo-code for a Sparse TS multiplication

Input: Sparse tensor X and scalar s;
Output: Sparse tensor Y;
1: **while** nonzeros left **do**
2: $val^Y = val^X \times s$;
3: $(m_0^Y \dots m_K^Y) = (m_0^X \dots m_K^X)$;
4: **end while**
5: **return** Y

2.3.3 Tensor Contraction Operations

A Tensor Contraction (TC) [37–41] is the analog to General Matrix Multiplication (GEMM) in the multidimensional realm of tensors. Two tensors are multiplied across their matching modes resulting in a tensor with the remaining modes. For example with two fourth-order tensors: $C = A \cdot B$ with $A \in \mathbb{R}^{I_0 \times I_1 \times I_2 \times I_3}$ and $B \in \mathbb{R}^{I_2 \times I_3 \times I_4 \times I_5}$, then $C \in \mathbb{R}^{I_0 \times I_1 \times I_4 \times I_5}$. I_0 to I_3 are the dimensions of tensor A in modes zero to three, while I_2 to I_5 are the dimensions of tensor B in modes zero to three.

There are a few TCs that are notable and are often even treated independently of regular contractions. One of these is GEMM, which is a contraction between two second-order tensors, though it is outside of the scope of this work. Others are Tensor Times Vector (TTV), which is a contraction between a Kth-order tensor and a first-order one, and TTM [42, 43], which is a contraction between a Kth-order tensor and a second-order one.

For TTV, all fibers of the Kth-order tensor are multiplied with the vector resulting in a tensor of order K-1. For TTM, all fibers are multiplied with each column of the matrix. Since a fiber is a vector, this operation consists of several vector-matrix dot products, which generate new vectors with length equal to the number of columns of the matrix. Therefore, the resulting fiber length becomes equal to the number of columns in the matrix.

Figure 2.4 provides an example of a dense TTM with a third-order tensor. In the example, the tensor has $I \times J$ fibers of length K. Each of these fibers multiplies with each of the F columns of the matrix, resulting in a dense tensor with $I \times J$ fibers of length F. For sparse TTM the rationale is similar to the

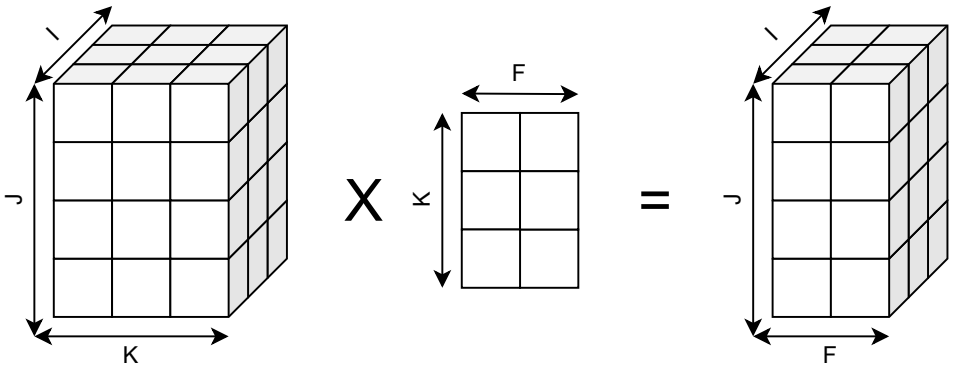


Figure 2.4: Example of a dense TTM with a third-order tensor

one presented in Figure 2.4, however only the fibers with one or more nonzero elements are computed. Also, the output of the operation is no longer a sparse tensor but instead a semi-sparse tensor meaning that not all of the output tensor's fibers are sparse. This happens because all fibers that have at least one nonzero element after the dot product with a dense column of the matrix generate a nonzero element for the output tensor. Since all columns of the matrix are dense, all fibers that have at least one nonzero element generate a dense fiber for the output tensor [43].

Algorithm 3 Pseudo-code for a Sparse TTM

Input: Sparse tensor X and matrix U;
Output: Semi-Sparse tensor Y;

```
1: while fibers left do
2:     for column : matrix do
3:         Y.append(fiber · column);
4:     end for
5: end while
6: return Y
```

Algorithm 3 implements sparse TTM by iterating over all fibers of the input tensor and for every fiber iterating over all columns of the matrix. When iterating over the columns, for each fiber-column dot-product a new element of the output tensor is computed and so after performing dot-products of a fiber with all columns of the matrix a fiber of the output tensor is computed.

2.3.4 Sequence Methods

Sequence methods [24] are, as the name suggests, a sequence of smaller methods combined to form a kernel. The most notable cases are Tensor Times Matrix chain (TTMc) and MTTKRP. Since the former is a sequence of TTM methods (already elaborated in Section 2.3.3), this section focuses mostly on the latter.

MTTKRP is widely used in tensor decomposition, as it is part of one of the most used algorithms

[25–28]. Tensor decomposition is used to approximate a high-order tensor with lower-order tensors. Canonical Polyadic Decomposition (CPD) in particular approximates high-order tensors with matrices, with MTTKRP being key to computing the matrices that better approximate the tensor.

MTTKRP uses a K-th order tensor and K-1 matrices as input and outputs a matrix. The tensor is matricized and the matrices are used to compute a Khatri-Rao product. Therefore, to properly understand the MTTKRP, it is necessary to introduce the concept of tensor matricization, as well as Khatri-Rao and Kronecker products. Both products are matrix products meaning they are applied to two matrices. The Kronecker product is a generalization of a matrix outer-product. Given two matrices $A \in \mathbb{R}^{I \times J}$ and $B \in \mathbb{R}^{K \times R}$ their Kronecker product is denoted by matrix $C \in \mathbb{R}^{IK \times JR}$,

$$C = A \otimes B = \begin{bmatrix} a_{00}B & \dots & a_{0J}B \\ \vdots & & \vdots \\ a_{I0}B & \dots & a_{IJ}B \end{bmatrix} \tag{2.2}$$

The Khatri-Rao product is a column-wise Kronecker product. Given two matrices $A \in \mathbb{R}^{I \times R}$ and $B \in \mathbb{R}^{J \times R}$ their Khatri-Rao product is denoted by matrix $C \in \mathbb{R}^{IJ \times R}$,

$$C = A \odot B = \begin{bmatrix} a_{:0} \otimes b_{:0} & \dots & a_{:R} \otimes b_{:R} \end{bmatrix} \tag{2.3}$$

Matricization is a kind of reshaping that flattens all tensor's modes but one, hence changing the tensor to a second-order one, in other words, a matrix. Given $T \in \mathbb{R}^{M_0 \times \dots \times M_{K-1}}$ then matricizing the tensor along mode zero outputs $\tilde{T} \in \mathbb{R}^{M_0 \dots M_{K-2} \times M_{K-1}}$.

MTTKRP consists of a GEMM between the matricized tensor and the Khatri-Rao product of $K-1$ matrices with same number of columns, where K is the order of the tensor before matricizing. Given $K-1$ matrices with R columns the Khatri-Rao product of these is a matrix $U \in \mathbb{R}^{M_0 \dots M_{K-2} \times R}$ and the output of the MTTKRP is a matrix $V \in \mathbb{R}^{M_{K-1} \times R}$.

Algorithm 4 Pseudo-code for a Sparse MTTKRP

Input: Third-order sparse tensor $X \in \mathbb{R}^{I \times J \times K}$ and dense matrices $B \in \mathbb{R}^{J \times R}$, $C \in \mathbb{R}^{K \times R}$;
Output: Dense matrix $A \in \mathbb{R}^{I \times R}$;

```
1: while nonzeros left do
2:     (i, j, k) = indexes of nonzero;
3:     for r = 1 ... R do
4:         A(i, r) += val^X × B(j, r) × C(k, r);
5:     end for
6: end while
7: return Y
```

For sparse MTTKRP, as can be observed in Algorithm 4, from each mode index of a nonzero element it is possible to extract either which row of the output the element contributes to or which row of one of the matrices the element multiplies with. Since it is possible to compute each element's indi-

vidual contribution to the output matrix without having to compute the matricized tensor and Khatri-Rao products, in order to avoid redundant computation and extra storage, these formulations tend to be not implemented directly but rather integrated into tensor operations.

2.4 Heterogeneous Systems and Programmability

In order to improve the performance of the methods mentioned above, the state of the art approaches focus on their optimisation across a range of hardware devices with Application Specific Integrated Circuit (ASIC)s [22, 23], FPGAs [44, 45] and GPUs [37, 38, 43, 46, 47] being the most adopted. However, those approaches only target single-device architectures. In contrast, this Thesis focuses on modern heterogeneous platforms which may combine all these devices into a single execution environment. This is not an easy task since all devices have their own characteristics and programming model. This Thesis will focus on the GPU and the FPGA, as well as the CPU.

2.4.1 CPU

The CPU, with over a half-century of history, is the most well-known and ubiquitous architecture. It is sometimes referred to as a scalar architecture because it is designed to process serial instructions efficiently. It is optimized to exploit Instruction-Level Parallelism (ILP) so that serial programs can be executed faster.

Modern CPU processors consist of several multi-thread superscalar cores with sophisticated mechanisms used to dynamically exploit ILP and execute multiple out-of-order instructions per clock cycle. To deliver high performance, they fetch many instructions at once, detect dependencies, utilize sophisticated branch prediction mechanisms, and execute them in parallel. To mitigate slow accesses to main memory, Dynamic RAM (DRAM), the CPU comes with several cache levels, where the smaller and faster caches are private to each core, while the larger and slower cache level is shared across them. In order to make full use of the caches, the CPU is better suited for coarse-grained parallelism to increase temporal and spacial locality of data [48].

Figure 2.5 presents an example of a modern CPU architecture with four cores. Each core has two fetch/decode units making it superscalar and each fetch/decode unit has four Single Instruction Multiple Data (SIMD) lanes. There are also four execution contexts which allow interleaved thread execution. The out-of-order control logic, branch predictor and memory pre-fetcher are the mentioned sophisticated mechanisms used to exploit ILP and deliver high performance.

Being the most widely used generic processors in computing, every application that leverages compute acceleration still requires a CPU to handle task orchestration. In certain scenarios, using a CPU

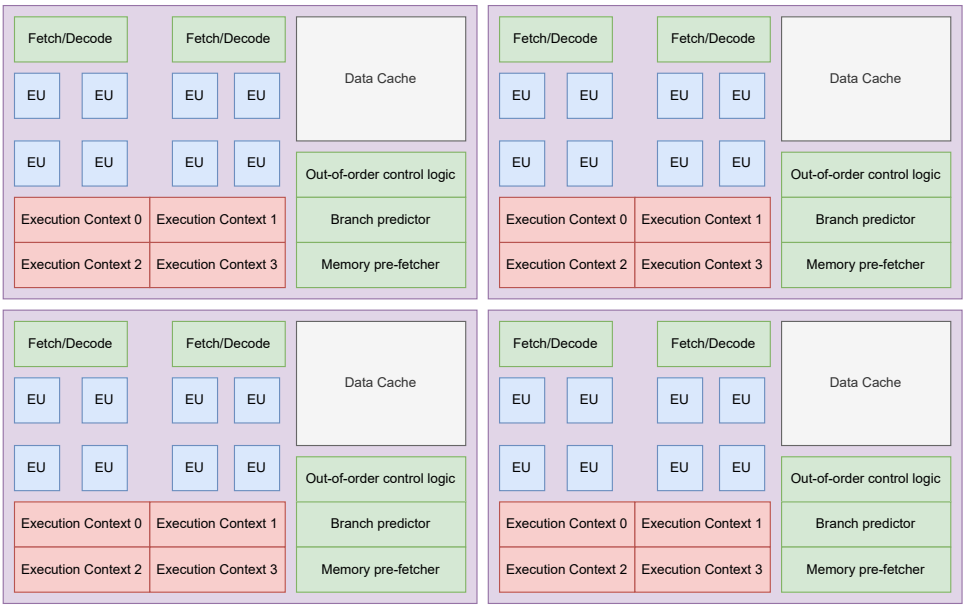


Figure 2.5: Modern Quad-Core CPU example

for computing may be advantageous when compared to offloading to a GPU or a FPGA, e.g. algorithms that are serial in nature.

There also are several programming frameworks and tools that exploit the available parallelism. OpenMP [49] exploits the parallelism within a multi-core CPU and vector intrinsics exploit parallelism in the explicit SIMD fashion.

2.4.2 GPU

The GPU is a processor comprised of massively parallel, smaller, and more specialized cores than those generally found in the CPU. Its architecture efficiently processes vector data and is often referred to as a vector architecture. They dedicate more silicon space to compute and less to cache and control. As a result, its hardware explores less ILP and relies instead on software-given parallelism to achieve performance and efficiency. High performance is achieved by exploiting fine-grained parallelism through multi-threaded execution of large and independent data, which amortizes the cost of simpler control and smaller caches. Employing a Single Instruction Multiple Threads (SIMT) execution model where multi-threading and SIMD are leveraged together [50].

Figure 2.6 presents an example of a GPU architecture in small scale. There are twenty streaming multiprocessors with each having four warp selectors. The warp selectors function similarly to the fetch/decode units on the CPU and each of them as multiple SIMD lanes. There is also a shared memory private to each streaming multiprocessor.

Due to their increased computational capabilities, data consumption also happens at a higher rate.

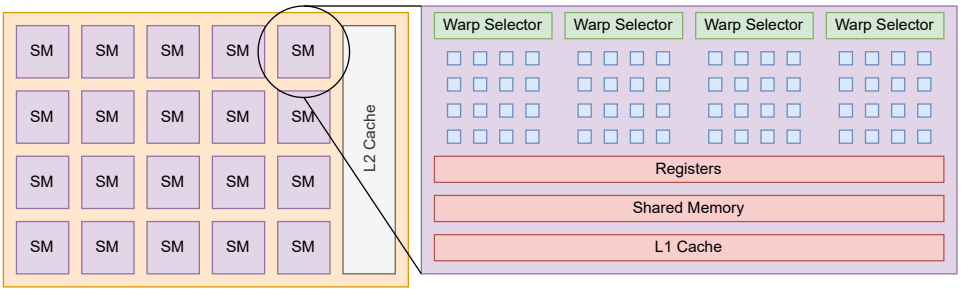


Figure 2.6: GPU example with 20 streaming multiprocessors

Therefore, the GPU usually comes with state of the art memories with higher bandwidth when compared with ones offered by the CPU.

The preferred Application Programming Interface (API) for computation on Nvidia GPUs is CUDA, while OpenCL is the standard for the remaining vendors. Both allow thread-level control of the work distribution, making the GPU ideal for fine-grained parallelism.

2.4.3 FPGA

Unlike the previously elaborated devices, which are software-programmable fixed architectures, the FPGA is a re-configurable device and its compute engine is defined by the user. When targeting an FPGA, the specific hardware components are designed for the problem at hand, which are laid out on its fabric in space, and those components can all execute in parallel or in pipelined fashion. Because of this, its architecture is sometimes referred to as an architecture for spatial computation.

The FPGA is a massive array of small processing units consisting of a massive amount of programmable 1-bit Adaptive Logic Module (ALM), configurable memory blocks and Digital Signal Processing (DSP) blocks, that support variable precision floating-point and fixed-point operations. All these resources are connected by a mesh of programmable wires that can be activated in a as needed basis [51].

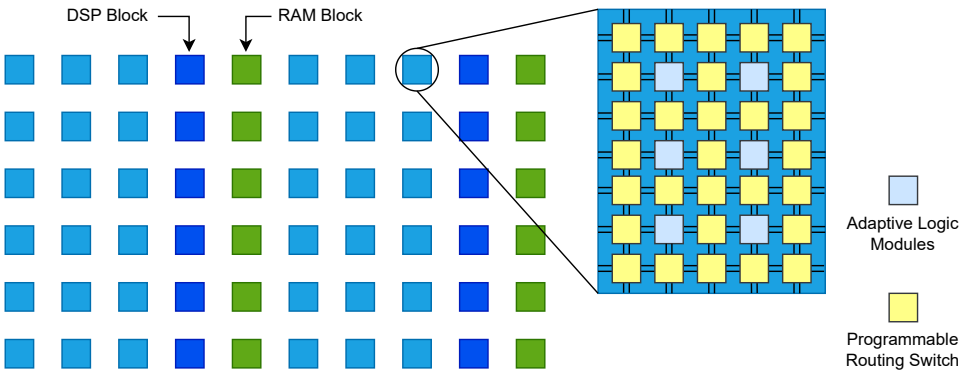


Figure 2.7: FPGA's segment example

Figure 2.7 presents a segment of an FPGA fabric as an example. The segment has multiple ALM, digital signal processors and memory blocks. All connected by routing switches which compose the mesh of programmable wires.

Software is not executed on the FPGA in the same sense that compiled and assembled instructions are executed on fixed architectures. Instead, data flows through customized deep pipelines that match the operations expressed in the software. Because the dataflow pipeline hardware matches the software, control overhead is eliminated, which results in improved performance and efficiency. Operations are pipelined, so new instruction streams operating on different data start executing every clock cycle. While with the previous architectures, instruction stages are pipelined and new instruction start executing every clock cycle. Although pipeline parallelism is the primary form of parallelism for the FPGA, it can be combined with other types of parallelism. For example, SIMD, task parallelism and superscalar execution can be utilized with pipeline parallelism to achieve maximum performance.

Programming an FPGA is done through a Hardware Description Language (HDL), as in software programming languages there are several kinds of HDL. Some are more closely related to the Register-Transfer Level (RTL) abstraction that focuses on describing the flow of signals between registers and the logical operations performed on them. For example, Verilog and VHDL. These, however, create a huge burden on the developer when designing larger projects, similar to Assembly in software. Other languages describe the specialized hardware components through syntax very similar to that of the high-level languages used in software. This high-level languages are then translated to RTL. One example is Xillinx's Vivado HLS. Naturally, these offer more simplicity when describing the circuits, however they provide less freedom on how to implement the logic itself.

2.4.4 Heterogeneous Computing and Open Challenges

With the rising relevance of heterogeneous computing, the need for a programming model that would facilitate the development of high performance applications for all this range of systems arose.

The Khronos SYCL defines an abstract Single Program Multiple Data (SPMD) programming model that tackles the mentioned challenges [52]. It provides the desired unified model, developers program at a higher level than the native acceleration API, for example CUDA or OpenCL, but always have access to lower-level code that allows users to target any accelerator without having to change their source code. However, in order for a developer to take the most performance out of an accelerator, the code must be adapted to the characteristics of that accelerator's architecture. As seen in the previous sections, vector and spatial architectures, for instance, are different from the general-purpose processors. Therefore, it is natural that a code optimized for one might not be optimized for the other, even though it still executes.

When comparing FPGAs and GPUs, the FPGA tends to deal better with single-task kernels whereas the GPU tends to go along better with parallel kernels. This happens as the latter relies on large data

parallel workloads to achieve performance. On the other hand, the former, when using single-task kernels, attempts to pipeline loop execution, making it more efficient, since every clock cycle, successive iterations of the loop enter the first stage of the pipeline. Dependencies across loop iterations expressed in the software can be resolved in the hardware by routing the output of one stage to the input of an earlier dependent stage.

When compared with other APIs such as CUDA, SYCL has proven to provide comparable performance [53]. With said performance and portability, it is a candidate to future standard in heterogeneous computing. There are many SYCL implementations in development with Intel's OneAPI DPC++ being one of the most actively developed open-source implementations.

2.5 State of the Art on Sparse Tensor Storage and Processing

As mentioned in Section 2.2, the current state of the art approaches tend to rely proposing novel tensor storage formats in order to derive high performance algorithms, by providing many optimizations over COO and CSF. Some of those novel formats are Hierarchical Coordinate (HiCOO) [54] or Compressed Interleaved Sparse Slice (CISS) [22].

HiCOO stores a sparse tensor in a sparse-blocked pattern with a pre-specified cubic block. Within these blocks the regular COO format is used. Since the mode indexes are limited by the size of the block, its possible to use less bits to represent them. This way HiCOO can still be mode generic like COO while in average occupying less space in memory. An example of a conversion of a third-order tensor from COO to HiCOO is shown on Figure 2.8. The tensor entries in COO format are sorted and then partitioned in blocks. Then, from their original coordinates and from the block coordinates, their in-block coordinates are computed. To revert the tensor back to COO format the block coordinates are multiplied by the block size and added with the in-block coordinates of the elements.

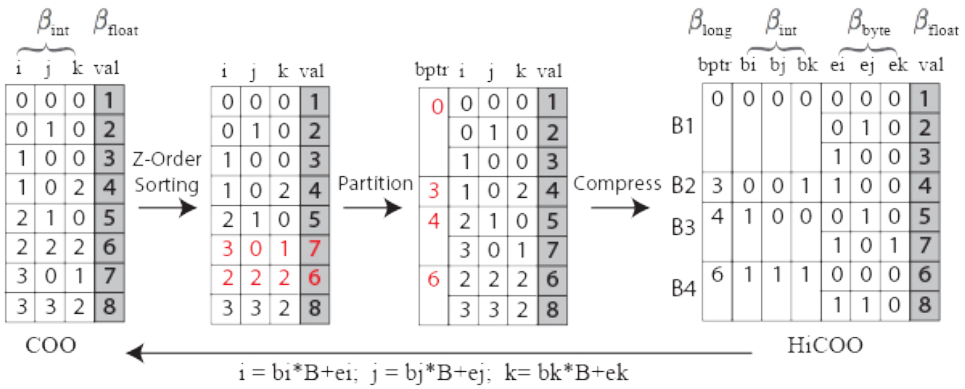


Figure 2.8: Example of conversion of a tensor from COO to HiCOO [54]

CISS is very similar to COO, but introduces a novelty in order to help mitigating its inherent redun-

dancy regarding the storage of mode indexes. Since all values stored in COO are non-zero, by creating an extra entry with value equal to zero, it is possible to create headers. For example, in a third-order tensor the header defines the index of the slice. Then all entries until the next header will only contain information about their fiber index and element index, as well as a nonzero value. This way it is possible to avoid storing the same slice index multiple times. Figure 2.9 has the same tensor as Figure 2.8 but represented in CISS format. Each header is in gray and as can be observed below each header are all elements that share the slice with that index.

i/j	0	0	1	1	0	0	2	1	2	3	0	3
k	-	0	0	-	0	2	-	0	2	-	1	2
val	0	1	2	0	3	4	0	5	6	0	7	8

Figure 2.9: Example of tensor in CISS format

In an attempt to further reduce the data stored, linearized formats were created. These formats used the mode indexes and combine them to make an unique identifier for each nonzero element. Some examples of these formats are Adaptative Linearised Tensor Order (ALTO) [55] and Blocked Linearised Coordinate (BLCO) [56].

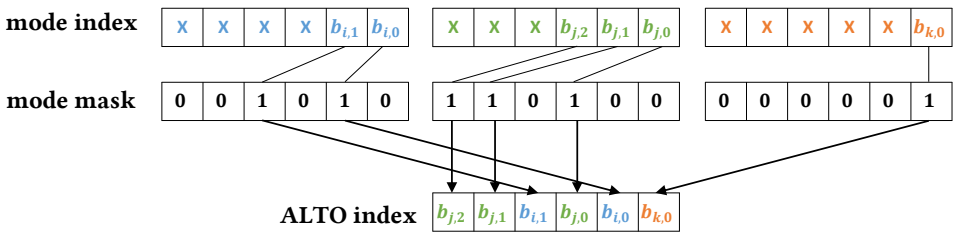


Figure 2.10: Example of encoding in a third-order tensor [55]

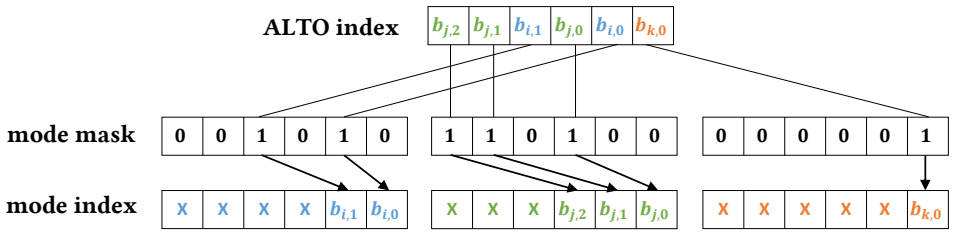


Figure 2.11: Example of decoding in a third-order tensor [55]

In Figures 2.10 and 2.11, it is possible to observe how the modes indexes are encoded to generate an unique identifier and how the identifier is reverted back to the original indexes. Each mode has a bit mask that is used to encode the corresponding mode index by shifting its bits as well as to decode

the ALTO index by shifting back the bits corresponding to that masks mode, while placing zeros on the remaining bits.

Depending on many factors, such as the tensor, the method and the architecture, different storage formats offer varying performances. However it is common for tensor libraries to adopt a format and implement the methods in that format. Nowadays, the most widely adopted format is CSF.

The preferred architectures for tensor methods were discussed in Section 2.4. As mentioned, with the ones further analysed being the GPU and the FPGA. The former is arguably the most widespread and accessible accelerator making it very prolific in tensor method implementations, specially for the most computationally challenging methods, MTTKRP and TTM. Also, these methods are major bottlenecks in the most used tensor decomposition algorithms [25–31].

Current state-of-the-art GPU implementations for MTTKRP focus on achieving good load-balance and efficient use of the shared memory of the GPU [46, 47]. The use of shared memory is important for the sparse version of the algorithm, since it involves highly irregular data accesses and is bandwidth-bounded. Hence the use of shared memory allows for shorter access times improving the overall performance. On the other hand, imbalance in the workload might not be obvious at first but is definitely a demanding problem. For example, in the case of a third-order tensor, the naïve approach would be to assign to each slice a work-group since all slices are independent in the output's computation. However slices are likely to differ in the number of nonzero elements as well as in their distribution. Another solution could be for each thread to compute the same number of elements, but then there would be a need for synchronization hindering the overall performance.

Algorithm 5 implements a sparse MTTKRP for a third-order tensor in a GPU as depicted in [46], its inputs are the third-order tensor in CSF format and two dense matrices. In the algorithm, the GPU's grid of threads is assigned in the following way: the thread's global index along the x-axis defines the column of the input and output matrices. The block's index along the y-axis, defines a slice to be computed and, therefore, also the output row. The thread's local index along the y-axis defines which fiber, within the block's slice, to be computed. With this distribution, each block computes their part of the output and stores it in the shared memory (lines 7-15). Afterwards, a local reduction is performed in an interleaved fashion, adding the intermediate results of each fiber under the same slice (lines 16-21). By the end of this reduction, the first thread of each block will then store the final result to the output matrix (lines 22-24).

The load balance in this algorithm is not guaranteed, in fact depending on the nonzero element distribution across the slices and fibers, different blocks and different threads within blocks may have different execution times. With the required synchronization in the algorithm, the worst case would be the dominant one. One possible solution to this problem is the preprocessing of the tensor and re-adaptation of the format for better balance.

Algorithm 5 Sparse MTTKRP in CSF for GPU [46]

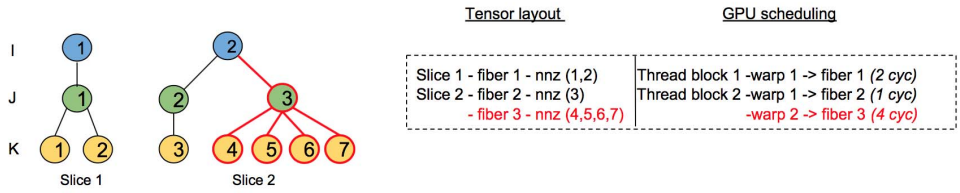
Input: Sparse tensor $X \in \mathbb{R}^{I \times J \times K}$ and dense matrices $B \in \mathbb{R}^{J \times R}$, $C \in \mathbb{R}^{K \times R}$;
Output: Dense matrix $M \in \mathbb{R}^{I \times R}$;

1: $bdx = blockDim.x$, $bdy = blockDim.y$;
2: $bix = blockIdx.x$, $biy = blockIdx.y$;
3: $thx = threadIdx.x$, $thy = threadIdx.y$;
4: $shr[bdy][bdx]$, $shr_{id}[bdy]$;
5: $f = bix \times bdx + thx$;
6: **if** $f < R$ **then**
7: **for** $i = thy + slcPtr[biy] \dots slcPtr[biy + 1]$ **by** $i+ = bdy$ **do**
8: $shr_{id}[thy] = fbrIdx[i]$;
9: **for** $j = fbrPtr[i] \dots fbrPtr[i + 1]$ **do**
10: $inC+ = vals[j] \times C[inFbrIdx[j]][f]$;
11: **end for**
12: $inB+ = inC \times B[sh_{id}[thy]][f]$;
13: **end for**
14: $shr[thy][thx] = inB$;
15: $\_sync()$;
16: **for** $stride = bdy/2 \dots 0$ **by** $stride \gg= 1$ **do**
17: **if** $thy < stride$ **then**
18: $shr[thy][thx]+ = shr[thy + stride][thx]$;
19: $\_sync()$;
20: **end if**
21: **end for**
22: **if** $thy == 0$ **then**
23: $M[biy][f] = shr[0][thx]$;
24: **end if**
25: **end if**

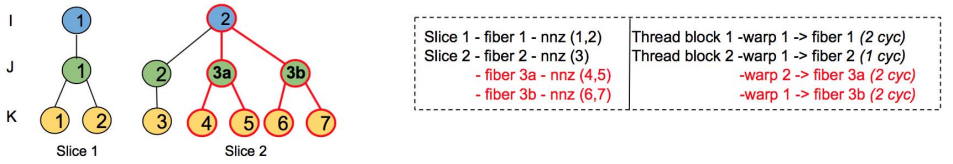
As depicted in Figure 2.12, the work proposed in [47] focuses on load balancing for CSF tensors, so that all fibers have a approximately the same number of elements and all slices have approximately the same number of fibers. This is achieved by introducing some redundancy in the CSF format. In Figure 2.12(a), a the tensor is represented in the regular CSF format. However, the slices and fibers are heavily imbalanced. To solve this issue, new slices and fibers with the same indexes as the existing ones are created. This goes against CSF's principle that each node in one level of the tree represents one index. Having more slices and fibers also requires more memory space, since more indexes need to be stored. Therefore, there is a trade-off. In order to achieve better work balance some redundancy is introduced the format.

The other method to be analysed is TTM. Since in it every fiber-column product is independent from another, fine-grained parallelism is the natural approach. Implementations also focus on data spatial and temporal locality as well as an efficient use of the shared memory available in the GPU.

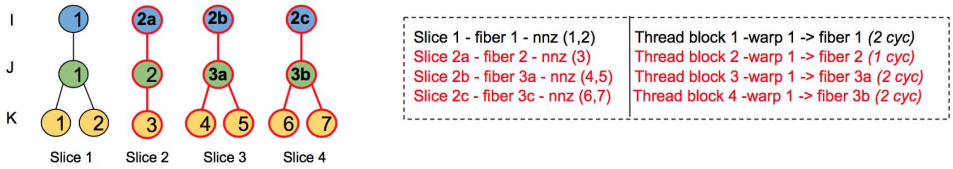
Algorithm 6 implements a sparse TTM for a third-order tensor in a GPU as proposed in [43], its inputs are the third-order tensor in CSF format and a dense matrix. In the algorithm, the fine-granularity is evident. Each thread computes the dot product of a single fiber with one or more columns, depending



(a) Original Tensor causing inter-warp and inter-thread block load imbalance. Required cycle 4.



(b) After splitting fibers to address inter-warp load imbalance. Required cycle 3.



(c) After splitting slices along with fibers to address inter-thread block load imbalance. Required cycle 2.

Figure 2.12: Example of the balancing of CSF [47]

Algorithm 6 Sparse TTM in CSF for GPU [43]

Input: Sparse tensor $X \in \mathbb{R}^{I \times \ldots \times K}$ and dense matrix $U \in \mathbb{R}^{K \times F}$;
Output: Semi-sparse tensor $Y \in \mathbb{R}^{I \times \ldots \times F}$;

1: $y_{shr}[blockDim.y][blockDim.x]$;
2: $n = \frac{F}{blockDim.x}$;
3: $tidy = threadIdx.y$, $tidx = threadIdx.x$;
4: $i = blockIdx.x \times blockDim.y + tidy$;
5: **for** $l = 0 \ldots n$ **do**
6: $\quad r = tidx + l \times blockDim.x$;
7: $\quad y_{shr}[tidy][tidx] = 0.0$;
8: $\quad \_sync()$;
9: $\quad$ **for** $j = fbr_{ptr}[i] \ldots fbr_{ptr}[i+1]$ **do**
10: $\quad\quad k = k_{idx}[j]$;
11: $\quad\quad y_{shr}[tidy][tidx] += values[j] \times u[k][r]$;
12: $\quad$ **end for**
13: $\quad \_sync()$;
14: $\quad output[i][r] = y_{shr}[tidy][tidx]$;
15: $\quad \_sync()$;
16: **end for**
17: **return** Y;

---

on the ratio between the size of a thread block along the x-axis and the number of columns. The global index of a thread along the y-axis defines which fiber is assigned. This way, each block contributes to the output with a portion of the fibers, the shared memory is used to keep the block's contribution which

afterwards is stored in the global memory. Since each fiber may have more than one nonzero element, storing constantly every intermediate result to the global memory would hinder severely performance and cause contention problems. By iterating first over the columns and only afterwards over the fiber elements, the chance for short rows staying in caches increases. There is also the use of rank blocking on the outer for-loop, however its benefit depends on the nonzero distribution of the input tensor.

As a possible optimisation, the decision of whether to use rank blocking or not could be done after loading the tensor and analysing its features. With two kernels available, one with rank blocking and the other without it.

2.6 Roofline Model

In order to analyse and compare the performance of different algorithms on different devices, a tool that ties a representation of the algorithm with the device capabilities is required. The roofline model does precisely this [57–59]. It translates the device's capabilities into two roofs: one representing the memory bandwidth, commonly expressed in bytes per second, and the other representing the peak performance, commonly expressed in operations per second.

The algorithm itself is represented by its performance, again in operations per second, and by its AI, commonly in operations per byte transferred from the memory. While the first allows to measure how close the algorithm's performance is to the maximum performance it can achieve on that specific device, the second offers insight on whether the algorithm is bound by the device's memory or by the device's computational power.

Modern architectures possess specific hardware components for complex single-cycle operations, e.g. Multiply-Accumulate (MAC), as well as complex memory hierarchies. As such the original roofline model does not fully portray the device's capabilities. For that reason, instead of the original roofline model, a more insightful version, Cache Aware Roofline Model (CARM) [60–64], is commonly used.

Figure 2.13 illustrates the inclusiveness of CARM in a modern architecture when compared to the original roofline model. The major difference between these models, when it comes to performance analysis, is the number of regions defined. While on the original roofline, the kernel can be either memory or compute bound, on CARM there is an undefined region between the memory and compute bound regions. This happens because the kernel can be bound by the bandwidth of a certain memory level, which would make it memory bound. However, by optimising and improving spatial and temporal locality of data accesses, the kernel starts utilising a faster memory level. Hence, since the AI did not change, the kernel becomes compute bound.

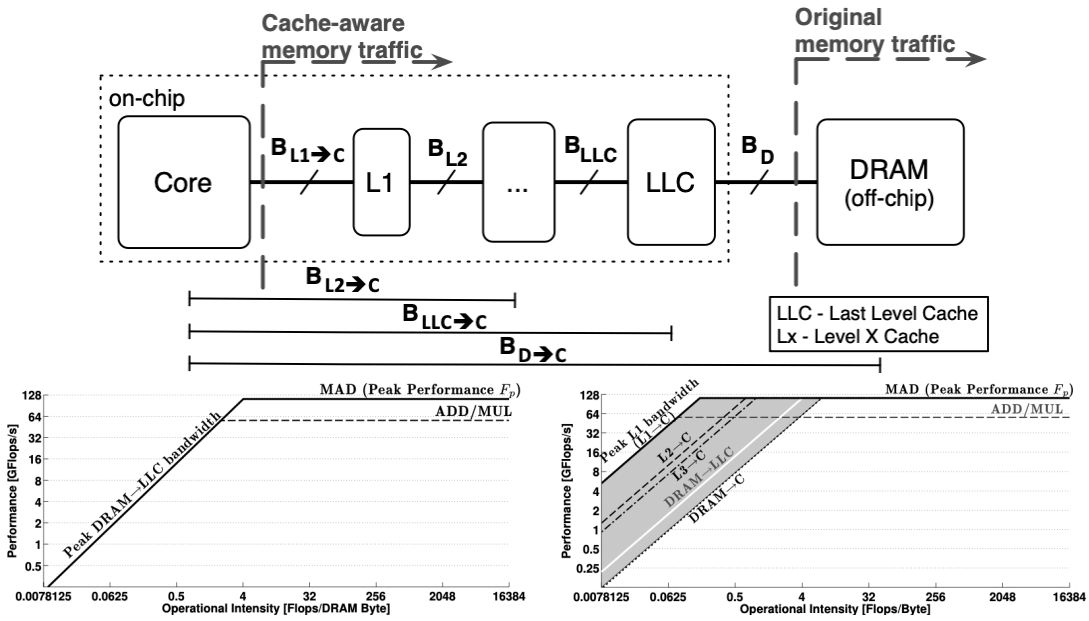


Figure 2.13: Comparison between original roofline model and CARM [60]

2.7 Summary

This chapter started with a introduction to the nomenclature used in tensor domain and the definition of sparsity and sparse tensors. Further, the baseline storage formats for sparse tensors were discussed as well as their main advantages and disadvantages. Four different categories of tensor methods were addressed and discussed alongside with illustrative algorithms for each. The building blocks of the framework to be implemented followed with the architectures to be explored being introduced. Their preferred type computation was analysed and the SYCL programming model was presented. Roofline models, an important tool for analysis on the wide variety of architectures used, were also addressed, which special emphasis on CARM. Finally, the state-of-the-art storage formats, algorithms and implementations for the methods were thoroughly discussed.

Chapter 3

Sparse Tensor Processing on Programmable Architectures

In this chapter, a solution for TTM and MTTKRP processing on general-purpose architectures, such as the CPU and GPU, is presented and has its AI derived as well as its peak performance explored with resort synthetic datasets. Before advancing to the kernels, let us establish Table 3.1 as the notation for the remainder of the chapter.

$SlcCnt$	total number of slices in the tensor
$FbrCnt$	total number of fibers in the tensor
$NnzCnt$	total number of non-zero elements in the tensor
$ColCnt$	total number of columns in the matrices
$FbrPSlc$	number of fibers in a slice
$NnzPSlc$	number of non-zero elements in a slice
$NnzPFbr$	number of non-zero elements in a fiber

Table 3.1: Notation used throughout this chapter

3.1 Data-Parallel Sparse Tensor Processing

The development of said solution was done in Intel's OneAPI Data Parallel C++ (DPC++), which is one of Khronos SYCL implementations. SYCL provides an unified model, where developers program at a higher level than the native acceleration API, but always have access to lower-level code that allows users to target any accelerator without having to change their source code. Besides its portability, when compared with other API such as CUDA, SYCL has proven to provide comparable performance [52]. Therefore it can be considered for a future standard in heterogeneous programming and has been the choice in this Thesis.

Also sparse tensors, even more than sparse matrices due to having higher order, are very prone to variability both in their shape and non-zero element distribution. Therefore creating one kernel that is fully optimized for all scenarios is not trivial and so, in order to minimize this problem, two different versions were created, tested, and compared for each method.

3.1.1 Tensor Times Matrix (TTM)

The TTM method consists in dot-products between each fiber of a tensor and all columns of a matrix, as was elaborated on Section 2.3.3. In this section, a solution for sparse TTM, with CSF as the storage format for the tensor, is derived and analysed. The purpose of this analysis is to measure the AI and performance of said solution against several datasets, real and synthetic, on both the CPU and the GPU. With this analysis and with resort to roofline models, described in Section 2.6, it is expected to confirm the bottlenecks and test the limits of utilisation of the architectures.

3.1.1.A Kernel V1: Element-centric TTM approach

For TTM with a sparse tensor stored in CSF format, the first approach is to make it so that each thread computes one element of the output. Therefore each thread requires access to one fiber and to one column of the matrix.

```
// FbrCnt * ColCnt threads -> one for each element of the output
range<2> globalSize(fbrCnt, colCnt);
range<2> localSize(1, colCnt);
nd_range<2> numItems(globalSize, localSize);

event e { q.submit([&](handler &h) {
    h.parallel_for(numItems, [=](nd_item<2> item) {
        const auto fbr { item.get_global_id(0) };
        const auto col { item.get_local_id(1) };
        auto tmp { 0.0f };

        // Load fiber boundaries: accFbrPtr[fbr] and accFbrPtr[fbr+1]
        for (auto ele { accFbrPtr[fbr] }; ele < accFbrPtr[fbr+1]; ++ele) {
            // Load in-fiber index and value
            const auto k { (accKIdx[ele] - 1) * colCnt };
            const auto val { accValues[ele] };

            // Load element of the column
```

```
19              // Compute product and accumulate
20              tmp += val * accMatrix[k + col];
21          }
22
23          // Store fiber-column dot product to global memory
24          accOutput[fbr * colCnt + col] = tmp;
25      });
26  }) };
```

Listing 3.1: TTM Kernel V1

In Kernel 3.1, threads are created in the same number as output elements (line 2) with each of them computing the dot-product between their assigned fiber and column. Therefore each thread starts by loading its fiber boundaries (line 13) and then for all non-zero elements in that fiber loads both their in-fiber index as well as their value (lines 15-16). Hence, from the tensor, there are two loads for boundaries plus two more loads for each non-zero element in the fiber. From the matrix, for each non-zero element in the fiber there is one more load, to fetch the corresponding element in the column (line 20). As such, the total amount of loads from memory can be expressed as $2 + 2 \times NnzPFbr + NnzPFbr$. Since each thread computes the dot-product between a fiber and a column, the number of operations performed is one multiply and one addition per non-zero element in the fiber (line 20), i.e., the total amount of Floating-Point Operations (FLOPs) is equal to the $2 \times NnzPFbr$. This operation generates one element of the output, which is subsequently stored in the output fiber, thus contributing to the only one store operation performed (line 24). Assuming all data types are 4-byte wide, the AI can be expressed as follows:

$$AI_{v1} = \frac{1}{4} \times \frac{2 \times NnzPFbr}{2 + 2 \times NnzPFbr + NnzPFbr + 1} = \frac{1}{2} \times \frac{NnzPFbr}{3 + 3 \times NnzPFbr} = \frac{1}{6} \times \frac{NnzPFbr}{NnzPFbr + 1} \quad (3.1)$$

According to the expression derived in 3.1, the AI of each thread may be different depending on the amount of non-zero elements in the fibers that are assigned to the thread for processing. However, these always range between a minimum and maximum value. The minimum AI can be achieved when the fiber has the least possible number of non-zero elements, which is one. Thus, the minimum AI can be expressed as follows:

$$\min(AI_{v1}) = \min\left(\frac{1}{6} \times \frac{NnzPFbr}{NnzPFbr + 1}\right) = \frac{1}{6} \times \frac{1}{2} = \frac{1}{12} \quad (3.2)$$

The maximum AI is achieved when the number of non-zero elements in the fiber is large enough so that $\frac{NnzPFbr}{NnzPFbr+1} \approx 1$. As such, the maximum AI can be expressed as follows:

$$\max(AI_{v1}) = \max\left(\frac{1}{6} \times \frac{NnzPFbr}{NnzPFbr+1}\right) \approx \frac{1}{6} \times 1 = \frac{1}{6} \tag{3.3}$$

3.1.1.B Kernel V2: Fiber-centric TTM approach

Another approach to efficiently extract data-parallelism in TTM processing is to assign each thread to compute an entire fiber of the output (instead of a single element). In this approach, each thread still requires access to one fiber, but now it also requires access to the whole matrix instead of just a column (as previously elaborated in Kernel V1 with element-centric TTM processing).

```
1  // FbrCnt threads -> one for each fiber of the output
2  range<1> globalSize(fbrCnt);
3  range<1> localSize(wgSize);
4  nd_range<1> numItems(globalSize, localSize);
5
6  event e { q.submit([&](handler &h) {
7      h.parallel_for(numItems, [=](nd_item<1> item) {
8          const auto fbr { item.get_global_id(0) };
9          float tmp[colCnt];
10
11         for (auto col { 0 }; col < colCnt; ++col) {
12             tmp[col] = 0.0f;
13         }
14
15         // Load fiber boundaries: accFbrPtr[fbr] and accFbrPtr[fbr+1]
16         for (auto ele { accFbrPtr[fbr] }; ele < accFbrPtr[fbr + 1]; ++ele) {
17             // Load in-fiber index and value
18             const auto k { (accKIdx[ele] - 1) * colCnt };
19             const auto val { accValues[ele] };
20
21             // Load corresponding row of the matrix
22             // Compute product and accumulate
23             for (auto col { 0 }; col < colCnt; ++col) {
24                 tmp[col] += val * accMatrix[k + col];
25             }
26         }
27
28         // Store output fiber to global memory
```

```
29          for (auto col { 0 }; col < colCnt; ++col) {
30              accOutput[fbr * colCnt + col] = tmp[col];
31          }
32      });
33  }) };
```

Listing 3.2: TTM Kernel V2

In Kernel 3.2, the number of threads created is the same as the number of output fibers (line 2), where each thread is responsible for computing the dot-products of the assigned fiber against all columns of the matrix. Therefore each thread also starts by loading its fiber boundaries (line 16) and then for all non-zero elements in that fiber it also loads both their in-fiber index as well as their value (lines 18-19). As such, from the tensor, there are two loads (for boundaries) plus two more loads for each non-zero element in the fiber. From the matrix, there are as many loads as columns in the matrix for each non-zero element of the fiber (lines 23-25). As a result, the total amount of loads is equal to $2 + 2 \times NnzPFbr + NnzPFbr \times ColCnt$. Since each thread computes the dot-product between a fiber and all columns of the matrix (lines 23-25), the number of operations performed is one multiply and one addition per column per non-zero element in the fiber, i.e., the total amount of FLOPs is equal to $2 \times NnzPFbr \times ColCnt$. Since each thread in Kernel V2 generates one fiber of the output, and the output fibers have as many elements as there are columns in matrix, then one store per column is required (lines 29-31). This brings the total amount of stores to be equal to $ColCnt$. Assuming all data types are 4-byte wide the AI can be expressed as follows:

$$
\begin{aligned}
AI_{v2} &= \frac{1}{4} \times \frac{2 \times NnzPFbr \times ColCnt}{2 + 2 \times NnzPFbr + NnzPFbr \times ColCnt + ColCnt} = \\
&= \frac{1}{2} \times \frac{NnzPFbr \times ColCnt}{2 \times (NnzPFbr + 1) + ColCnt \times (NnzPFbr + 1)} = \\
&= \frac{1}{2} \times \frac{NnzPFbr}{NnzPFbr + 1} \times \frac{ColCnt}{ColCnt + 2}
\end{aligned}
\tag{3.4}
$$

According to the expression derived in 3.4, the AI varies depending on the number of non-zero elements in the fiber as well as on the number of columns in the matrix. Again, these AIs will also always range between a minimum and a maximum value, which can be calculated in a similar manner to the one previously adopted when analyzing the AI ranges for Kernel 3.1. As before, the minimum AI can be achieved when the fiber has the least non-zero elements (which is one), and the matrix has the least number of columns (which occurs when it is a vector). As such, the minimum AI for the fiber-centric TTM approach can be expressed as follows:

$$\min(AI_{v2}) = \min\left(\frac{1}{2} \times \frac{NnzPFbr}{NnzPFbr+1} \times \frac{ColCnt}{ColCnt+2}\right) = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} = \frac{1}{12} \tag{3.5}$$

The maximum AI, on the other hand, is achieved when both the number of non-zero elements in the fiber and the number of columns in the matrix are large enough such that $\frac{NnzPFbr}{NnzPFbr+1}$ and $\frac{ColCnt}{ColCnt+2}$ are approximately one. Correspondingly, the maximum AI for the fiber-centric TTM approach can be expressed as follows:

$$\max(AI_{v2}) = \max\left(\frac{1}{2} \times \frac{NnzPFbr}{NnzPFbr+1} \times \frac{ColCnt}{ColCnt+2}\right) \approx \frac{1}{2} \times 1 \times 1 = \frac{1}{2} \tag{3.6}$$

The comparison between the Element-centric (Kernel 3.1) and Fiber-centric (Kernel 3.2) TTM data-parallel approaches can be done by analysing their ranges of attainable AI. Kernels 3.1 and 3.2 have the same minimum AI ($1/12$), however kernel 3.2 has a 3$\times$ higher maximum ($1/2$ vs. $1/6$). It is important to reinforce that all calculations involving AI were done under the assumption that all fibers with no non-zero elements are not considered in the computation. If such fibers were to be introduced, the maximum AI would remain unchanged, but the minimum AI would drop down to zero.

3.1.2 Matricised Tensor Time Khatri-Rao Product (MTTKRP)

Next we present and analyse algorithms for sparse MTTKRP, with CSF as the storage format for the tensor. Sparse computation of this method consists of multiplying each non-zero element of the tensor with a row of a matrix for all dimensions but one, meaning a tree-dimensional tensor requires two matrices as was elaborated on Section 2.3.4. This analysis consists of evaluating the AI and performance of said solution against synthetic datasets on general-purpose architectures, namely the CPU and the GPU. Resorting to roofline models, described in Section 2.6, this analysis aims to confirm the bottlenecks and test the limits of utilization on the architectures.

3.1.2.A Kernel V1: Element-centric MTTKRP approach

For MTTKRP on sparse tensors stored in CSF format, the first approach is analogue to the one presented for TTM on Section 3.1.1.A. Each thread computes one element of the output, therefore each thread is assigned with one slice of the tensor, one column of matrix1 and one column of matrix2. Note that these columns must match, e.g. a thread gets column zero on both matrices. Similarly to the kernel presented for TTM, all threads are completely independent and can be run in parallel without any need for synchronisation.

```
1 // SlcCnt * ColCnt -> one for each element of the output
```

```
range<2> globalSize(slcCnt, colCnt);
range<2> localSize(1, colCnt);
nd_range<2> num_items(globalSize, localSize);

event e { q.submit([&](handler &h) {
    h.parallel_for(num_items, [=](nd_item<2> item) {
        const auto slc { item.get_global_id(0) };
        const auto col { item.get_local_id(1) };

        auto inB { 0.0f };
        auto inC { 0.0f };

        // Load slice boundaries: accSlcPtr[slc] and accSlcPtr[slc+1]
        for (auto fbr { accSlcPtr[slc] }; fbr < accSlcPtr[slc + 1]; ++fbr) {
            // Load fiber index
            const auto j { (accFbrIdx[fbr] - 1) * colCnt };

            // Load fiber boundaries: accFbrPtr[fbr] and accFbrPtr[fbr+1]
            for (auto ele { accFbrPtr[fbr] }; ele < accFbrPtr[fbr + 1]; ++ele) {
                // Load in-fiber index and value
                const auto k { (accKIdx[ele] - 1) * colCnt };
                const auto val { accValues[ele] };

                // Load element of the column of matrix2
                // Compute product and accumulate
                inC += val * accMatrix2[k + col];
            }

            // Load element of the column of matrix1
            // Compute product and accumulate
            inB += inC * accMatrix1[j + col];
            inC = 0.0f;
        }

        // Store output element to global memory
        accOutput[slc * colCnt + col] = inB;
    });
```

```
39  }) };
```

Listing 3.3: MTTKRP Kernel V1

Kernel 3.3 creates as many threads as there are output elements (line 2). Each thread starts by loading the slice boundaries (line 15). Then the fiber index (line 17) and the fiber boundaries (line 20), this process is repeated once for each fiber within the slice. Since one fiber's end is the start of a new fiber, the number of loads for the fiber boundaries is equal to number of fibers present in the slice plus one. Now for each non-zero element of the current fiber, both the element's index and value are loaded (lines 22-23). This is repeated for every fiber in the slice so it is done for every single non-zero element in the slice. Therefore from the tensor it is necessary to load twice for the slice boundaries, twice for each fiber in the slice for the fiber index and boundary, and twice for each non-zero in the slice for the element index and value. From matrix2 there is a load for each element in the slice (line 27) and from matrix1 there is one load for each fiber in the slice (line 32). As such, the total number of loads can be expressed as $2 + (2 \times FbrPSlc + 1) + 2 \times NnzPSlc + FbrPSlc + NnzPSlc$. Since each thread computes one element of the output, there is only one store per thread (line 37). As for computations, for each non-zero in the slice there is one MAC (line 27) and for each fiber there is one more MAC (line 32). Since one MAC corresponds to two operations, the total number of computations per thread can be expressed as $2 \times FbrPSlc + 2 \times NnzPSlc$. Assuming all data types used are 4-byte wide, the AI of this kernel can be expressed as follows:

$$\begin{aligned} AI_{v1} &= \frac{1}{4} \times \frac{2 \times FbrPSlc + 2 \times NnzPSlc}{2 + (3 \times FbrPSlc + 1) + (3 \times NnzPSlc) + 1} = \\ &= \frac{1}{2} \times \frac{FbrPSlc + NnzPSlc}{3 \times (FbrPSlc + NnzPSlc) + 4} \end{aligned} \tag{3.7}$$

According to the expression derived in 3.7, the AI of each thread depends on the characteristics of the slice it was assigned. Namely, on the number of fibers and non-zero elements of the slice. Like in the previous method, it is possible to determine the upper and lower bounds of the AI for this kernel. The lower bound can be achieved when the slice has the minimum amount of fibers and non-zero elements possible, which is one in both cases. Thus, the minimum AI is as follows:

$$\min(AI_{v1}) = \min\left(\frac{1}{2} \times \frac{FbrPSlc + NnzPSlc}{3 \times (FbrPSlc + NnzPSlc) + 4}\right) = \frac{1}{2} \times \frac{2}{10} = \frac{1}{10} \tag{3.8}$$

On the other hand, the upper bound can be achieved when both the number of fibers and the number of non-zero elements in the slice are large enough so that the following equation is verified.

$$\max(AI_{v1}) = \max\left(\frac{1}{2} \times \frac{FbrPSlc + NnzPSlc}{3 \times (FbrPSlc + NnzPSlc) + 4}\right) \approx \frac{1}{2} \times \frac{1}{3} = \frac{1}{6} \tag{3.9}$$

3.1.2.B Kernel V2: Row-centric MTTKRP approach

Data-parallelism can also efficiently extracted in MTTKRP processing by assigning to each thread the entirety of both matrices, making it responsible for the computation of a whole output row, instead of being responsible for the computation of only a single element as happened in Kernel 3.3.

```
// SlcCnt threads -> one for each row of the output
range<1> globalSize(slcCnt);
range<1> localSize(4);
nd_range<1> num_items(globalSize, localSize);

event e { q.submit([&](handler &h) {
    h.parallel_for(num_items, [=](nd_item<1> item) {
        const auto slc { item.get_global_id(0) };

        float inB[colCnt], inC[colCnt];

        // Load slice boundaries: accSlcPtr[slc] and accSlcPtr[slc+1]
        for (auto fbr { accSlcPtr[slc] }; fbr < accSlcPtr[slc + 1]; ++fbr) {
            // Load fiber index
            const auto j { (accFbrIdx[fbr] - 1) * colCnt };

            // Load fiber boundaries: accFbrPtr[fbr] and accFbrPtr[fbr+1]
            for (auto ele { accFbrPtr[fbr] }; ele < accFbrPtr[fbr + 1]; ++ele) {
                // Load in-fiber index and value
                const auto k { (accKIdx[ele] - 1) * colCnt };
                const auto val { accValues[ele] };

                // Load corresponding row of matrix2
                // Compute product and accumulate
                for (auto col { 0 }; col < colCnt; ++col) {
                    inC[col] += val * accMatrix2[k + col];
                }
            }

```

```
30              // Load corresponding row of matrix1
31              // Compute product and accumulate
32              for (auto col { 0 }; col < colCnt; ++col) {
33                  inB[col] += inC[col] * accMatrix1[j + col];
34                  inC[col] = 0.0f;
35              }
36          }
37
38          // Store output row to global memory
39          for (auto col { 0 }; col < colCnt; ++col) {
40              accOutput[slc * colCnt + col] = inB[col];
41          }
42      });
43  }) };
```

Listing 3.4: MTTKRP Kernel V2

Kernel 3.4, creates threads in the same number as slices in the tensor, which leaves each of the threads responsible for computing one row of the output. As the tensor loads are the same as in the previous kernel, the difference in loads comes from the matrices where Kernel 3.4 always loads one row from whichever matrix (lines 25-27 and 32-35). As such, using the same notation as before, the total number of loads can be expressed as $2 + (2 \times FbrPSlc + 1) + 2 \times NnzPSlc + ColCnt \times (FbrPSlc + NnzPSlc)$. The differences also extend to the the number of stores and to the computations. In the first, instead of storing a single element, a row of elements is stored (lines 39-41). While on the latter, as each fiber is now computed against the whole of the matrix (lines 25-27 and 32-35), instead of single column, the total number of computations grows proportionally to the number of columns in the matrices, being expressed as $ColCnt \times (2 \times FbrPSlc + 2 \times NnzPSlc)$. Again assuming all data types used are 4-bytes wide, the AI of this kernel can be expressed as follows:

$$
\begin{aligned}
AI_{v2} &= \frac{1}{4} \times \frac{2 \times ColCnt \times (FbrPSlc + NnzPSlc)}{(2 \times FbrPSlc + 2 \times NnzPSlc + 3) + ColCnt \times (FbrPSlc + NnzPSlc) + ColCnt} = \\
&= \frac{1}{2} \times \frac{ColCnt \times (FbrPSlc + NnzPSlc)}{2 \times (FbrPSlc + NnzPSlc + 1) + ColCnt \times (FbrPSlc + NnzPSlc + 1) + 1} = \\
&= \frac{1}{2} \times \frac{ColCnt \times (FbrPSlc + NnzPSlc)}{(FbrPSlc + NnzPSlc + 1) \times (ColCnt + 2) + 1}
\end{aligned}
\tag{3.10}
$$

According to the expression derived in 3.10, the characteristics of the slice that was assigned to the

thread influences the AI of said thread, much like in Kernel 3.3. The AI for all threads is also influenced by the number of columns in the matrices. Following the reasoning on the previous sections, the AI lower bound for this kernel can be expressed as follows:

$$\min(AI_{v2}) = \min\left(\frac{1}{2} \times \frac{ColCnt \times (FbrPSlc + NnzPSlc)}{(FbrPSlc + NnzPSlc + 1) \times (ColCnt + 2) + 1}\right) = \frac{1}{2} \times \frac{2}{10} = \frac{1}{10} \tag{3.11}$$

For the AI's upper bound the reasoning is also similar, so the number of columns in the matrices, the number of fibers and the number of non-zeros must all be large enough in order to verify the following equation:

$$\begin{aligned} \max(AI_{v2}) &= \max\left(\frac{1}{2} \times \frac{ColCnt \times (FbrPSlc + NnzPSlc)}{(FbrPSlc + NnzPSlc + 1) \times (ColCnt + 2) + 1}\right) = \\ &= \max\left(\frac{1}{2} \times \frac{ColCnt}{ColCnt + 2} \times \frac{FbrPSlc + NnzPSlc}{FbrPSlc + NnzPSlc + 1}\right) \approx \frac{1}{2} \times 1 \times 1 = \frac{1}{2} \end{aligned} \tag{3.12}$$

Similarly to the behaviour of the TTM kernels, both have the same minimum AI ($1/10$), however Kernel 3.4 has a 3$\times$ higher maximum ($1/2$ vs. $1/6$). It is once again important to reinforce that all calculations involving AI were done under the assumption that both all slices and fibers with no non-zero elements are not introduced in the computation. If such slices and/or fibers were to be introduced, the maximum AI would remain unchanged, but the minimum AI would drop down to zero.

3.2 Exploring Performance Upper-Bounds with Synthetic Tensors

A direct performance comparison between the previously elaborated data-parallel approaches, for TTM and MTTKRP, also depends on factors other than the kernels' AI ranges, such as the processing capabilities of the device in which the computations are performed (e.g., multi-core CPU or GPU), as well as on the characteristics of the sparse tensor dataset under evaluation. Therefore, the following study aims at describing the behaviour of the aforementioned kernels as well as uncovering their performance upper-bounds. For this purpose, we also construct a set of synthetic sparse tensors in such a way that the worst-case and best-case performance can be attained. These synthetic best-case and worst-case sparse tensors are constructed based on the AI, derived on 3.1 and 3.4 for TTM and on 3.7 and 3.10 for MTTKRP, as well as on the architectures of the CPU and GPU devices. All tests and measurements for this study were collected under Intel's Devcloud environment, the hardware setup used is depicted in Table 3.2.

Device	Model	Frequency	Cores
CPU	Intel Core i9-11900KB	3,30 GHz	16[1]
GPU	Intel 11th Gen UHD Graphics	1,45 GHz	32[2]

[1] Physical Cores $\times$ Concurrent Threads per Core $= 8 \times 2$
[2] Execution Units

Table 3.2: Hardware setup for study of general-purpose architectures

3.2.1 TTM Best-Case Performance Analysis

The challenges in most approaches to sparse TTM are load balancing and data locality. While the first challenge can to some extent be mitigated, both are mostly dependent on the specific features of the dataset used. For that reason, the prime candidate to achieve maximum performance with both Kernels 3.1 and 3.2 is to construct a semi-sparse tensor. The semi-sparse tensor is a tensor that is sparse in all its dimensions except for one.

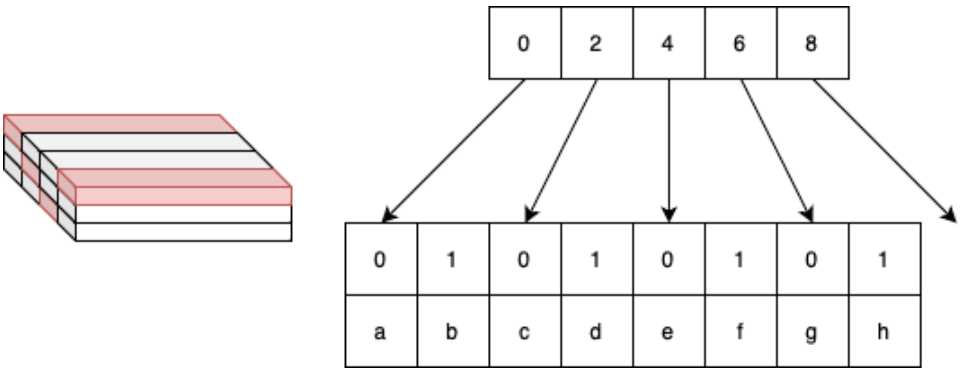


Figure 3.1: Semi-sparse tensor and CSF's representation of its fibers

This specific disposition, depicted in Figure 3.1, consists of a tensor with its horizontal mode dense, meaning all fibers of this mode are either empty or fully dense. In other words, there are several dense fibers sparsely scattered across the tensor.

The distribution of the non-zero elements in the proposed best-case synthetic sparse tensor allows for both load balancing, since all fibers have the same length, as well as data locality, since all fibers access consecutively the rows of the matrix. To construct the best-case scenario, there are three parameters that must be modeled: *i)* $FbrCnt$ – the number of fibers with non-zero elements, *ii)* $NnzPFbr$ – the number of non-zeros in each fiber and *iii)* $ColCnt$ – the number of columns in the matrix. It is important to notice that for this specific kind of tensor, since fibers are dense, the number of rows of the matrix is the same as the number of non-zero elements in each fiber.

Two more parameters can be deduced from three aforementioned parameters, i.e., the number of threads created and the matrix size, which may slightly differ depending on the kernel used. For Kernel 3.1, since every fiber-column dot product is assigned to a different thread, the number of threads created

is $FbrCnt \times ColCnt$. On the other hand, for Kernel 3.2, each thread computes one fiber against the whole matrix, therefore the number of threads created is $FbrCnt$. Finally, the size of the matrix is defined by $NnzPFbr \times ColCnt$.

3.2.1.A CPU Analysis

The CPU architecture under evaluation (Intel Core i9-11900KB) involves a memory hierarchy that includes a set of private (L1 and L2) and shared L3 caches. As such, there are different best-cases depending on which of the cache levels is being addressed. Since the fastest cache is L1, the best-case corresponds to the scenario when the matrix elements fit into L1. However, this requires the matrix to be very small which, according to 3.1 and 3.4, causes the AI to be very low. It is non-trivial to find the optimal trade-off between a cache fitting size and a high AI.

To find this optimal trade-off some empirical analysis is required, for which a starting point is required and must be chosen bearing in mind three crucial aspects: *i)* $FbrCnt$ should be large enough, such that the workload is significant and, therefore, does not influence the measures taken; *ii)* $NnzPFbr$ is large enough, such that the AI of the kernel is close to its maximum; and *iii)* $NnzPFbr \times ColCnt$ is small enough, such that the matrix fits in the desired cache level. Let the following parameters be considered as a starting point: $FbrCnt = 131072$ for both kernels. $NnzPFbr = 256$ for Kernel 3.1 and $NnzPFbr = 512$ for Kernel 3.2. $ColCnt = 64$ for Kernel 3.1 and $ColCnt = 256$ for Kernel 3.2.

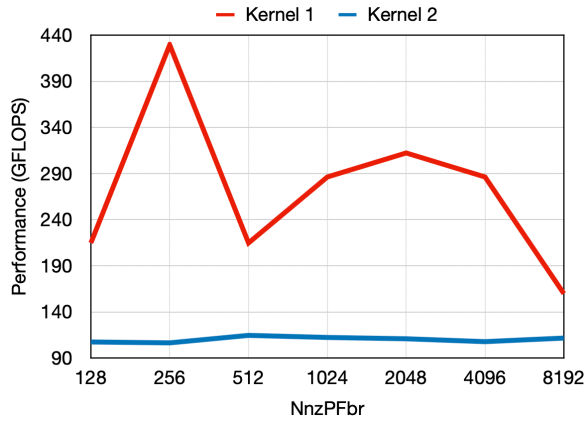


Figure 3.2: CPU best-case performance for different number of non-zero elements in the fiber

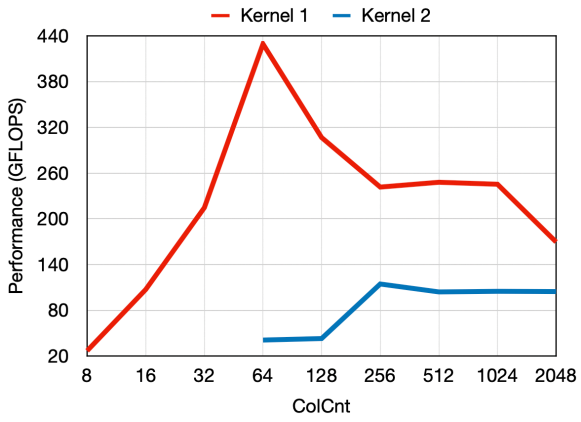


Figure 3.3: CPU best-case performance for different number of columns in the matrix

Figure 3.2 shows the behaviour of kernels 3.1 and 3.2 when changing just one of the parameters at a time. One can notice irregular spikes and drops of performance in the obtained experimental results for Kernel 3.1, while for Kernel 3.2 the performance does not change drastically. The spikes are explained by having a larger $NnzPFbr$, since it provides higher AI and therefore allows for better performance. On the other hand, the drops are explained by the cache levels in the architecture, i.e., whenever the matrix

becomes too large to fit in a cache level, the kernel becomes bound by the next level, which has lower bandwidth, thus making the TTM execution slower.

In Figure 3.3, Kernel 3.1 displays increasing performance until $ColCnt = 64$ and then the performance starts to decrease, since the dataset no longer fits in L1 cache. A similar behaviour can be observed for $ColCnt = 1024$, but for L2 cache. Kernel's 3.2 AI also depends on $ColCnt$, but its behaviour is similar, performance increases until $ColCnt = 256$ and then slowly decreases as the number of columns is growing.

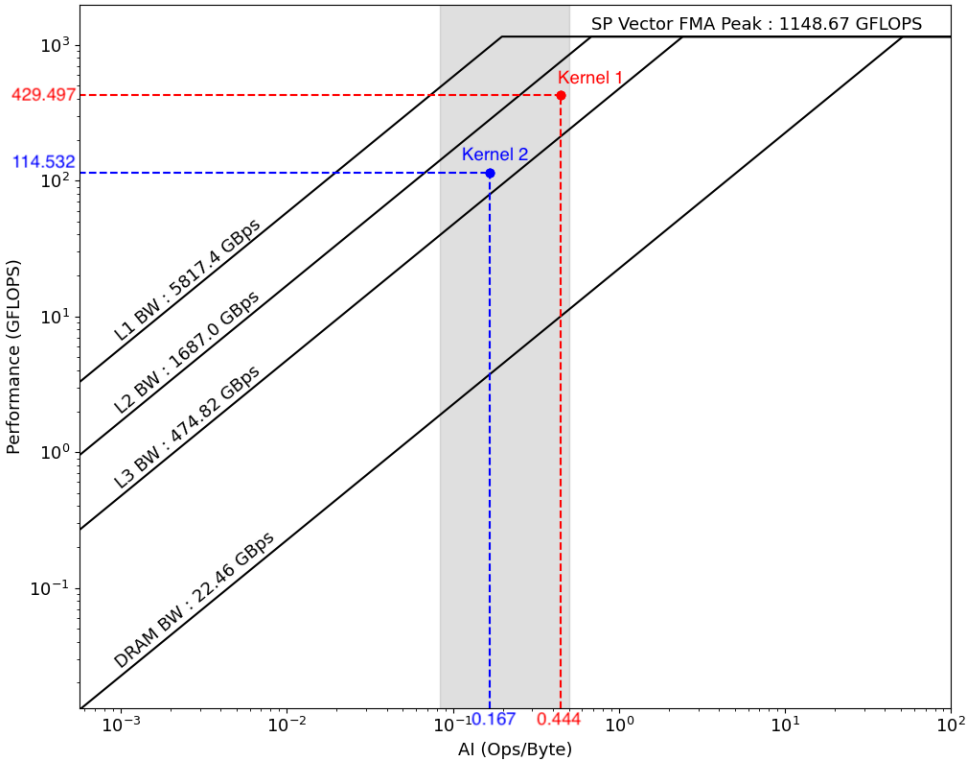


Figure 3.4: Roofline model for CPU best-case scenario with both kernels

Figure 3.4 provides the CARM characterization of kernels 3.1 and 3.2 for this specific CPU architecture. The gray zone represents the theoretical limits of AI calculated for Kernel 3.2. Kernel's 3.1 limits are also within these limits. It is possible to observe that due to compiler optimisations, namely vectorisation, Kernel 3.1 achieves higher performance and AI than Kernel 3.2.

3.2.1.B GPU Analysis

For the GPU, given the memory-bound nature of TTM kernels, the best-case exploration strategy aims at ensuring that the AI of kernels is as high as possible. In addition, the parallelism should be high in order to fully exploit the massively parallel GPU architecture, while the matrix should fit in the GPU L3 cache, to avoid additional loads from the DRAM.

For Kernel 3.1, this means making $NnzPFbr$ sufficiently large such that Equation 3.3 is verified, while ensuring $FbrCnt \times ColCnt$ large enough to keep the GPU units occupied and keeping $NnzPFbr \times ColCnt$ small enough to ensure that the dataset fits in the GPU L3 cache. A similar rationale is followed for Kernel 3.2, where sufficiently large $NnzPFbr$ and $ColCnt$ should be provisioned to satisfy Equation 3.6, while large enough $FbrCnt$ is required to maximize the occupancy of the GPU units, as well as small enough $NnzPFbr \times ColCnt$ to guarantee that the dataset fits in the L3 cache. As in the CPU case, a starting point is required to be chosen according to these restriction, which is in this case set to: $FbrCnt = 131072$ which is enough to fully utilize the GPU's compute units, $NnzPFbr = 512$ which ensures AI very close to the theoretical maximum and $ColCnt = 64$.

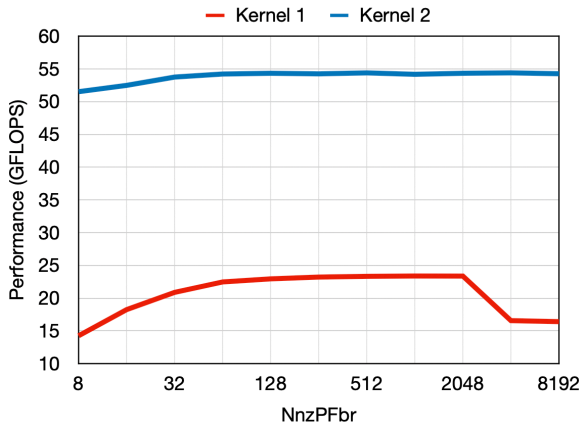


Figure 3.5: GPU best-case performance for different number of non-zero elements in the fiber

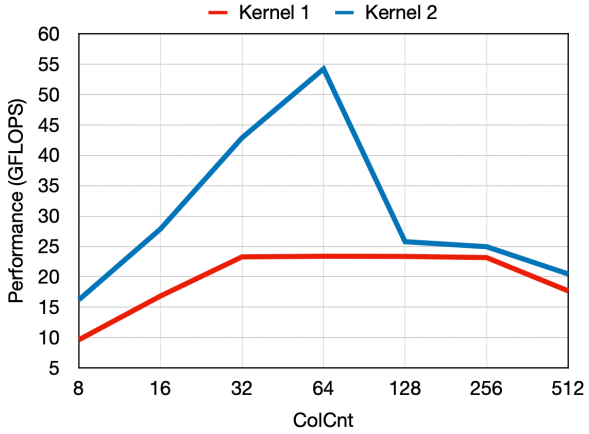


Figure 3.6: GPU best-case performance for different number of columns in the matrix

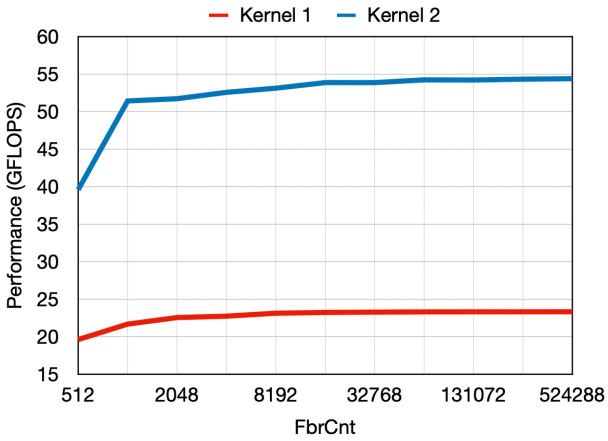


Figure 3.7: GPU best-case performance for different number of fibers with non-zero elements

Figures 3.5, 3.6 and 3.7 depict the behaviour of Kernel 3.1 on the tested GPU devices for different ranges of parameters on both kernels. In Figure 3.5, the performance increases as the length of the fibers is increasing and starts stabilising around $NnzPFbr = 256$. However, when $NnzPFbr$ is in-

creased even further performance starts to drop drastically for Kernel 3.1. The reason for this behaviour lies in the fact that the matrix size is directly proportional to the $NnzPFbr$ parameter, thus beyond the $NnzPFbr = 2048$ the matrix does not fit in GPU L3 cache. Such phenomenon is not observed for Kernel 3.2, since each thread has a fiber assigned and therefore it streams through the tensor fibers. In Figure 3.6, the behaviour is very similar to the one observed in Figure 3.5. Performance increases until $ColCnt = 32$ for Kernel 3.1 and $ColCnt = 64$ for Kernel 3.2. Where it maintains stable until $ColCnt = 256$ for Kernel 3.1. After which any further increase in matrix size results in reduced performance for both kernels. In Figure 3.7, one can observe that the performance increases as the number of fibers is increasing and starts stabilising around $FbrCnt \geq 65536$, signalling that the GPU units are fully occupied. As previously exposed, the fiber-centric TTM approach in Kernel 3.2 creates less threads, concentrating more workload in each thread, thus lading to having more factors affecting its AI and making its behaviour more irregular.

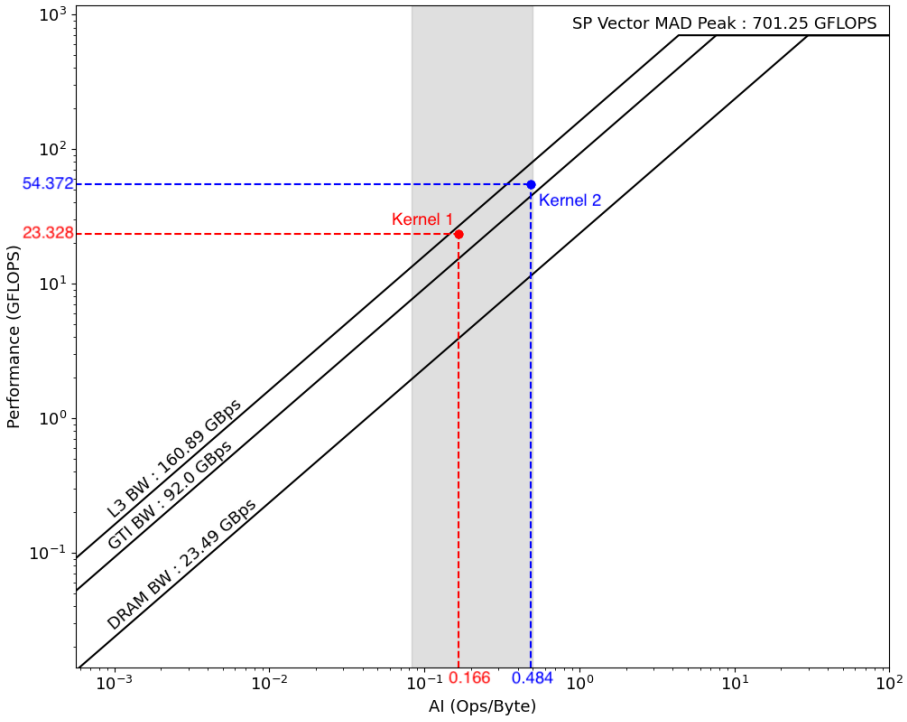


Figure 3.8: Roofline model for GPU best-case scenario with both kernels

Finally, in Figure 3.8, the roofline model for the architecture is presented and it is possible to observe that the performance is very close to the maximum achievable by the device for the corresponding AI in both kernels.

3.2.2 TTM Worst Case Performance Analysis

Following a similar reasoning as for the best-case, it is also possible to determine the worst-case TTM processing scenario, in order the uncover the lower bounds on the performance attainable with the proposed TTM kernels on both CPU and GPU architectures. The strategy followed herein aims at analysing the worst-case scenario under the condition that the full utilization of processing resources is attained with a data distribution in the specifically created synthetic sparse tensor that hinders performance.

For this purpose, the data distribution can be modeled with four parameters: *i)* $FbrCnt$ – the number of fibers with non-zero elements; *ii)* $NnzPFbr$ – the number of non-zeros in each fiber; *iii)* $ColCnt$ – the number of columns in the matrix, and *iv)* $RowCnt$ – the number of rows in the matrix. It is important to notice that, unlike what happens in the semi-sparse tensor, the number of non-zero elements in the fiber do not match the number of rows in the matrix. Instead, $RowCnt$ is now used to denote the effective size of the fiber, which is always greater than $NnzPFbr$.

Two more parameters can be deduced from these four parameters, which slightly differ depending on the kernel used. For Kernel 3.1, since every fiber-column dot product is assigned to a different thread, the number of threads created is equal to $FbrCnt \times ColCnt$. On the other hand, for Kernel 3.2, each thread computes one fiber against the whole matrix, therefore the number of threads created is equal to $FbrCnt$. Finally, the size of the matrix is defined by $RowCnt \times ColCnt$.

To facilitate the comparison of the results obtained with this scenario to the previously presented ones, both $FbrCnt$ and $ColCnt$ were kept the same, while only $RowCnt$ was varied. This way, the workload is always kept at about the same size, with the only difference being the distribution of the non-zero elements across the fibers.

3.2.2.A CPU Analysis

For the CPU, instead targeting specific cache levels, the idea behind the worst-case scenarios is to prevent any data reuse in the cache hierarchy, thus limiting the kernel performance to the lowest bandwidth available in the CPU memory hierarchy, i.e., DRAM.

To further accentuate the worst-case performance scenario, a lower AI is also desirable. For Kernel 3.1, this means making $NnzPFbr = 1$ such that Equation 3.2 is verified, while keeping $RowCnt \times ColCnt$ large enough to ensure that the tensor data does not fit in any cache level. For Kernel 3.2, both $NnzPFbr$ and $ColCnt$ should be equal to one in order to satisfy Equation 3.5, while keeping $RowCnt \times ColCnt$ large enough, such that the dataset does not fit in any cache level.

Figure 3.9 provides one possible representation of a synthetic worst-case sparse tensor with long fibers, each with a single non-zero element. It is worth noting that the non-zero elements are displaced across fibers in such a way that they do not allow for any reuse of the matrix elements. For the CPU used to test and model the parameters of the proposed approach setting $RowCnt = 524288$ proved to

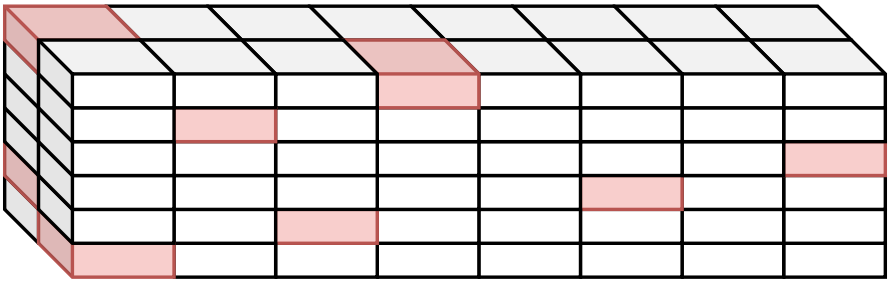


Figure 3.9: Depiction of worst case tensor

be large enough to display a minimal performance.

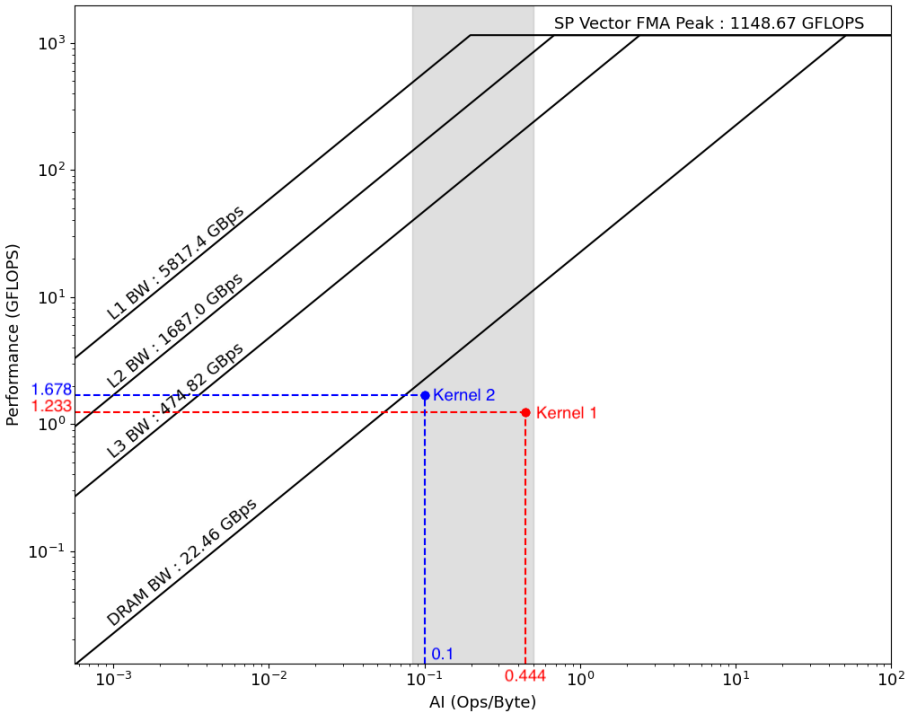


Figure 3.10: Roofline model for CPU worst case scenario with both kernels

Figure 3.10 represents the roofline model for kernels 3.1 and 3.2 for Intel Core i9-11900KB CPU under the worst-case performance scenarios. In both cases, it is possible to observe that the objective of being outside of the cache levels is achieved, since both kernels are positioned below the respective DRAM roof. It is possible to observe that due to compiler optimisations, namely vectorisation, Kernel 3.1 achieves higher AI than kernel 3.2.

3.2.2.B GPU Analysis

In order to exercise the worst-case performance scenario on the GPU architecture, it is necessary to construct the synthetic sparse tensors that allow for the kernels' AI to be as low as possible, as TTM is memory bound on the GPU, while also ensuring that the matrix does not fit in the GPU L3 cache, thus enforcing constant loads from the DRAM.

To create the worst-case synthetic sparse tensor for Kernel 3.1, it is needed to ensure $NnzPFbr = 1$ such that Equation 3.2 is verified, while keeping $RowCnt \times ColCnt$ large enough such that the matrix does not fit in GPU L3 cache. For Kernel 3.2, both $NnzPFbr$ and $ColCnt$ should be equal to one such that Equation 3.5 is satisfied, while keeping $RowCnt \times ColCnt$ large enough to prevent reuse of matrix elements in GPU L3 cache.

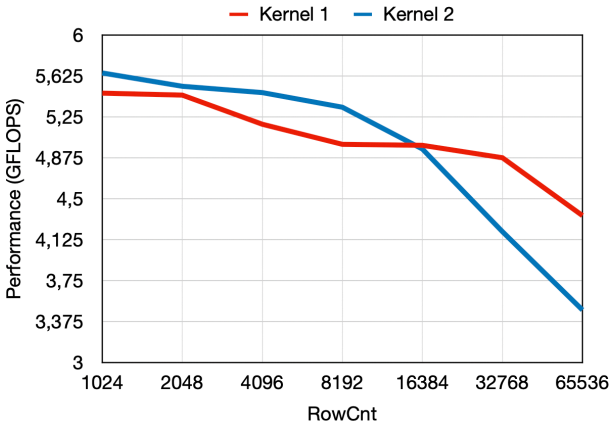


Figure 3.11: GPU worst case performance for different number of rows in the matrix

Figure 3.11 presents the performance variation of both kernels with respect to the different $RowCnt$ values. As it can be observed in Figure 3.11, the performance is increasing with the increase of $RowCnt$. This happens because the size of the matrix is increasing, thus forcing the GPU to load data from outside the GPU L3 cache.

Figure 3.12 represents the roofline model for kernels 3.1 and 3.2 for this specific architecture under the worst-case evaluation scenario. In both cases, it is possible to observe that the objective of being outside of the GPU L3 cache is achieved. However, it is important to notice that, since $ColCnt$ was kept at 64 columns (necessary to achieve a fair comparison), the AI of Kernel 3.2 is higher than the minimum.

3.2.3 MTTKRP Best-Case Performance Analysis

Load balancing and data locality are, arguably, the greatest challenges in most approaches to sparse MTTKRP. While there are attempts to mitigate these issues, they mostly depend on specific features of the datasets used. Therefore a synthetic tensor that allows a MTTKRP kernel to achieve its best

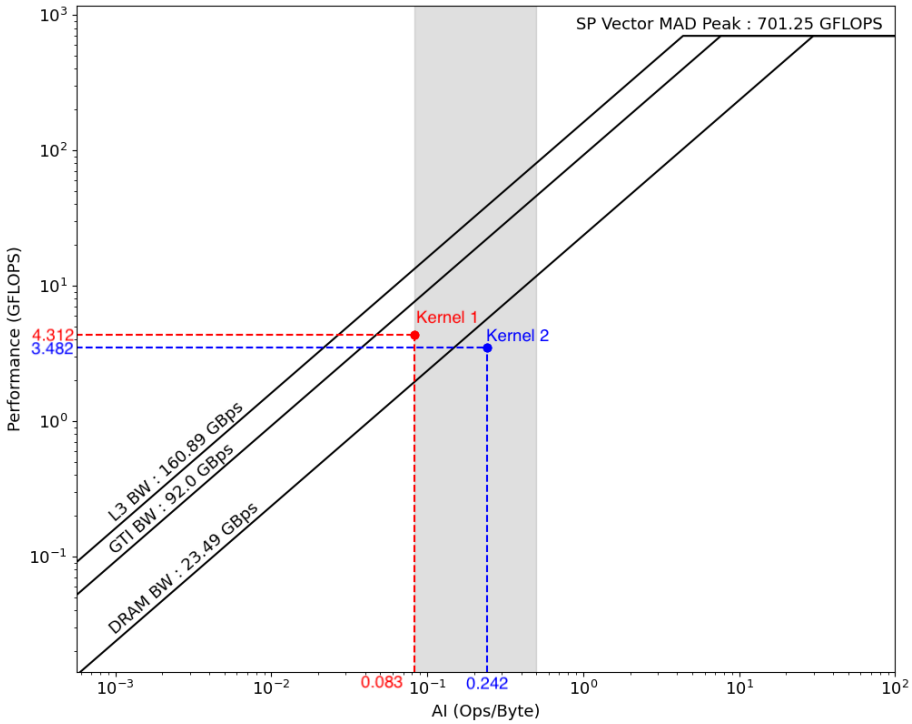


Figure 3.12: Roofline model for GPU worst case scenario with both kernels

performance is a tensor whose characteristics facilitate the achievement of load balancing and data locality. For that reason the prime candidate to achieve maximum performance with both kernels 3.3 and 3.4 is a tensor which consists of a set of dense slices sparsely spread out. Similar to the semi-sparse tensor presented for the best-case performance in TTM processing but now with dense slices instead of fibers. Naturally, it is arguable to which extent such tensor can be considered sparse, however, since the purpose of this section is to determine the upper bounds of our MTTKRP kernels, such considerations are not relevant.

3.2.3.A CPU Analysis

Modern CPUs architecture possess a memory hierarchy that includes a set of private (L1 and L2) and shared L3 caches. As such, depending on which of the cache levels is being targeted the best-case performance varies. L1 is the smallest and fastest memory in the hierarchy, therefore it is the one that can provide the best performance, namely when the matrices fit in it. However, this requires the matrices to be very small which, according to 3.7 and 3.10, causes the AI to be very low.

In order to help finding an optimal solution between a cache fitting size and a high AI an empirical analysis is required, as it is non-trivial to solve such trade-off. After some experimentation, we settled on $SlcCnt = 32768$ for both kernels. We also decide on $FbrCnt = 524288$ and $NnzCnt = 8388608$ for Kernel 3.3 and $FbrCnt = 2097152$ and $NnzCnt = 134217728$ for Kernel 3.4. The only parameter that

still needs to be set is $ColCnt$.

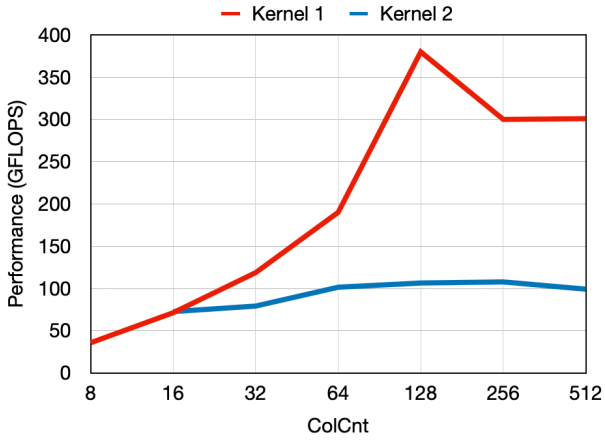


Figure 3.13: CPU best-case performance for varying number of columns

Figure 3.13 shows the behaviour of kernels 3.3 and 3.4 when changing just that one parameter. For Kernel 3.3, performance increases with the increase in number of columns, specially from $ColCnt = 16$ as the compiler starts vectorising the kernel loops. As of $ColCnt = 256$, the performance drops as the size of matrices is large enough to start being affected by the cache size and the loads from the tensor, which pollute the cache. For Kernel 3.4, performance also increases with the increase in number of columns, however, due to the compiler never vectorising its loops, it ends up never reaching as high performance as Kernel 1.

Figure 3.14 provides the CARM characterisation of Kernels 3.3 and 3.4 for this specific CPU architecture. The gray zone represents the theoretical delimits the attainable AI by Kernel 3.2, as the range of Kernel's 3.3 is a subset of it. From the Figure, it is possible to observe that, even though both kernels are reusing their matrices from L1 cache, their performance is close to the L2 cache roof. The reason behind this relates with the loads from the tensor, namely the fiber and element indexes as well as the values. These are never reused as each element is only computed once, therefore they are bounded by DRAM bandwidth which impacts the overall performance.

3.2.3.B GPU Analysis

For the GPU analysis, given a memory-bound nature of both our MTTKRP kernels, ensuring that the AI of kernels is as high as possible is crucial to achieve the best-case performance. Besides achieving a high AI, in order to fully exploit the massively parallel architecture of the GPU, parallelism must be high, while the matrices still fit in the GPU L3 cache, to avoid additional loads from the DRAM.

For Kernel 3.3, this means making $FbrPSlc + NnzPSlc$ sufficiently large such that Equation 3.9 is verified, while ensuring that the $SlcCnt \times ColCnt$ threads are in quantity enough to keep the GPU units occupied. Also for this specific tensor $FbrPSlc$ and $NnzPFbr$ are the number of rows for matrix1 and

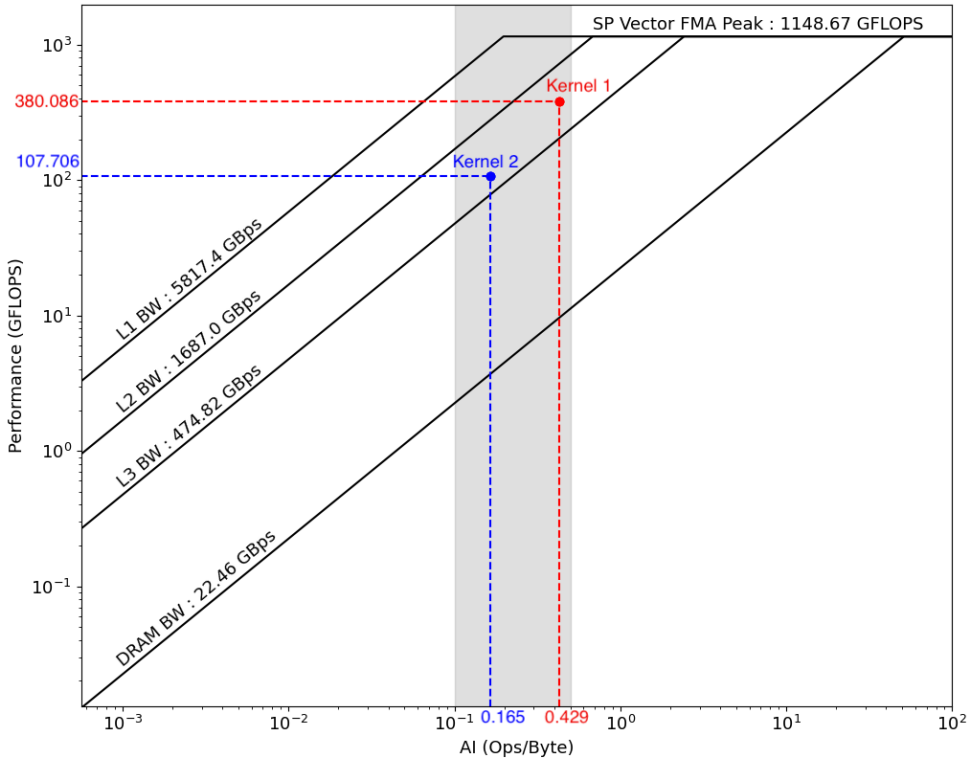


Figure 3.14: Roofline model for CPU best-case scenario with both kernels

matrix2 respectively, therefore ensuring that their product with the number of columns is small enough to fit in the GPU's L3 cache is also fundamental. For Kernel 3.4, a similar rationale is followed. However, to satisfy Equation 3.12, $ColCnt$ must also be as high as possible and the parallelism is only dependent on the number of slices. Having these thoughts in mind and after some empirical experimentation, we arrived at $SlcCnt = 32768$, $FbrCnt = 2097152$, $NnzCnt = 134217728$ and $ColCnt = 32$ which proved to be the combination that provided the best performance for both kernels.

In Figure 3.15, the roofline model for the architecture is presented and it is possible to observe that the performance is very close to the maximum achievable by the device for the corresponding AI in both kernels.

3.2.4 MTTKRP Worst Case Performance Analysis

In order the uncover the lower bounds on the performance attainable with the proposed MTTKRP kernels on both CPU and GPU, it is also possible to determine the worst-case MTTKRP processing scenario by following a similar reasoning as for the best-case. To achieve meaningful results, we aim at analysing the worst case scenario under the condition that the full utilization of processing resources is attained with a data distribution in the specifically created synthetic sparse tensor that hinders performance.

Besides ensuring full device occupation to facilitate the comparison of the results obtained with this

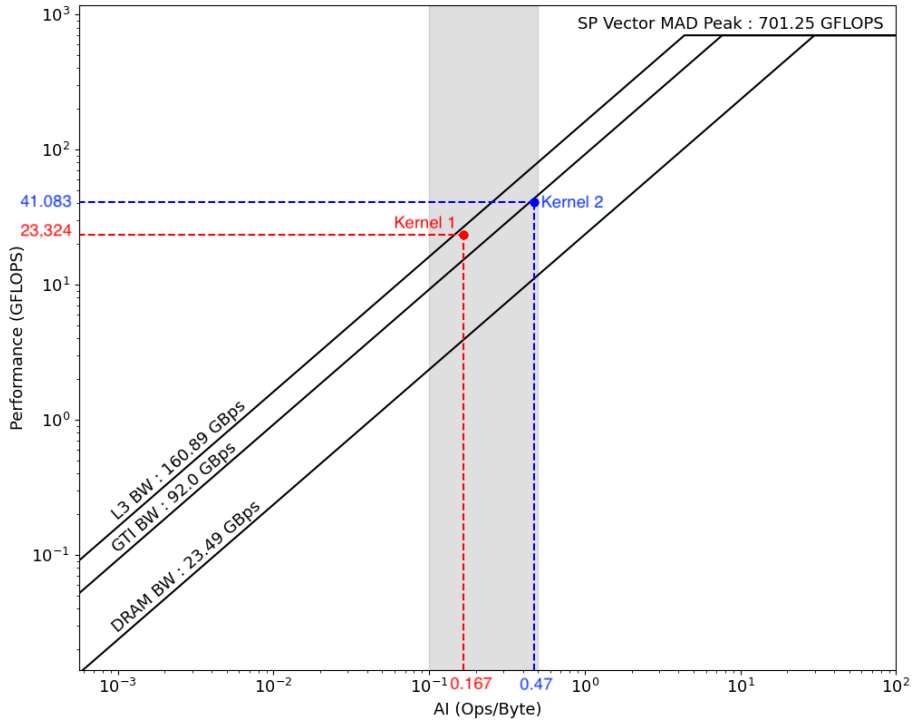


Figure 3.15: Roofline model for GPU best-case scenario with both kernels

scenario to the previously presented ones, we also kept both $SlcCnt$ and $ColCnt$ the same. So the differences are that now each slide has only one fiber with non-zero elements and each fiber has only one non-zero element, meaning $SlcCnt = FbrCnt = NnzCnt$. It is important to notice that, unlike in the best-case, for this tensor the number of rows is not the same as $FbrPSlc$ for matrix1 and $NnzPSlc$ for matrix2. This way, the workload is always kept at about the same size, with the only difference being the distribution of the non-zero elements across the fibers.

3.2.4.A CPU Analysis

To achieve the worst possible performance on the CPU, one should prevent data reuse in any cache level, hence limiting the kernel performance to the bandwidth of the slowest memory available on the CPU, i.e., DRAM.

To further accentuate the impact of being bound by such memory, a lower AI is also desirable. For both kernels, this means making $SlcCnt = FbrCnt = NnzCnt$, which in term leads to $FbrPSlc = NnzPFbr = 1$. For Kernel 3.4, to ensure minimum AI, it is also necessary to have $ColCnt$ as low as possible. Other important considerations are ensuring that both matrices are large enough to ensure that the tensor data does not fit in any cache level. For simplicity, we set the number of rows in both matrices to $RowCnt$.

Increasing number of rows in the matrices has the effect displayed on Figure 3.16. As can be

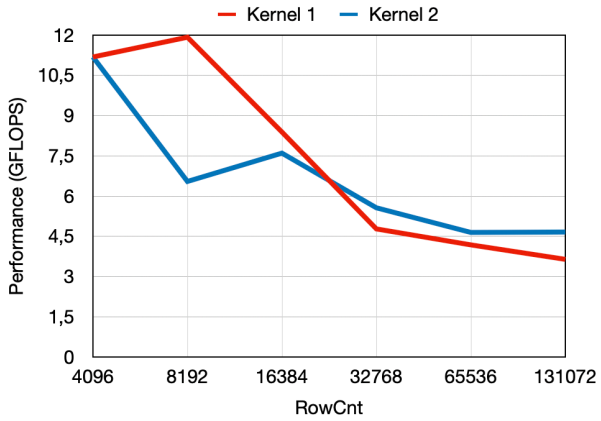


Figure 3.16: CPU worst-case performance for different number of rows in the matrices

observed, performance gradually drops, especially from $RowCnt = 16384$ as it is from that moment that the size of the matrices becomes to great for any cache level to fit, therefore making the kernels bounded by the bandwidth of the DRAM.

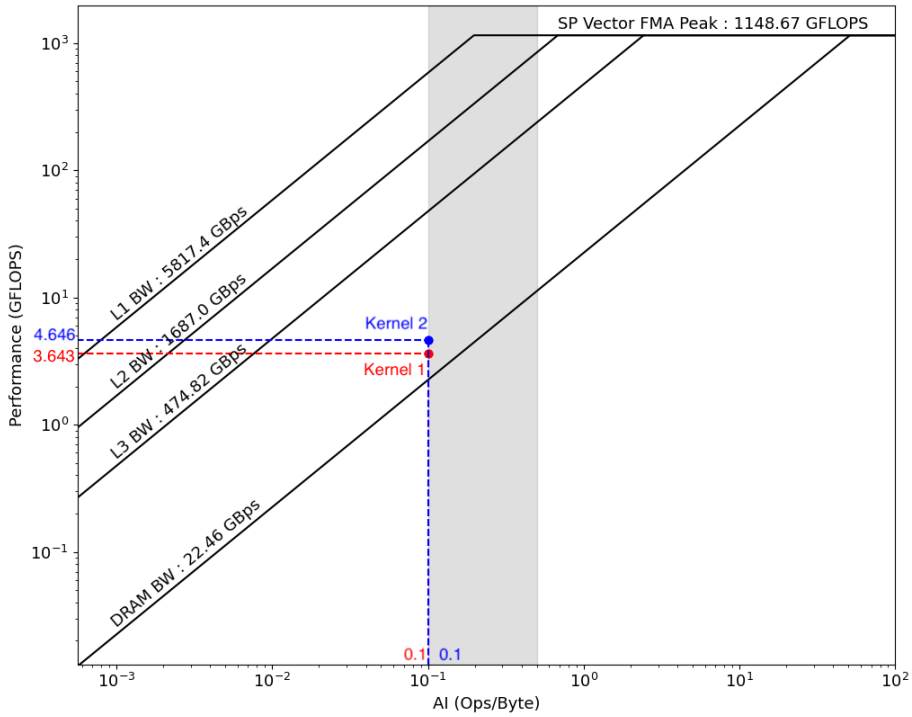


Figure 3.17: Roofline model for CPU worst-case scenario with both kernels

Figure 3.17 represents the roofline model for kernels 3.3 and 3.4 under the worst-case performance scenarios. For this method, performance is above the DRAM roof due to the limitations imposed when defining the parameters, namely the number of slices as well as the number of columns in the matrices. However, it is still possible to observe that the performance achieved is closer to the DRAM roof than it

is to the L3 roof and that the AI is at its minimum for both kernels.

3.2.4.B GPU Analysis

Since MTTKRP is memory bound on the GPU, in order to exercise the worst-case performance scenario on the GPU, it is necessary to construct a synthetic sparse tensor that allows for the kernels' AI to be as low as possible, while also ensuring that the matrices are large enough in order to force constant loads from the DRAM.

To create the worst-case synthetic sparse tensor for both kernels, it is necessary to experimented with several combinations of number of rows for the matrices, in order to reduce the search options, we decided that both matrices would have the same number of rows.

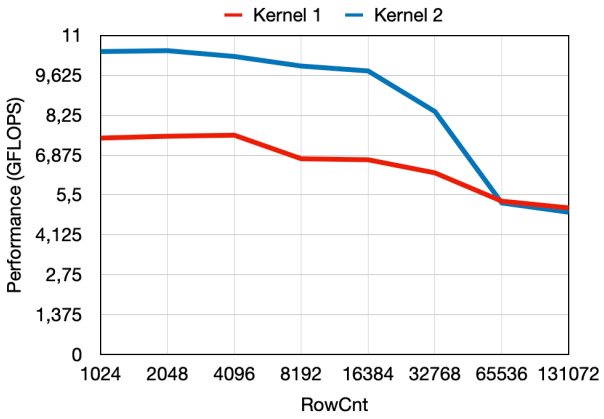


Figure 3.18: GPU worst-case performance for different number of rows in the matrices

Figure 3.18 portrays the performance of both kernels for an increasing number of rows in the matrices. As can be observed, performance gradually drops as would be expected, since the lack of reuse causes most of the data to be streamed from the DRAM, the increasing size of the matrices forces the GPU to load data from outside the GPU L3 cache.

Figure 3.19 displays the roofline model for the both kernels in the GPU architecture, where it is possible to observe that the performance is very close to bandwidth of the slowest memory on the device. It is also important to notice that due to the limitations imposed when generating the worst-case synthetic tensor, the performance of Kernel 3.3 is slightly above the DRAM bandwidth and the AI of Kernel 3.4 is higher than the minimum the kernel can achieve.

3.3 Heterogeneous Approach

In order to take full advantage of the system available and described in Table 3.2, we also developed and tested an heterogeneous solution for TTM and MTTKRP. Since we already have two kernels for

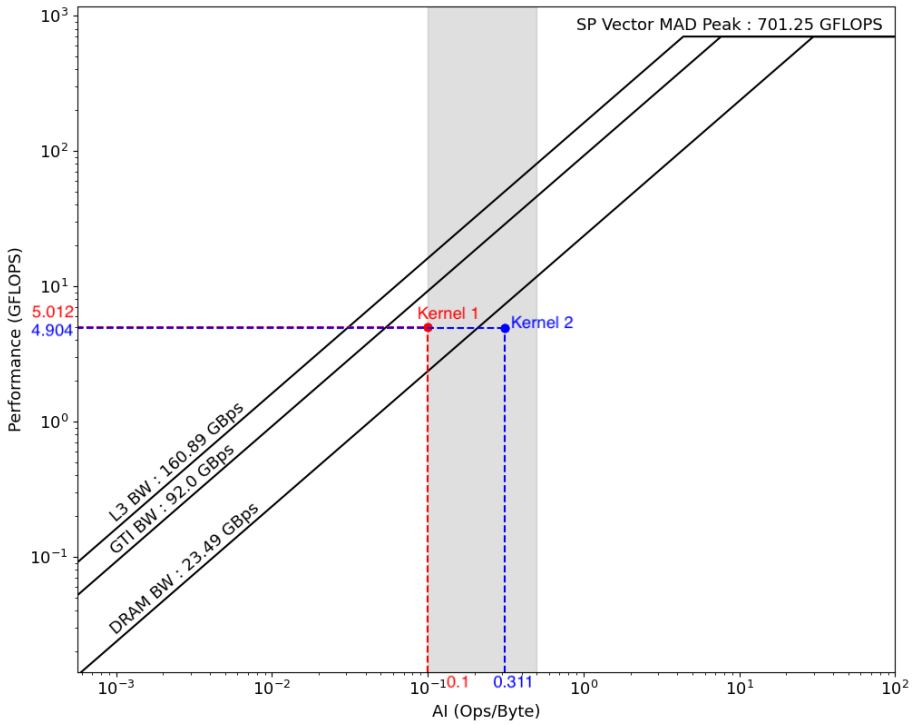


Figure 3.19: Roofline model for GPU worst-case scenario with both kernels

each method, the greatest challenge remaining in developing an heterogeneous kernel is to decide on how to split the workload across the targeted architectures.

Our first approach was to do a pure static workload distribution of the fibers and slices for TTM and MTTKRP respectively. This can be achieved by creating multiple SYCL queues, one for each device, and then submitting the corresponding fibers or slices to each one. Naturally, this approach comes with flaws, such as lack load balance which can be due to the lack of prior knowledge over the architectures present in the system, as well as the dataset's characteristics in general.

For TTM, the number of fibers assigned to each architecture must be in the proportion of its processing capabilities when compared to processing capabilities of the remaining. Also since the number of elements to process in each fiber may be different, it is also important to ensure that the number of non-zero elements assigned to each architecture follow a similar proportion as the fibers they are present in. For MTTKRP, the reasoning is the same but with an added layer of depth, as now besides the slice distribution, the number of fibers and non-zero elements in the slice must be considered.

Bearing these flaws in mind, we developed a different static approach. This approached aimed at probing the system to estimate the optimal proportions for the workload distribution and only then actually distributes the workload, hence herein called adaptive approach. To achieve such behaviour, we start by separating the initial dataset into two parts, one consisting of around 10% of the data and the other with the remaining. The smaller portion is divided equally between CPU and GPU, with the

execution time on each device being used to define said proportion. The other part, consisting of around 90% of the data, is then separated into two different sized parts based on the computed proportion.

Figure 3.20: Step-by-step example of Adaptive Approach

Figure 3.20 displays a step-by-step example of the distribution procedure executed by our adaptive approach. Relating the the previous explanation with the Figure, one can see that from step 0 to step one the data is separated in two parts. In TTM the data is the fibers, therefore in a tensor with one hundred thousand fibers, the two parts would consist of approximately ten thousand fibers and ninety thousand fibers respectively. For MTTKRP, the reasoning is the same but with slices.

The main disadvantage in this approach is that requires the first 10% to be processed serially in relation to the remaining 90%. When compared to the purely static approach, this tends to be better when the proportion of the previous is far from the actual computational capabilities of the architectures in the system, and tends to be worst when these match. Also for both versions there can be still some issues with the workload distribution.

For TTM, even if the number of fibers is well distributed, the number of non-zeros in the each fiber can differ, therefore it may happen that the workloads do not completely follow the proportion. For MTTKRP, the same can happen but even worse, because now not only the number of non-zero elements on the slice matters, but also the number of fibers on the slice. This could be avoided with resort to dynamic assignment.

3.4 Summary

This chapter started by introducing our TTM and MTTKRP implementation for general-purpose architectures. For each of the methods, two kernels were presented and for each of the kernels, the AI was derived. From the derived AI expressions, the upper and lower-bounds of the kernels' AI were also presented. With resort to synthetic datasets that exacerbate the characteristics of both the kernels and the architectures, the performance limits were explored and discussed. Finally, we thoroughly explaining our heterogeneous approaches.

Chapter 4

Sparse Tensor Processing on Specialised Architectures

Unlike the CPU and GPU, the FPGA's architecture is not fixed, it is instead defined depending on the algorithm that is targeting it. Therefore, the approach must be different for these architectures. This analysis consists of theoretically measuring the AI and peak performance on the FPGA. With this analysis and the concepts of roofline models, described in section 2.6, it is expected to confirm the bottlenecks and test the limits of utilization on the architecture. Repeating the notation defined in Chapter 3, let us resort to Table 4.1:

$SlcCnt$	total number of slices in the tensor
$FbrCnt$	total number of fibers in the tensor
$NnzCnt$	total number of non-zero elements in the tensor
$ColCnt$	total number of columns in the matrices
$FbrPSlc$	number of fibers in a slice
$NnzPSlc$	number of non-zero elements in a slice
$NnzPFbr$	number of non-zero elements in a given fiber

Table 4.1: Notation used throughout this chapter

It is important to notice that the FPGA does not create threads for processing, therefore the AI does not depend on the data distribution, but instead on the characteristics of the whole tensor. Furthermore, the FPGA's peak performance can be derived from the number of DSPs present in the device.

Device	Model	Frequency	ALMs	Registers	DSPs
FPGA	Intel Arria 10 GX 1150	450 MHz	427 200	1 708 800	1518

Table 4.2: Hardware setup for study of specialised architectures

Table 4.2 refers to the FPGA used for this analysis. From the Table, it is possible to infer the FPGA's

peak performance. With 1518 DSPs available, each capable of a MAC every cycle, meaning two operations, the FPGA is capable of 3036 single-precison floating-point operations every cycle. Considering a frequency of 450 MHz, the theorectical peak performance is of 1.366 TFLOPS. Note that for the peak performance the remaining resources of the FPGA, specially the soft-logic, were left out. Depending on the design and on the operation being executed, the operating frequency and percentage of available resources for the actual design may change, hence, in order to achieve a fair comparison they are not considered. Naturally, each design uses different number of DSPs for one Processing Element (PE) and therefore the peak performance of the design itself may not be the same as the peak performance of the FPGA. For each developed design, we analyse the attainable peak performance of that design.

4.1 Tensor Times Matrix (TTM)

The first approach taken, in order to create a design that would efficiently process TTM on the FPGA, was to create as many PEs as columns in the matrix. Then each PE would compute all fibers against their respective column. This approach lead to poor spatial and temporal locality in data accesses which hindered the performance. However such design provided a much needed insight on the fundamentals of FPGA design and optimisation with SYCL. After completely revamping the original design, we arrived at the core component in TTM processing on the FPGA.

```
1  event e { q.submit([&](handler &h) {
2      h.single_task([=]() [[intel::kernel_args_restrict]] {
3          float tmp[colCnt];
4
5          // Iterate over all fibers with non-zero elements
6          for (auto fbr { 0 }; fbr < fbrCnt; ++fbr) {
7
8              #pragma unroll
9              for (auto col { 0 }; col < colCnt; ++col) {
10                 tmp[col] = 0.0f;
11             }
12
13             // Load fiber boundaries: accFbrPtr[fbr] and accFbrPtr[fbr+1]
14             for (auto ele { accFbrPtr[fbr] }; ele < accFbrPtr[fbr+1]; ++ele) {
15                 // Load in-fiber index and value
16                 const auto k { (accKIdx[ele] - 1) * colCnt };
17                 const auto val { accValues[ele] };
```

```
18
19                  // Load corresponding row of matrix
20                  // Compute product and accumulate
21                  #pragma unroll
22                  for (auto col { 0 }; col < colCnt; ++col) {
23                      tmp[col] += val * accMatrix[k + col];
24                  }
25              }
26
27              // Store output fiber to global memory
28              #pragma unroll
29              for (auto col { 0 }; col < colCnt; ++col) {
30                  accOutput[fbr * colCnt + col] = tmp[col];
31              }
32          }
33      });
34  }) };
```

Listing 4.1: TTM Kernel for FPGA

Kernel 4.1 generates a design that processes TTM similarly to how the kernels for general-purpose architectures did. For each fiber in the tensor there are two loads for the fiber boundaries (line 14), then for each element in the fiber there are two more loads for the non-zero element's index and value (lines 16-17). Finally, a row of the matrix is loaded and computed against the element (lines 21-24). When all elements of the tensor's fiber have been computed the output fiber is stored to global memory (lines 28-31).

In Figure 4.1, a general diagram of the design created on the FPGA is depicted. By analysing it along side Kernel 4.1, it is possible to confirm the accuracy of the diagram and to have a more visual perspective of the design. The start block refers to the initialization of the variable fbr on line 6. Then the condition to check whether there are more fibers on the same line, followed by the load of the fiber's boundaries and the condition to check whether there are more non-zero elements on line 14. If so the non-zero element is processed, the processing of the element is shown in further detail and consists of loading both the in-fiber index and value for the element. These can be done in parallel since they are independent and correspond to lines 16-17. Then a row of the matrix is loaded, this has to be sequential in relation to the in-fiber index as it is the index that refers which row must be loaded. Both the value and the row are fed to an array of DSPs that perform a MAC. This corresponds to lines 21-24 and it is important to notice that, due to the pragmas used in the kernel, it is possible to load all elements of the row simultaneously as well as performing the MAC of the non-zero element against the entire row

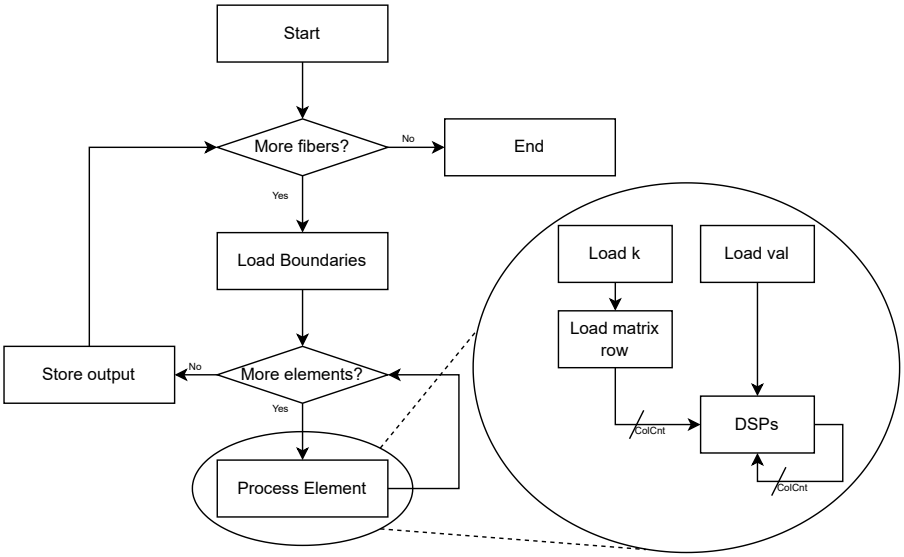


Figure 4.1: Diagram of PE

simultaneously. Finally, when there are no more non-zeros to process in the fiber, the output is stored to global memory, this part of the diagram corresponds to lines 28-31.

The main bottlenecks in this Kernel are the loads, specifically the ones done from the matrix. This happens because for each non-zero element a full row is loaded, however the FPGA does not have a cache hierarchy like the previous architectures and so, even if the same row is requested by one of the following elements, the FPGA will always have to load that same row again. In order to face this challenge, we resorted to the FPGA's on-chip memory. By loading the whole matrix to this faster memory first, it is possible to reduce the average access time to data on the matrix.

```
1 float localMatrix[colCnt];
2
3 #pragma unroll 16
4 for (auto ele { 0 }; ele < dim2 * colCnt; ++ele) {
5     localMatrix[ele] = accMatrix[ele];
6 }
```

Listing 4.2: TTM add-on for matrix pre-load on FPGA

This approach is achieved by introducing the code depicted in 4.2 before line 6 on Kernel 4.1. However it comes with a trade-off, as matrices tend to be very large, the amount of resources used is greater which causes the design to either not fit in some FPGAs or to be constrained in the maximum amount of columns the matrix have. It is also important to notice that the unroll factor may be adjusted depending on the resources available.

Now to analyse the AI of this design for a generic tensor. Assuming all data types are 4-byte wide, it can be expressed as follows:

$$AI_{sp} = \frac{1}{4} \times \frac{2 \times NnzCnt \times ColCnt}{FbrCnt + 1 + (2 \times NnzCnt) + (NnzCnt \times ColCnt) + (FbrCnt \times ColCnt)} = \\ = \frac{1}{2} \times \frac{NnzCnt \times ColCnt}{FbrCnt \times (ColCnt + 1) + NnzCnt \times (ColCnt + 2) + 1} \tag{4.1}$$

It is still possible to determine a minimum and maximum AI, but from the tensor's characteristics. The minimum AI can be obtained when $NnzCnt$ and $ColCnt$ are at their minimum and $FbrCnt$ is at its maximum. For the first two, the minimum is one, meaning one non-zero element in the tensor and a matrix with one column, therefore a vector. As for the latter, since only fibers with at least non-zero element are stored, the maximum is as many fibers as non-zero elements, which in this case is also one. Thus, the design's minimum AI can be expressed as follows:

$$\min(AI_{sp}) = \min\left(\frac{1}{2} \times \frac{NnzCnt \times ColCnt}{FbrCnt \times (ColCnt + 1) + NnzCnt \times (ColCnt + 2) + 1}\right) = \frac{1}{12} \tag{4.2}$$

Consequently, the maximum AI can be computed in a similar manner. In this case both $NnzCnt$ and $ColCnt$ must be as large as possible in order to have the design perform the most computations possible, while $FbrCnt$ must be as little as possible in order to decrease the number of loads, meaning one fiber with all non-zero elements in it. The following equation derives the design's AI upper bound:

$$\max(AI_{sp}) = \max\left(\frac{1}{2} \times \frac{NnzCnt \times ColCnt}{FbrCnt \times (ColCnt + 1) + NnzCnt \times (ColCnt + 2) + 1}\right) = \\ = \max\left(\frac{1}{2} \times \frac{ColCnt}{ColCnt + 2} \times \frac{NnzCnt}{NnzCnt + 1}\right) \approx \frac{1}{2} \tag{4.3}$$

Finally, we analyse the maximum attainable performance by this PE and and how it compares to the total FPGA peak performance. This design uses one DSP for every column on the matrix, therefore one PE performs $2 \times ColCnt$ operations per cycle. From Table 4.2, we also know the maximum frequency at which the DSPs can operate, therefore the peak performance of one PE is $0.9 \times ColCnt$ GFLOPS. Considering the FPGA has 1518 DSPs and assuming that the number of DSPs is the bottleneck in terms of area available, the maximum number of PEs the FPGA can accommodate is $\lfloor 1518 \div ColCnt \rfloor$. As an example, considering $ColCnt = 16$, the peak performance of one PE is 14.4 GFLOPS and the number of PEs that fit in the FPGA are 94, thus the total peak performance is 1353.6 GFLOPS.

4.2 Matricised Tensor Time Khatri-Rao Product (MTTKRP)

The first approach for MTTKRP processing on this device was to parallelize the processing of the matrices' columns similar to the approach for TTM in Kernel 4.1. While this approach proved to be an interesting starting point, a few adjustments in regards to the operation order were performed in order to achieve maximum performance.

```
1  event e { q.submit([&](handler &h) {
2      h.single_task([=]() [[intel::kernel_args_restrict]] {
3          float inTmp[colCnt], outTmp[colCnt];
4
5          // Iterate over all slices with non-zero elements
6          for (auto slc { 0 }; slc < slcCnt; ++slc) {
7
8              #pragma unroll
9              for (auto col { 0 }; col < colCnt; ++col) {
10                 outTmp[col] = 0.0f;
11             }
12
13             // Load slice boundaries and within that slice,
14             // iterate over all fibers with non-zero elements
15             for (auto fbr { accSlcPtr[slc] }; fbr < accSlcPtr[slc+1]; ++fbr) {
16                 // Load fiber index
17                 const auto j { (accFbrIdx[fbr] - 1) * colCnt };
18
19                 #pragma unroll
20                 for (auto col { 0 }; col < colCnt; ++col) {
21                     inTmp[col] = 0.0f;
22                 }
23
24                 // Within each fiber, iterate over all non-zero elements
25                 for(auto ele { accFbrPtr[fbr] }; ele < accFbrPtr[fbr+1]; ++ele){
26                     // Load element's index and value
27                     const auto k { (accKIdx[ele] - 1) * MATRIX_DIM };
28                     const auto val { accValues[ele] };
29
30                     // Load corresponding row from matrix2
31                     // Compute product and accumulate
```

```
32                      #pragma unroll
33                      for (auto col { 0 }; col < colCnt; ++col) {
34                          inTmp[col] += val * accMatrix2[k + col];
35                      }
36                  }
37
38                  // Load corresponding row from matrix1
39                  // Compute product and accumulate
40                  #pragma unroll
41                  for (auto col { 0 }; col < colCnt; ++col) {
42                      outTmp[col] += inTmp[col] * accMatrix1[j + col];
43                  }
44              }
45
46              // Store output row to global memory
47              #pragma unroll
48              for (auto col { 0 }; col < colCnt; ++col) {
49                  accOutput[slc * colCnt + col] = outTmp[col];
50              }
51          }
52      });
53  }) };
```

Listing 4.3: MTTKRP Kernel for FPGA

Kernel 4.3 starts by iterating over all slices with non-zero elements (line 6). Within each of those slices, it iterates over all fibers with non-zero elements (line 15) and within each of those fibers, it iterates over every last non-zero element (line 25). Then for every non-zero element, its values is multiplied against the corresponding row of matrix1 and accumulates the result (lines 32-35). When all non-zero elements within the fiber are done processing, the accumulated results of all those elements are multiplied against a row of matrix2 (lines 40-43). The original Kernel did not include the lines 40-43, instead it multiplied the element's value by the two matrices on line 34. However, since the indexed row of matrix1 is the same for all elements in the same fiber, it is inefficient to constantly load and multiply the same row. Hence, the mentioned adjustments in order to increase performance.

By correlating Kernel 4.3 with Figure 4.2, which depicts a general diagram of the design created on the FPGA, it is possible to better visualise and understand the design. The start block refers to the initialization of the variable slc on line 6. The following block is represented by the condition to check whether there are more slices on the same line, followed by the load of the slice's boundaries and the condition to check whether there are more fibers within the current slice (line 15). Continuing

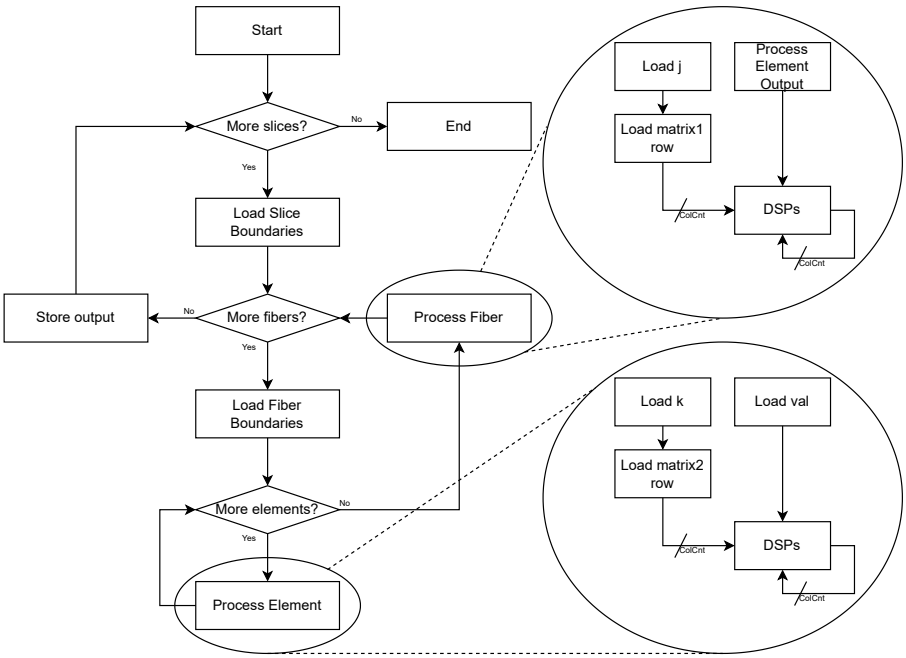


Figure 4.2: Diagram of PE

down the same path, the fiber's boundaries are loaded and the existence of more non-zero elements is verified (line 25). If the current fiber still has unprocessed non-zero elements, the Process Element block does the computations required on the next element (lines 26-35). Otherwise, the results accumulated throughout the several executions of the Process Element block are fed to the Process Fiber block. This block multiplies all elements of the Process Element output against the corresponding row of matrix1 (lines 40-43). Finally, when all fibers in the current slice are done processing, the accumulated results from block Process Fiber are stored to the global memory (lines 47-50). It is important to notice that, due to the pragmas used, like in the design for TTM, it is possible to load, compute and store entire rows in one go.

Moving to the AI derivations of this design, the number of loads the tensor requires are $SlcCnt + 1$ for the slice boundaries, $2 \times FbrCnt + 1$ for the fiber index and its boundaries and $2 \times NnzCnt$ for the index and value of the non-zero element. From the matrices, since we always load a row, all loads are of length $ColCnt$. For matrix2 a row is requested $NnzCnt$ times and for matrix1 a row is requested $FbrCnt$ times. Therefore the total amount of loads from the matrices is $ColCnt \times (FbrCnt + NnzCnt)$. For each slice, there is one store for each column, which makes the total amount of stores equal to $SlcCnt \times ColCnt$. The computations required are $ColCnt$ MACs for each $NnzCnt$ as well as $ColCnt$ more MACs for each fiber and since each MAC is composed of two operations, the total amount of operations is $2 \times ColCnt \times (FbrCnt + NnzCnt)$. Let all data types be 4-byte wide and the AI for a generic tensor can be expressed as follows:

$$AI_{sp} = \frac{1}{4} \times \frac{2 \times ColCnt \times (FbrCnt + NnzCnt)}{SlcCnt \times (ColCnt + 1) + FbrCnt \times (ColCnt + 2) + NnzCnt \times (ColCnt + 2) + 2} = \tag{4.4}$$
$$= \frac{1}{2} \times \frac{ColCnt \times (FbrCnt + NnzCnt)}{SlcCnt \times (ColCnt + 1) + (FbrCnt + NnzCnt) \times (ColCnt + 2) + 2}$$

The AI, once again, is determined by the tensor's characteristics and is bounded between a maximum and a minimum. The minimum AI can be obtained when $NnzCnt$ and $ColCnt$ are at their minimum, meaning one. Since there is only one non-zero element in the tensor, this causes $SlcCnt$ and $FbrCnt$ to also be one, as only slices and fibers with at least one non-zero element are stored. Thus, the design's minimum AI can be expressed as follows:

$$\min(AI_{sp}) = \min\left(\frac{1}{2} \times \frac{ColCnt \times (FbrCnt + NnzCnt)}{SlcCnt \times (ColCnt + 1) + (FbrCnt + NnzCnt)(ColCnt + 2) + 2}\right) = \frac{1}{10} \tag{4.5}$$

As for the maximum AI, the rationale is the same, however, in this case, $ColCnt$ must be as large as possible in order to have the design perform the most computations possible, on the other hand, $SlcCnt$ must be as little as possible in order to decrease the number of loads, meaning all non-zero elements are in the same slice. The following equation derives the design's AI upper bound:

$$\max(AI_{sp}) = \max\left(\frac{1}{2} \times \frac{ColCnt \times (FbrCnt + NnzCnt)}{SlcCnt \times (ColCnt + 1) + (FbrCnt + NnzCnt) \times (ColCnt + 2) + 2}\right) = \tag{4.6}$$
$$= \max\left(\frac{1}{2} \times \frac{FbrCnt + NnzCnt}{1 + (FbrCnt + NnzCnt)}\right) \approx \frac{1}{2}$$

As denoted in Equation 4.6, in order to achieve maximum AI, either $FbrCnt$, $NnzCnt$ or both must be large enough to verify the condition $\frac{FbrCnt+NnzCnt}{1+(FbrCnt+NnzCnt)} \approx 1$. However, since the CSF format does not store slice nor fibers without non-zero elements, then $SlcCnt \leq FbrCnt \leq NnzCnt$. Therefore ensuring $NnzCnt$ is large enough is both sufficient and necessary.

At last to analyse the peak performance of the design and and how it compares to the one of the FPGA. Per PE, the design uses two DSP for every column on the matrices, therefore each PE performs $4 \times ColCnt$ operations per cycle. Resorting to Table 4.2 the maximum frequency at which the DSPs operate is 450 MHz, therefore the peak performance of each PE is $1.8 \times ColCnt$ GFLOPS. Considering the number of DSPs as the bottleneck in terms of area available, the maximum number of PEs the FPGA can accommodate is $\lfloor 1518 \div (2 \times ColCnt) \rfloor$. As an example, considering $ColCnt = 16$, the peak

performance for a single PE is 28.8 GFLOPS and the number of PEs that fit in the FPGA are 47, thus the total peak performance is 1353.6 GFLOPS.

4.3 Summary

This chapter thoroughly describes our designs for TTM and MTTKRP on specialised architectures. An in-depth analysis of these designs is provided alongside the derivation of the AI expressions. The AI limits and the kinds of data-set that can originate such corner cases are presented followed by a peak performance analysis based on the designs resource utilisation and on the total available resources on the FPGA.

Chapter 5

Experimental Results on Real-World Tensors

After having both the worst and best-case scenarios discussed, as well as theoretically and experimentally verified, we focus on analysing the two proposed approaches for data-parallel processing with real-world execution scenarios for each of the methods. For this analysis, like in previous sections, all tests and measurements were collected under Intel's Devcloud environment and the hardware setup used is described in Table 5.1.

Device	Model	Frequency	Cores
CPU	Intel Core i9-11900KB	3,30 GHz	16[1]
GPU	Intel 11th Gen UHD Graphics	1,45 GHz	32[2]

[1] Physical Cores $\times$ Concurrent Threads per Core $= 8 \times 2$
[2] Execution Units

Device	Model	Frequency	ALMs	Registers	DSPs
FPGA	Intel Arria 10 GX 1150	450 MHz	427 200	1 708 800	1518

Table 5.1: Hardware setup

Tensors nell-2 and vast-3D from the FROSTT data-set [65] were used, as described in Table 5.2. Since all parameters of these tensors are already predetermined, the only parameter that can be modified for TTM and MTTKRP computation is the number of columns in the matrices, i.e., $ColCnt$.

	SlcCnt	FbrCnt	NnzCnt	Mode 0	Mode 1	Mode 2
nell-2 [66]	12 092	337 365	76 879 419	12 092	9 184	28 818
vast-3D [67]	165 427	26 021 945	26 021 945	165 427	11 374	2

Table 5.2: Description of data-sets used

Given the characteristics of these tensors, we can determine the expected AI for each kernel. Tensor

nell-2 has, in average, 228 non-zero elements per fiber, 28 fibers per slice and 6358 non-zero elements per slice. While tensor vast-3D has always $NnzPFbr = 1$ and therefore $FbrPSlc = NnzPSlc$, with the number of fiber per slice being, in average, 157.

According to Equation 3.1, the expected AI for Kernel 3.1 is 0.166 for tensor nell-2, which is the theoretical maximum, and 0.083 for tensor vast-3D, which is the theoretical minimum. For Kernel 3.2, the AIs are the same but multiplied by a factor dependent on the number of columns in the matrix. For that reason, further considerations are made below alongside the empirical values. We resort now to Equation 3.7 for Kernel 3.3, attaining 0.167 and 0.166 as the expected AIs for tensors nell-2 and vast-3D respectively. These are both very close to the theoretical maximum. For Kernel 3.4, the AI also depends on $ColCnt$, however it still possible to determine the expected range of AI. For tensor nell-2 the expected AI ranges from 0.167 to 0.500 with the increase of $ColCnt$, while for tensor vast-3D ranges from 0.166 to 0.500.

5.1 CPU Results

We start with TTM on the CPU, for this method the most relevant parameters are the total number of fibers in the tensor, to ensure enough parallel work, as well as the number of non-zero elements per fiber on average, to determine the AI.

The first tensor to be analysed is tensor nell-2. This tensor's mode-2 dimension imposes 28818 rows on the matrix, thus it never fits in L1 cache.

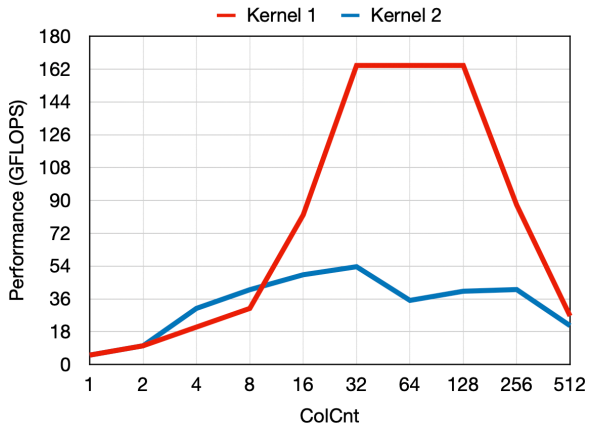


Figure 5.1: TTM performance on CPU for tensor nell-2 with varying number of matrix columns

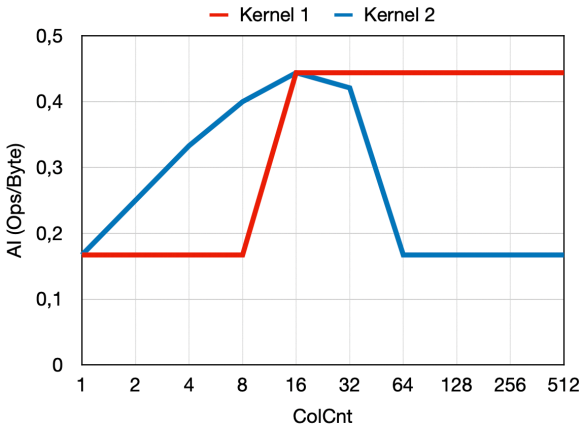


Figure 5.2: TTM AI on CPU for tensor nell-2 with varying number of matrix columns

Figures 5.1 and 5.2 represent the performance and AI for both kernels for tensor nell-2 with the increase of number of columns in the matrix. For Kernel 3.1, the AI should be fixed, since it only depends on $NnzPFbr$ and not on $ColCnt$, however from $ColCnt = 16$, the compiler is capable of

vectorising kernel loops and therefore provoking an increase in both the AI and the performance. On the other hand, for Kernel 3.2, as expected, since the AI depends on the number of columns, it starts with the same value as in the previous Kernel and, with the increase in number of columns, increases until very close to the maximum calculated in Equation 3.6. However, since the kernel loops are never vectorised by the compiler, the AI ends up dropping for larger workloads. From $ColCnt = 128$, the performance starts reducing for Kernel 3.1, while the AI is maintained, since the matrix does not fit anymore in any of the caches of the CPU. On Kernel 3.2, performance increases with the AI as the problem is always memory bound for all cache levels but L1, where the matrix would never fit anyway. It is also important to notice that the performance drops mostly match with the sizes of the cache levels.

When compared with the performance of Kernel 3.1 for this same tensor, the Kernel 3.2 attains the lower performance. The main reason behind this behavior lies in the irregularity of data accesses and workload imbalance that were not present in the best case evaluation.

For the vast-3D tensor, which has all fibers with a single non-zero element, the AI should be very close to the theoretical minimum. Due to its mode-2 dimension, the matrix only has 2 rows, thus for all numbers of columns tested, it always fits in L1 cache.

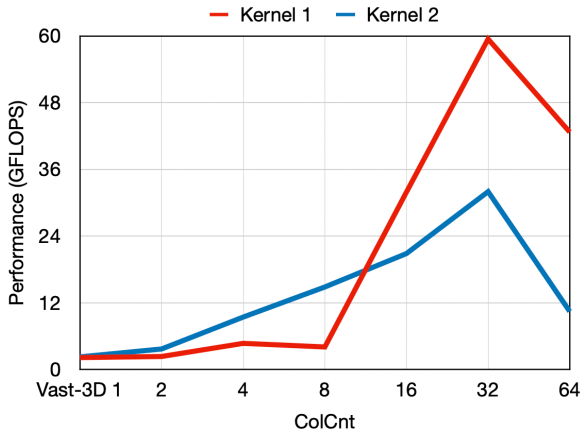


Figure 5.3: TTM performance on CPU for tensor vast-3D with varying number of matrix columns

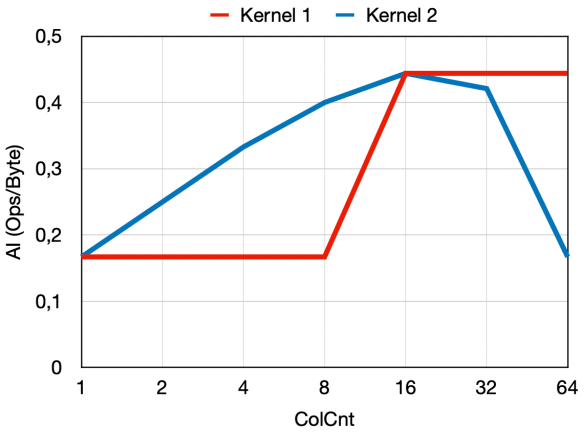


Figure 5.4: TTM AI on CPU for tensor vast-3D with varying number of matrix columns

Figures 5.3 and 5.4 represent the performance and AI of both kernels for tensor vast-3D with the increase of number of columns in the matrix. As in the nell-2 tensor case, the expected constant AI for Kernel 3.1 was not observed due to the compiler optimisations, which provokes an increase in both AI and kernel performance for a larger number of columns. This behaviour can be observed around $ColCnt = 16$, where a notable AI increase occurs due to the compiler's ability to vectorise the kernel loops. Since the matrix always fits in L1 cache the performance drop that happened in the nell-2 tensor for the same Kernel does not happen with the vast-3D tensor. For Kernel 3.2, although the AI was expected to be lower, it is not due to compiler optimization, it still behaves as expected. It starts with the same value as in the previous Kernel and, with the increase in number of columns, increases until

very close to the maximum calculated in 3.6. However, since the kernel loops are never vectorised by the compiler, the AI ends up dropping for larger workloads. The performance also increases with the AI, which can be explained by most of the workload coming from the number of columns, since each fiber only has one element to compute.

Now to analyse MTTKRP also on the CPU. The parameters most relevant for the analysis of this method are the total number of slices in the tensor, to ensure parallelism, and the number of non-zero elements per slice and per fiber, both for the expected AI.

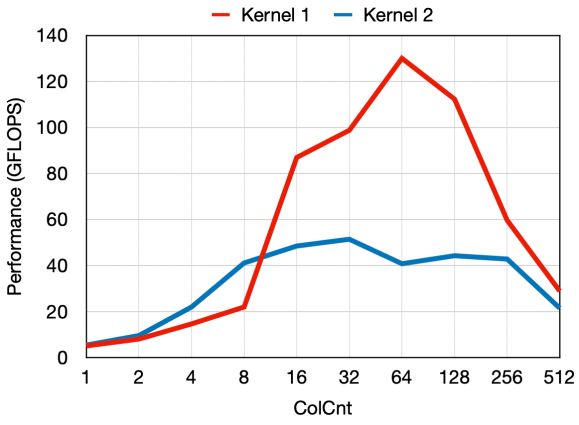


Figure 5.5: MTTKRP performance on CPU for tensor nell-2 with varying number of columns

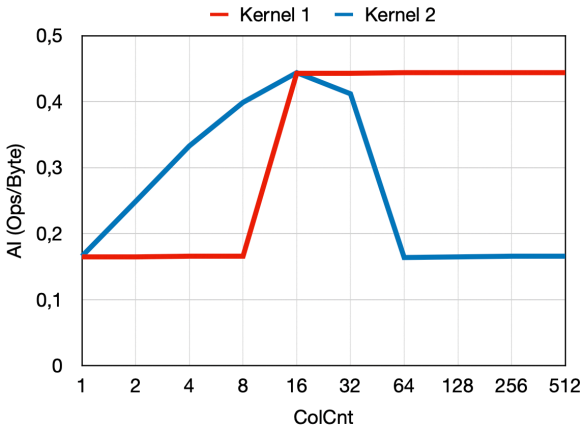


Figure 5.6: MTTKRP AI on CPU for tensor nell-2 with varying number of columns

Figures 5.5 and 5.6 represent Kernels' 3.3 and 3.4 performance and AI for tensor nell-2 with the increase of number of columns in the matrices. The first kernel's AI should be stable as does not depend on $ColCnt$, however, similarly to what happened with the CPU processing of TTM, from $ColCnt = 16$, the compiler starts being capable of vectorising kernel loops, leading to an increase in both the AI and the performance. The second kernel's AI also behaves differently to what was expected. Despite the AI depending on the number of columns, therefore, with the increase in number of columns, increasing until very close to the maximum calculated in Equation 3.12, it drops for larger workloads. The reason is the fact that the kernel loops are never vectorised by the compiler. The performance for Kernel 3.3 starts reducing from $ColCnt = 128$, while the AI is maintained. Such phenomenon is explained by the matrices does not fitting anymore in any of the cache levels of the CPU. On Kernel 3.4, performance drops mostly match with the sizes of the cache levels. While performance increases as the AI increases, due to the problem always being memory bound for all cache levels but L1, where the matrices would never fit anyway, there are still performance drops that mostly coincide with the cache level transitions.

Tensor vast-3D, for MTTKRP, imposes 11374 and 2 rows on matrix1 and matrix2, respectively. Due to matrix2's more frequent accesses and smaller dimension, it can be reused from L1 cache, while matrix1 is mostly bounded to either L2 or L3 cache depending on the number of columns. Contrary to what happened in tensor nell-2 where matrix2, which never fits in L1 cache, polluted all cache levels, forcing

all access of both matrices to come from either L3 or even DRAM. Therefore performace is higher for this data-set.

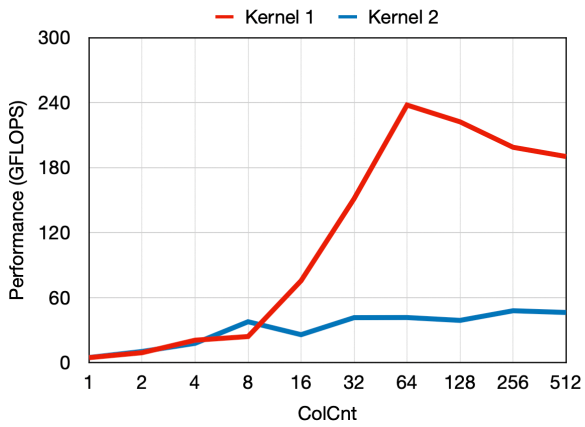


Figure 5.7: MTTKRP performance on CPU for tensor vast-3D with varying number of columns

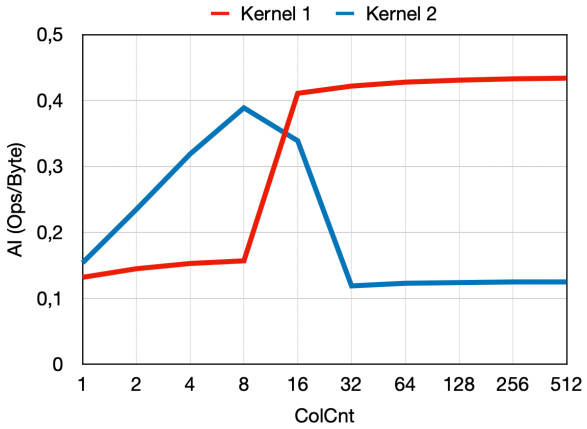


Figure 5.8: MTTKRP AI on CPU for tensor vast-3D with varying number of columns

Figures 5.7 and 5.8 denote the performance and AI attained by both MTTKRP kernels on the CPU when processing tensor vast-3D. The AIs follow a similar pattern to the previous kernels, with Kernel 3.3 achieving vectorisation around $ColCnt = 16$ and with Kernel 3.4 starting to drop in AI due to the lack of said vectorisation. In regards to performance, performance raises moderately for both kernels until $ColCnt = 8$, which can be explained with the increase in AI. From there Kernel's 1 performance spikes with vectorisation. Kernel's 2 performance, on the other hand, starts to stabilise as most of the workload comes from the number of columns and increasing the number of columns also provokes more loads (each thread loads full rows from the matrices).

5.2 GPU Results

Starting again with tensor nell-2. This tensor has an average of $NnzPFbr = 228$ so the AI is very close to the theoretical maximum. Also the tensor has 337365 fibers so GPU occupation is not an issue. Kernels 3.1 and 3.2 behave as follows:

Figures 5.9 and 5.10 represent the performance and AI for both kernels for tensor nell-2 with the increase of number of columns in the matrix. For Kernel 3.1, the AI is fixed as it only depends on the $NnzPFbr$ and not on $ColCnt$. The main difference between this situation and the best case scenario, described in Section 3.2.1.B, is the irregularity and unbalance in the distribution of the non-zero elements over the fibers as well as the size of the matrix. For such reasons, the peak performance is achieved for $ColCnt = 32$, instead of $ColCnt = 64$, and is slightly less than the best case. For Kernel 3.2, as expected, the AI starts with the same value as in the previous Kernel but increases until very close to

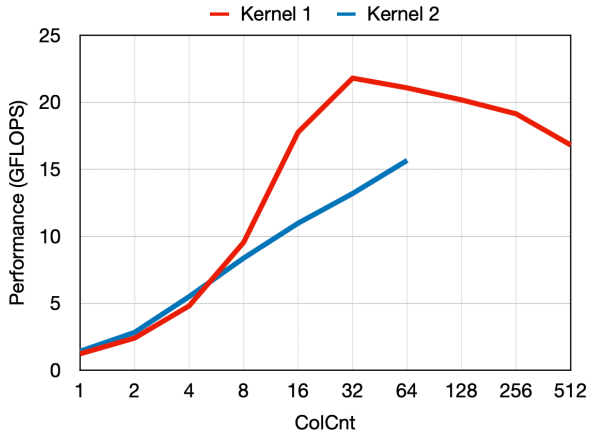


Figure 5.9: TTM performance on GPU for tensor nell-2 with varying number of matrix columns

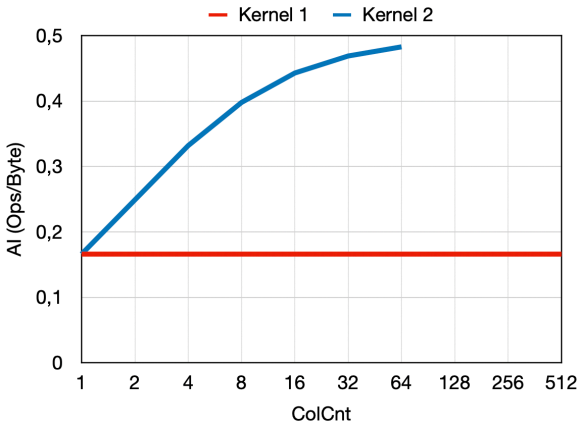


Figure 5.10: TTM AI on GPU for tensor nell-2 with varying number of matrix columns

the theoretical maximum calculated in 3.6. The performance also increases with the AI as the problem is memory bound on the GPU.

When compared with the performance of Kernel 3.1 for the same tensor, this Kernel performs worst even though it performed better for the best case tensor. The main reason is the irregularity in data accesses and workload imbalance that were not present in the best case, when Kernel 3.2 was the better option.

Tensor vast-3D, which has all fibers with a single non-zero element, should force the Kernel to have an AI very close to the theoretical minimum. Also the tensor has 26021945 fibers therefore GPU occupation is not an issue.

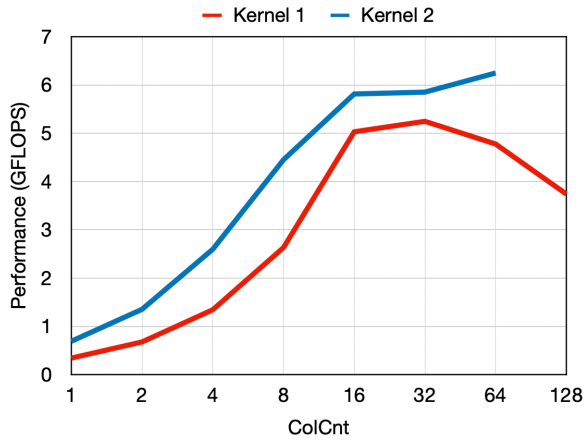


Figure 5.11: TTM performance on GPU for tensor vast-3D with varying number of matrix columns

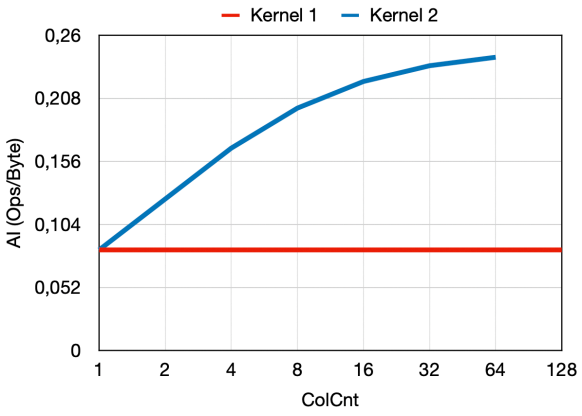


Figure 5.12: TTM AI on GPU for tensor vast-3D with varying number of matrix columns

Figures 5.11 and 5.12 represent the performance and AI for both kernels for tensor vast-3D with the increase of number of columns in the matrix. For Kernel 3.1, the AI is fixed, as expected, since it only

depends on $NnzPFbr$ and not $ColCnt$. The main difference between this situation and the worst case scenario, described in Section 3.2.2.B, is $RowCnt$ which causes the matrix to be much smaller in this tensor. For Kernel 3.2, again as expected, the AI starts with the same value as in the previous Kernel but increases with the increasing number of columns. The performance also increases with the AI as TTM is by nature memory bound on the GPU.

When compared, Kernel 3.2 performs better than Kernel 3.1 for tensor vast-3D. The main reason is the matrix size, since it is so small, it is efficient to compute whole rows as it is likely that the row is already cached.

Finally, it is possible to create a roofline model sweeping the AI by changing the number of columns. This analysis, however, only makes sense for Kernel 3.2, since in Kernel 3.1 the AI is fixed, and for the GPU, since in the CPU the compiler tends to perform optimizations which make the AI's behaviour unpredictable. These can be found in Figures 5.13 and 5.14.

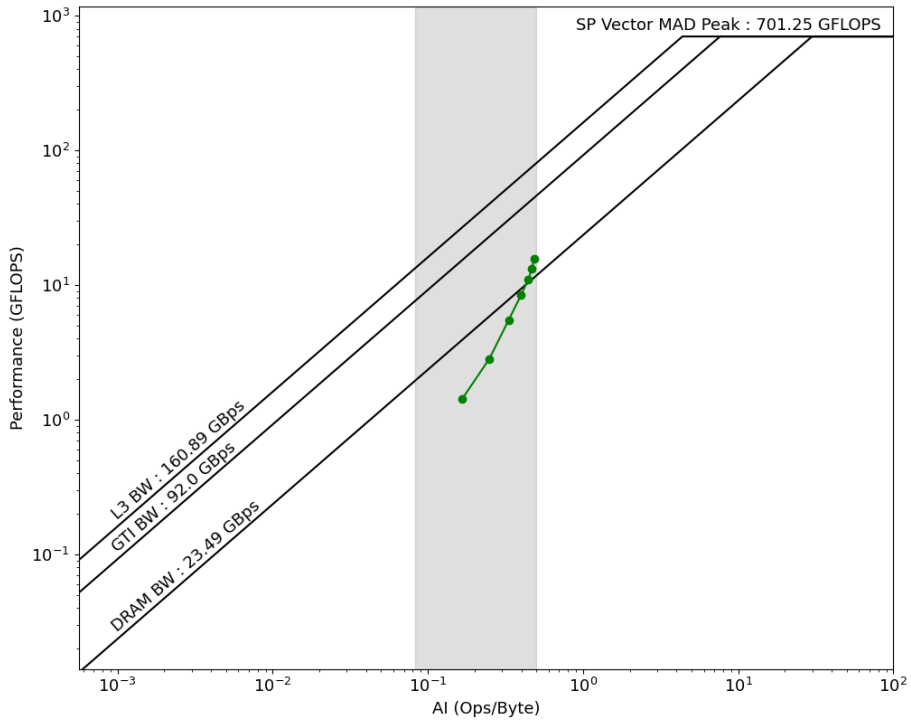


Figure 5.13: Roofline model for Kernel 3.2 with tensor nell-2 on the GPU

Now to analyse MTTKRP on the GPU. The parameters most relevant for the analysis of this method are the total number of slices in the tensor, to ensure parallelism, and the number of non-zero elements per slice and per fiber, both for the expected AI. In regard to tensor nell-2, it has enough slices to ensure full occupation on the GPU and its characteristics, namely $FbrPSlc$ and $NnzPSlc$, ensure that the AI of the kernels for this specific data-set will be close to the theoretical maximum.

Figures 5.15 and 5.16 represent the performance and AI for both kernels for tensor nell-2 with the

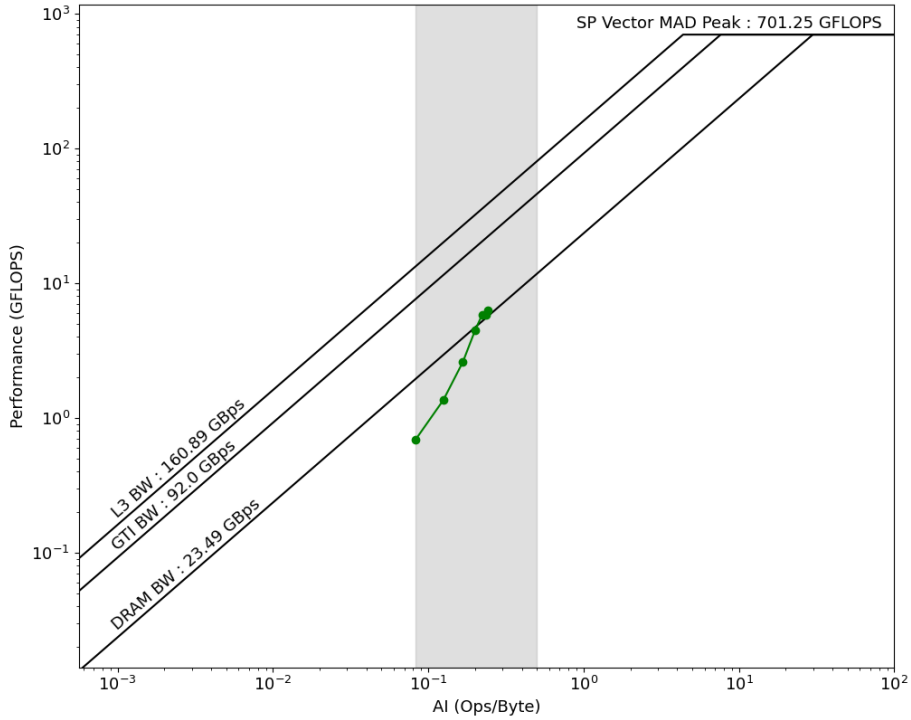


Figure 5.14: Roofline model for Kernel 3.2 with tensor vast-3D on the GPU

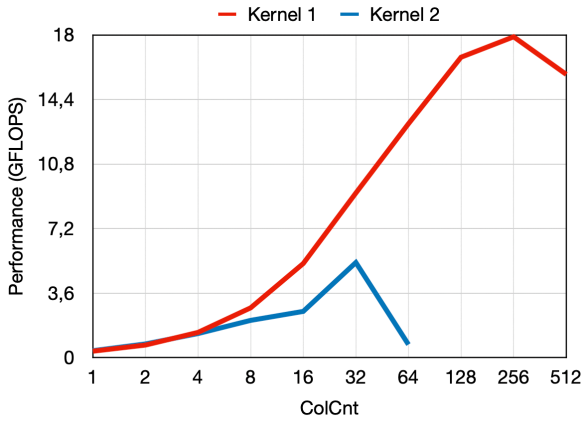


Figure 5.15: MTTKRP performance on GPU for tensor nell-2 with varying number of columns

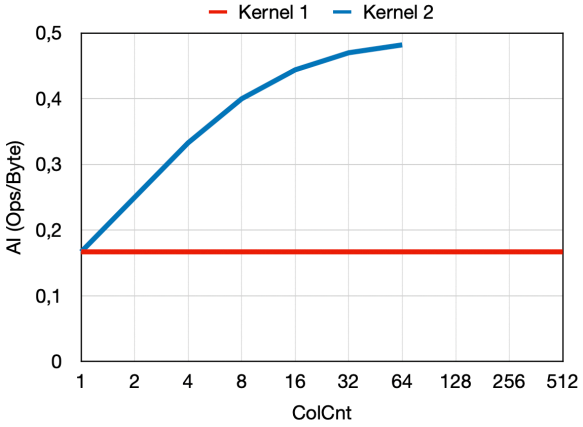


Figure 5.16: MTTKRP AI on GPU for tensor nell-2 with varying number of columns

increase of number of columns in the matrices. Both performance and AI behave as expected for both kernels. The AI, for Kernel 3.3, is fixed, since it does not depend on $ColCnt$, whereas for Kernel 3.4 it steadily approximates the maximum this data-set allows. For Kernel 3.3, performance increases until $ColCnt = 256$ and then drops, the increase can be explained with the increase reuse of data between threads. Each thread either shares the same slice with $ColCnt$ other or the same column with $SlcCnt$ others. For Kernel 3.4, performance increases with the AI as MTTKRP is memory bound on the targeted device. From $ColCnt = 32$, performance drops, this happens due to size of the matrices compared to

L3 capacity, meaning these no longer fit in L3 forcing occasional loads from the DRAM.

Tensor vast-3D has 165427 fibers therefore GPU occupation is not an issue. Since, unlike the CPU, the GPU does not possess a complex cache structure, the performance differences between this data-set and tensor nell-2 are much reduced and justifiable by the better locality present in vast-3D.

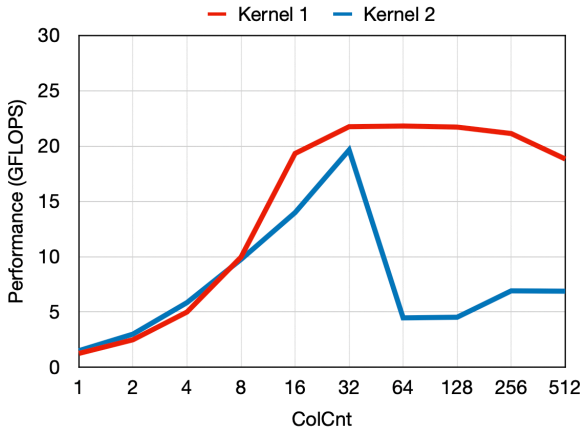


Figure 5.17: MTTKRP performance on GPU for tensor vast-3D with varying number of columns

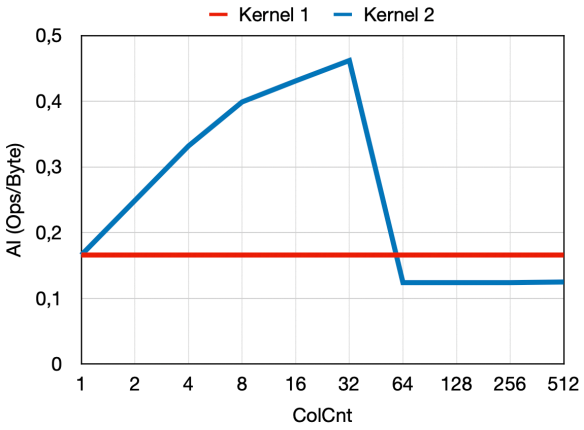


Figure 5.18: MTTKRP AI on GPU for tensor vast-3D with varying number of columns

Figures 5.17 and 5.18 represent the performance and AI for both kernels for tensor vast-3D with the increase of number of columns in the matrices. For Kernel 3.3, the AI, most like in the prior data-set, does not depend on $ColCnt$ therefore being fixed. The performance, on the other hand, increases and stabilises until $ColCnt = 128$, where, due to not fitting in L3 cache anymore, it starts dropping. For Kernel 3.4, the AI starts with the same value as in the previous Kernel and increases with the increasing number of columns, however due to the lack of reuse and increase in size of the matrices it drops from $ColCnt = 64$ onward. The performance follows the AI's trend as MTTKRP is memory bound on the GPU by nature.

5.3 FPGA Results

We start by exploring the performance and AI of our TTM design on the FPGA. For this method the most relevant parameters are the total number of fibers and non-zero elements in the tensor as well as the total amount of columns in the matrix, all fundamental in order to determine the AI.

By resorting to Equation 4.1 and Table 5.2, it is possible to make the design's AI only dependent on $ColCnt$. Tensor nell-2 is the first to be analysed and for this tensor we have $FbrCnt = 337365$ and $NnzCnt = 76879419$, therefore the AI starts at 0.166 when $ColCnt = 1$ and tends to 0.500 as $ColCnt$ grows larger.

Figures 5.19 and 5.20 portray the performance as well as the AI's behaviour for a varying number

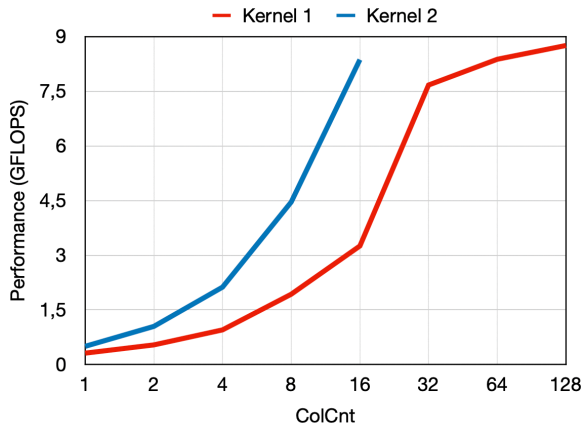


Figure 5.19: TTM performance on FPGA for tensor nell-2 with varying number of columns

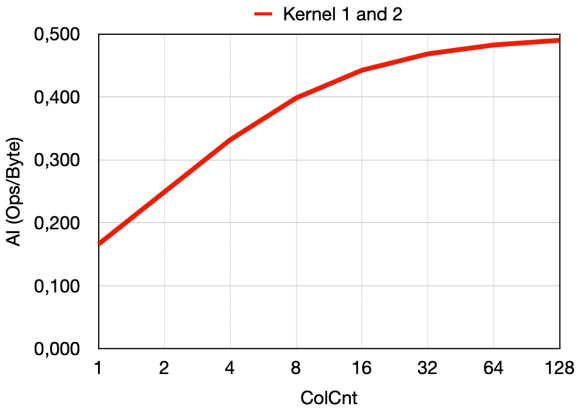


Figure 5.20: TTM AI on FPGA for tensor nell-2 with varying number of columns

of columns in the matrix. Performance increases exponentially with the increase of columns for both kernels. The reason behind this behaviour is the fact that, in order to permit parallel processing of a full row at a time, a new DSP is added to the design for each column in the matrix. There is, however, a limitation for this trait: due to a maximum data-width for each load, for $ColCnt \geq 64$, the design requires more than one load to retrieve the full row from the memory, causing the increase in performance to decelerate. Through the difference in performance between both kernels, it is also clear the impact of having the matrix transferred to local memory before the computation starts. As for the AI, it is, naturally, the same for both kernels as the only difference between them is where they load the rows from, namely the DRAM for the first Kernel and the on-chip memory for the second kernel.

For tensor vast-3D, it is also possible to resort to Equation 4.1 and Table 5.2 to get the design's AI in function of $ColCnt$. The tensor has $FbrCnt = NnzCnt = 26021945$, therefore the AI starts at the theoretical minimum, meaning 0.100, when $ColCnt = 1$ and tends to 0.250 as $ColCnt$ grows larger.

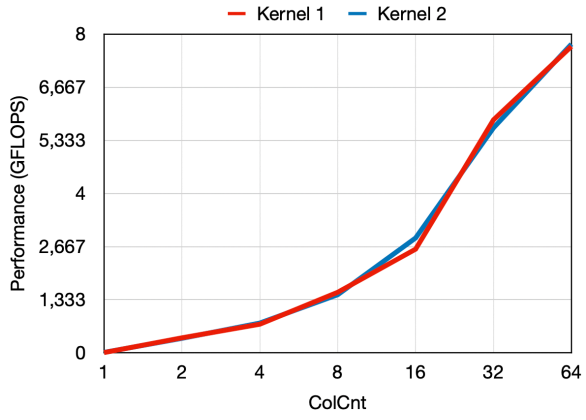


Figure 5.21: TTM performance on FPGA for tensor vast-3D with varying number of columns

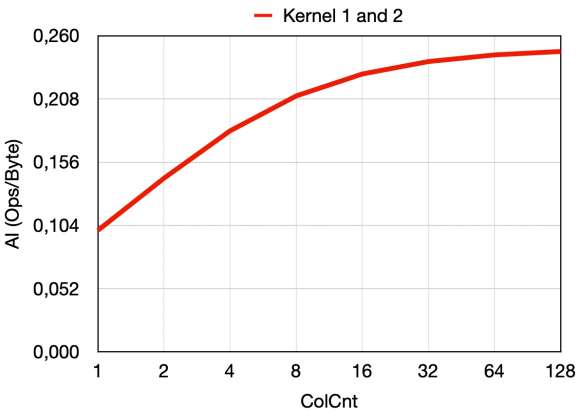


Figure 5.22: TTM AI on FPGA for tensor vast-3D with varying number of columns

Figures 5.21 and 5.22 depict the performance and AI with increase in number of columns in the matrix. Like for the previous data-set, performance increases exponentially with the increase of columns for both kernels. The explanation for this behaviour is the same as before, in order to permit parallel processing of a full row at a time, a new DSP is added to the design for each column in the matrix. One can also notice the same limitation for $ColCnt \geq 64$ as the performance growth decelerates due to the need of multiple loads to retrieve a single a row. For vast-3D, there is no noticeable difference in performance for both versions, this happens due to the small size of the matrix, only two rows. The compiler when generating the design, stores the values loaded from memory in registers therefore loading to on-chip memory has minimum impact on the performance. The only difference between both kernels is, once again, where they load the rows from, namely the DRAM for the first Kernel and the on-chip memory for the second kernel, therefore the AI is, naturally, the same for both.

Now to explore the performance and AI of our MTTKRP design on the FPGA. This method's most relevant parameters are the total number of slices, fibers and non-zero elements in the tensor as well as the total amount of columns in the matrices, all fundamental in order to determine the AI.

By resorting to Equation 4.4 and Table 5.2, it is possible to make the design's AI only dependent on $ColCnt$. Tensor nell-2 is the first to be analysed and for this tensor we have $SlcCnt = 12092$, $FbrCnt = 337365$ and $NnzCnt = 76879419$, therefore the AI starts at 0.167 when $ColCnt = 1$ and tends to 0.500 as $ColCnt$ grows larger.

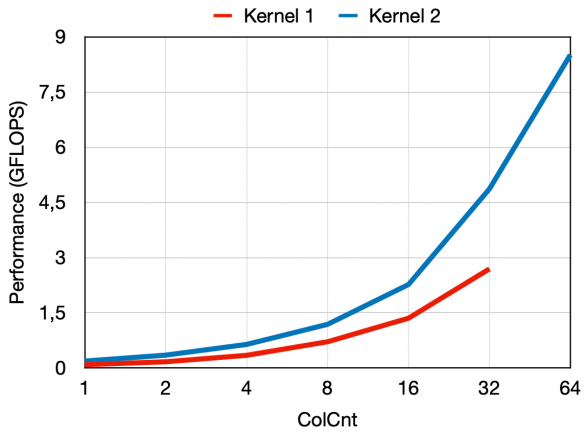


Figure 5.23: MTTKRP performance on FPGA for tensor nell-2 with varying number of columns

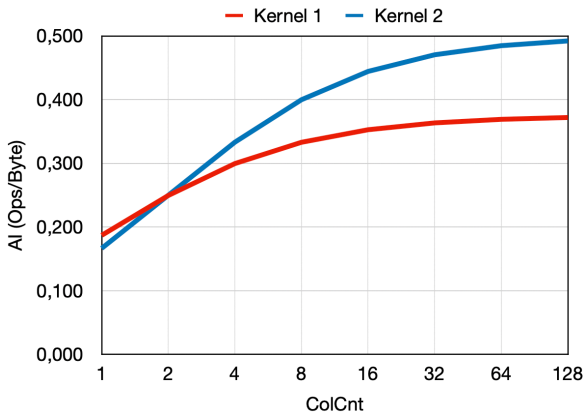


Figure 5.24: MTTKRP AI on FPGA for tensor nell-2 with varying number of columns

In Figures 5.23 and 5.24, a depiction of the performance and AI of our MTTKRP design for different number of columns in the matrices is presented. For comparison sake, measures for the design before and after the optimisations described in Section 4.2 are provided. Performance-wise both grow exponentially as expected, the reason being, once again, the addition of extra DSPs every time the number of columns increases. This allows for the execution time to be the same while processing more

columns simultaneously, hence increasing the performance. Since FPGAs have variable architectures, it is possible for an optimised design to perform less operations while still arriving at the same results, therefore, in order to achieve a fair comparison, it is important to not consider any unnecessary operations in the performance calculations. Thus for both kernels the effective number of operations, meaning the number of operations after optimisation, was used. For the AI calculations, however, the real number of operations and loads for both kernels were used. For $ColCnt \leq 2$, Kernel 1 has higher AI because the higher number of operations manages to outweigh the extra loads necessary, however as $ColCnt$ increases the extra number of loads becomes dominant leading to a higher AI for the second kernel.

For tensor vast-3D, it is also possible to resort to Equation 4.4 and Table 5.2 to get the design's AI in function of $ColCnt$. The tensor has $SlcCnt = 165427$ and $FbrCnt = NnzCnt = 26021945$, therefore the AI starts at 0.166, when $ColCnt = 1$ and tends to the theoretical maximum, meaning 0.500, as $ColCnt$ grows larger.

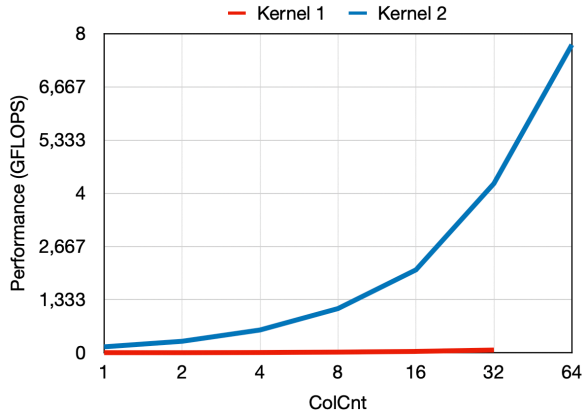


Figure 5.25: MTTKRP performance onFPGA for tensor vast-3D with varying number of columns

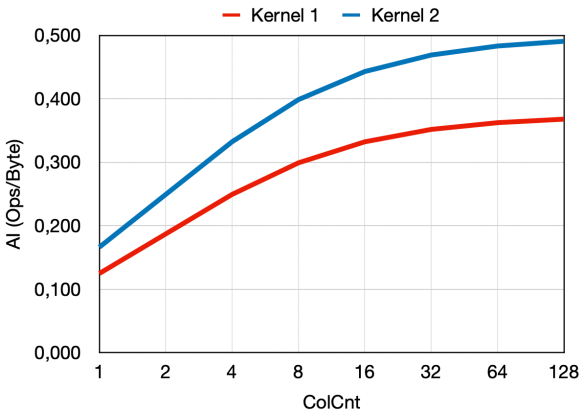


Figure 5.26: MTTKRP AI on FPGA for tensor vast-3D with varying number of columns

In Figures 5.25 and 5.26, the performance and AI of our MTTKRP design for different number of columns in the matrices are portrayed. Like before, for comparison sake, measures for the design before and after the optimisations described in Section 4.2 are provided. Due to the nature of vast-3D ($FbrCnt = NnzCnt$), not only does the Kernel before the optimisations (Kernel 1) have lower AI for all numbers of columns, but also does it have drastically less performance. The optimised design performs as expected, with the performance increasing exponentially with the increase in number of columns. When compared the performance of this same design for both data-sets, one can notice that for tensor vast-3D is higher, the reason being that locality in data accesses, which is much higher.

5.4 Comparison with State of the Art

We focus herein in comparing our implementations to some of the state-of-the-art implementations for sparse tensor processing. Said implementations are SPLATT [68], which implements MTTKRP on the CPU, and ParTI [69], which implements TTM and MTTKRP on CPU and GPU architectures. From our implementations, we chose the Element-Centric approaches, Kernels 3.1 and 3.3, since they perform better on real-world datasets.

Table 5.3: Hardware setup

Device	Model	Frequency	Cores	Environment
CPU	Intel Core i9-11900KB	3.30 GHz	16[1]	Intel Devcloud
CPU	AMD EPYC 7B13	3.5 GHz	112[2]	Google Cloud
GPU	Nvidia A100 - 40GB	1.41 GHz	108[3]	Google Cloud

[1] Physical Cores $\times$ Concurrent Threads per Core $= 8 \times 2$
[2] vCPUs [3] Streaming Multiprocessors

For this set of results, we resorted to Google Cloud with Ubuntu 22.04 LTS in addition to Intel's Devcloud. Since GPU state-of-the-art implementations were developed in CUDA, which is vendor-specific, we did not use Intel's GPU for the measures taken on those implementations. The Hardware setup is described in Table 5.3.

Table 5.4: Description of data-sets used

	SlcCnt	FbrCnt	NnzCnt	Mode 0	Mode 1	Mode 2
vast-3D [67]	165 427	26 021 945	26 021 945	165 427	11 374	2
nell-2 [66]	12 092	337 365	76 879 419	12 092	9 184	28 818
nell-1 [66]	2 902 330	17 372 417	143 599 552	2 902 330	2 143 368	25 495 389
amazon [70]	4 821 207	29 865 499	1 741 809 018	4 821 207	1 774 269	1 805 187

Also, in addition to the tensors described in Table 5.2, we analysed two more tensors. All tensors still belong to the FROSTT dataset and their characteristics are described in Table 5.4.

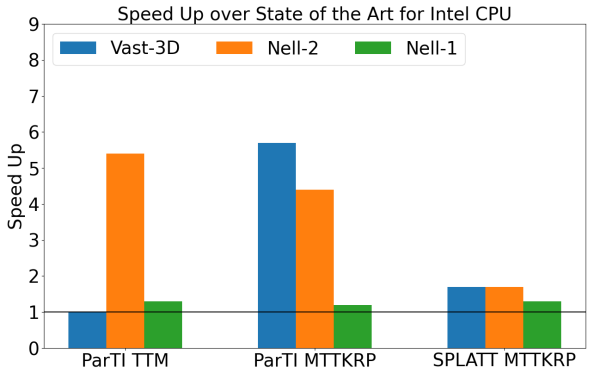


Figure 5.27: Speed up over State of the Art on Intel Core i9-11900KB

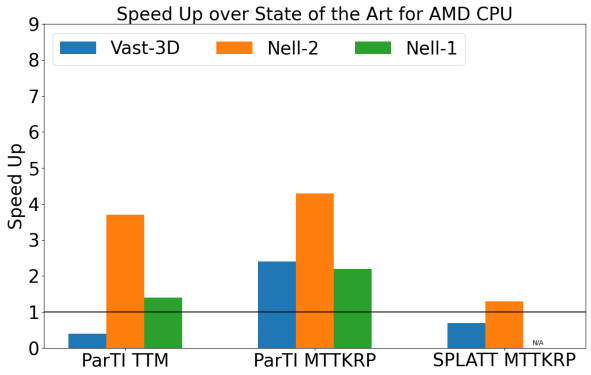


Figure 5.28: Speed up over State of the Art on AMD EPYC 7B13

Figures 5.27 and 5.28 portray the speed ups of our implementations over the state-of-the-art ones for CPU platforms. For MTTKRP, we outperform ParTI for all tested datasets (up to $6\times$ speed up), justified by the choice in storage formats. In average, CSF requires less memory accesses per element processed when compared to COO, hence delivering higher performance. A similar behaviour can be observed for the speed ups of our TTM implementation over ParTI's (up to $5\times$ speed up), except for tensor Vast-3D. The reason being it only having one element per fiber, therefore the advantage of CSF is not present. When comparing with SPLATT, the speed ups are always close to one since the major difference in approaches is the framework. While we resort to SYCL for our implementations, SPLATT uses OpenMP. It is also important to notice that the speed ups for the Intel CPU are, in average, higher since we use Intel's SYCL implementation and compilers, which, naturally, are more optimised for their own architectures.

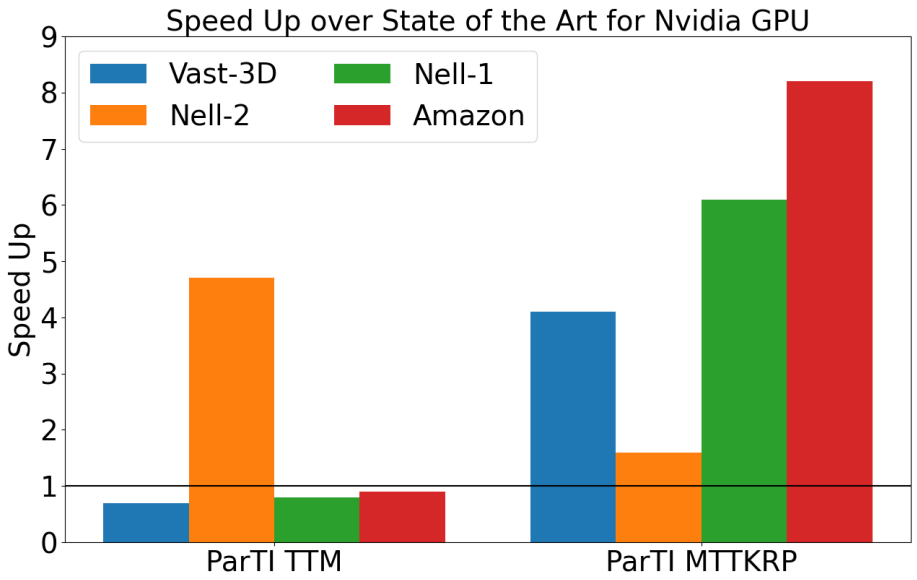


Figure 5.29: Speed up over State of the Art on Nvidia A100 - 40GB

Figure 5.29 depicts the achieved speed ups for the GPU platform. For MTTKRP, we always outperform ParTI with the justification, once again, lying mostly on the storage format. Unlike ParTI, which distributes the workload element-wise to ensure load balance, we distribute the workload slice-wise to avoid synchronisation. Therefore, the main source of parallelism of our MTTKRP implementations comes from the number of slices in the tensor. Thus, the lower speed up for tensor Nell-2, which has a reduced number of slices when compared with the other datasets. For TTM, we never achieve speed up, except for tensor Nell-2, which contrasts with the previous method. Nvidia GPUs impose extra limitations when compared to Intel GPUs, namely in the number of work-groups that can be launched per kernel submission. This forces our TTM implementation to launch multiple kernels, each processing a portion of the tensor, therefore hindering the performance. The exception is tensor Nell-2, since it has

less fibers, as can be observed in Table 5.4, and for that tensor we achieve approximately $4\times$ speed up.

5.5 Summary

In this chapter, we test our algorithms against real-world datasets on one Intel multi-core CPU and on its integrated GPU. Our designs are evaluated against the same tensors on an Intel FPGA. We also analyse the performance and AI behaviour by varying the number of columns on the matrices, $ColCnt$. Finally, we compare state-of-the-art implementations against our own, where we achieve up to $7\times$ speed-up.

Chapter 6

Conclusion

The main goal of this Thesis was to delve into sparse tensor computations, specifically TTM and $MTTKRP$, on heterogeneous systems. To achieve that, an extensive study of the most common computational architectures was made along with a detailed analysis of the algorithms' behaviour on the architectures. For the programmable devices, we exploited the available parallelism and data locality in memory accesses. Then to provide a better insight on the algorithms' implementation, we predicted and validated the AI and performance of our kernels with resort to a set of synthetic tensors, built from the theoretical analysis. This Thesis, also featured designs for specialised architectures that, by utilising their programmable hardware, allowed the development of the most scalable of our implementations.

Developing implementations in SYCL presented a gentle learning curve in comparison to the wide range of frameworks that otherwise would have been necessary. It also allowed us to create an implementation that leverages both CPU and GPU, hence a heterogeneous solution for sparse tensor computations. While there is still potential for more optimisation, this Thesis takes the first steps towards sparse tensor computations on heterogeneous systems.

For the duration of this dissertation, potential future studies were identified:

- Explore other optimisation possibilities on general-purpose architectures, e.g. cache blocking on the CPU and use of shared memory on the GPU.
- Develop an heterogeneous approach with dynamic assignment.
- Memory distributions and attributes for FPGA, e.g. memory banks.
- Scale FPGA implementations to multiple PEs.

These suggestions allow a deeper analysis within the context of this dissertation, as well as help face the surging needs for sparse tensor computations, especially on modern heterogeneous architectures.

Bibliography

[1] Y. Wang, R. Chen, J. Ghosh, J. C. Denny, A. Kho, Y. Chen, B. A. Malin, and J. Sun, "Rubik: Knowledge guided tensor factorization and completion for health data analytics," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, p. 1265–1274.

[2] J. C. Ho, J. Ghosh, and J. Sun, "Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014, p. 115–124.

[3] A. Zadeh, M. Chen, S. Poria, E. Cambria, and L. Morency, "Tensor fusion network for multimodal sentiment analysis," *CoRR*, vol. abs/1707.07250, 2017.

[4] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2015.

[5] A. Anandkumar, R. Ge, D. J. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," *CoRR*, vol. abs/1210.7559, 2012.

[6] Y. Kwon, Y. Lee, and M. Rhu, "Tensor casting: Co-designing algorithm-architecture for personalized recommendation training," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 235–248.

[7] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights," *Proceedings of the IEEE*, vol. 109, no. 10, pp. 1706–1752, 2021.

[8] E. G. Hohenstein, R. M. Parrish, and T. J. Martínez, "Tensor hypercontraction density fitting. I. Quartic scaling second- and third-order Møller-Plesset perturbation theory," *jcp*, vol. 137, no. 4, pp. 044 103–044 103, jul 2012.

[9] F. Hummel, T. Tsatsoulis, and A. Grüneis, “Low rank factorization of the coulomb integrals for periodic coupled cluster theory,” *The Journal of Chemical Physics*, vol. 146, no. 12, p. 124105, mar 2017.

[10] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, “Parcube: Sparse parallelizable candecomp-parafac tensor decomposition,” *ACM Trans. Knowl. Discov. Data*, vol. 10, no. 1, jul 2015.

[11] F. Yu, H. Cui, and X. Feng, “Vtensor: Using virtual tensors to build a layout-oblivious ai programming framework,” in *2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP)*, 2019, pp. 1–6.

[12] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, “Tie: Energy-efficient tensor train-based inference engine for deep neural network,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 264–277.

[13] D. Abts, J. Ross *et al.*, “Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 145–158.

[14] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, “Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 598–611.

[15] M. Zhang, Z. Hu, and M. Li, “Duet: A compiler-runtime subgraph scheduling approach for tensor programs on a coupled cpu-gpu architecture,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 151–161.

[16] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “Accpar: Tensor partitioning for heterogeneous deep learning accelerators,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 342–355.

[17] H. Fanaee-T and J. Gama, “Tensor-based anomaly detection: An interdisciplinary survey,” *Knowledge-Based Systems*, vol. 98, pp. 130–147, 2016.

[18] N. C. Thompson, K. H. Greenewald, K. Lee, and G. F. Manso, “The computational limits of deep learning,” *CoRR*, vol. abs/2007.05558, 2020.

[19] Ricci and Levi-Civita, “Méthodes de calcul différentiel absolu et leurs applications,” *Mathematische Annalen*, vol. 54, pp. 125–201, 1900.

[20] H. Farias, C. Nuñez, and M. Solar, “Tensorfit a tool to analyse spectral cubes in a tensor mode,” *Astronomy and Computing*, vol. 25, pp. 195–202, 2018.

[21] J. Li, B. Uçar, U. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, “Efficient and effective sparse tensor reordering,” in *Proceedings of the ACM International Conference on Supercomputing*, 2019, p. 227–237.

[22] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, “Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 689–702.

[23] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “Extensor: An accelerator for sparse tensor algebra,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, p. 319–333.

[24] J. Li, Y. Ma, X. Wu, A. Li, and K. Barker, “Pasta: A parallel sparse tensor algorithm benchmark suite,” 2019.

[25] S. Smith, J. Park, and G. Karypis, “Sparse tensor factorization on many-core processors with high-bandwidth memory,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1058–1067.

[26] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, “Model-driven sparse CP decomposition for higher-order tensors,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1048–1057.

[27] Y. Soh, P. Flick, X. Liu, S. Smith, F. Checconi, F. Petrini, and J. Choi, “High performance streaming tensor decomposition,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 683–692.

[28] L. Ma and E. Solomonik, “Efficient parallel CP decomposition with pairwise perturbation and multi-sweep dimension tree,” *CoRR*, vol. abs/2010.12056, 2020.

[29] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar, “On optimizing distributed tucker decomposition for dense tensors,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1038–1047.

[30] V. T. Chakaravarthy, S. S. Pandian, S. Raje, and Y. Sabharwal, “On optimizing distributed non-negative tucker decomposition,” in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS ’19. Association for Computing Machinery, 2019, p. 238–249.

[31] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, P. Murali, S. S. Pandian, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for sparse tensors," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. Association for Computing Machinery, 2018, p. 374–384.

[32] B. Madathil, S. V. M. Sagheer, A. Rahiman, A. J. Tom, B. P. S, J. Francis, and S. N. George, "Tensor low rank modeling and its applications in signal processing," 2019.

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

[34] M. Safayet Hossain, K. M. Azharul Hasan, and T. Tsuji, "Performance analysis of higher order tensor storages for highly sparse multidimensional data," in *2021 5th International Conference on Electrical Information and Communication Technology (EICT)*, 2021, pp. 1–6.

[35] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015.

[36] Q. Sun, Y. Liu, H. Yang, M. Dun, Z. Luan, L. Gan, G. Yang, and D. Qian, "Input-aware sparse tensor storage format selection for optimizing mttkrp," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[37] T. Herault, Y. Robert, G. Bosilca, R. J. Harrison, C. A. Lewis, E. F. Valeev, and J. J. Dongarra, "Distributed-memory multi-gpu block-sparse tensor contraction for electronic structure," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 537–546.

[38] J. Kim, A. Sukumaran-Rajam, C. Hong, A. Panyala, R. K. Srivastava, S. Krishnamoorthy, and P. Sadayappan, "Optimizing tensor contractions in ccsd(t) for efficient execution on gpus," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, p. 96–106.

[39] J. Liu, D. Li, R. Gioiosa, and J. Li, "Athena: High-performance sparse tensor contraction sequence on heterogeneous memory," in *Proceedings of the ACM International Conference on Supercomputing*, 2021, p. 190–202.

[40] J. Liu, J. Ren, R. Gioiosa, D. Li, and J. Li, "Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, p. 318–333.

[41] R. Levy, E. Solomonik, and B. K. Clark, "Distributed-memory DMRG via sparse and dense parallel tensor contractions," *CoRR*, vol. abs/2007.05540, 2020.

[42] J. Li, Y. Ma, C. Yan, and R. Vuduc, "Optimizing sparse tensor times matrix on multi-core and many-core architectures," in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, 2016, pp. 26–33.

[43] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, and R. Vuduc, “Optimizing sparse tensor times matrix on gpus,” *J. Parallel Distrib. Comput.*, vol. 129, no. C, p. 99–109, jul 2019.

[44] L. Jia, Z. Luo, L. Lu, and Y. Liang, “Analyzing the design space of spatial tensor accelerators on fpgas,” in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2021, pp. 230–235.

[45] S. Wijeratne, R. Kannan, and V. Prasanna, “Reconfigurable low-latency memory system for sparse matricized tensor times khatri-rao product on fpga,” 2021.

[46] R. Hu, W. Yang, X. Zhou, K. Li, and K. Li, “Performance analysis and optimization for mttkrp of sparse tensor on cpu and gpu,” in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2020, pp. 545–550.

[47] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan, “Load-balanced sparse mttkrp on gpus,” in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 123–133.

[48] A. Gonzalez, “Trends in processor architecture,” 2018.

[49] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[50] M. Arora, “The architecture and evolution of cpu-gpu systems for general purpose computing,” *By University of California, San Diago*, vol. 27, 2012.

[51] A. Boutros and V. Betz, “Fpga architecture: Principles and progression,” *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.

[52] R. Reyes, G. Brown, R. Burns, and M. Wong, “Sycl 2020: More than meets the eye,” in *Proceedings of the International Workshop on OpenCL*, 2020.

[53] G. K. Reddy Kuncham, R. Vaidya, and M. Barve, “Performance study of gpu applications using sycl and cuda on tesla v100 gpu,” in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

[54] J. Li, J. Sun, and R. Vuduc, “Hicoo: Hierarchical storage of sparse tensors,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 238–252.

[55] A. E. Helal, J. Laukemann, F. Checconi, J. J. Tithi, T. Ranadive, F. Petrini, and J. Choi, “Alto,” in *Proceedings of the ACM International Conference on Supercomputing*, 2021.

[56] A. Nguyen, A. E. Helal, F. Checconi, J. Laukemann, J. J. Tithi, Y. Soh, T. Ranadive, F. Petrini, and J. W. Choi, “Efficient, out-of-memory sparse mttkrp on massively parallel architectures,” 2022.

[57] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, p. 65–76, apr 2009.

[58] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, “Applying the roofline model,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 76–85.

[59] J. Czaja, M. Gallus, J. Wozna, A. Grygielski, and L. Tao, “Applying the roofline model for deep learning performance optimizations,” 2020.

[60] A. Ilic, F. Pratas, and L. Sousa, “Cache-aware roofline model: Upgrading the loft,” *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.

[61] A. Lopes, F. Pratas, L. Sousa, and A. Ilic, “Exploring gpu performance, power and energy-efficiency bounds with cache-aware roofline modeling,” in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 259–268.

[62] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev, “Performance analysis with cache-aware roofline model in intel advisor,” in *2017 International Conference on High Performance Computing and Simulation (HPCS)*, 2017, pp. 898–907.

[63] A. Ilic, F. Pratas, and L. Sousa, “Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores,” *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 52–58, 2017.

[64] D. Marques, A. Ilic, Z. A. Matveev, and L. Sousa, “Application-driven cache-aware roofline model,” *Future Generation Computer Systems*, vol. 107, pp. 257–273, 2020.

[65] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[66] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell, “Toward an architecture for never-ending language learning.” in *AAAI*, vol. 5, 2010, p. 3.

[67] M. Whiting, K. Cook, G. Grinstein, J. Fallon, K. Liggett, D. Staheli, and J. Crouser, “Vast challenge 2015: Mayhem at dinofun world,” in *Visual Analytics Science and Technology (VAST), 2015 IEEE Conference on*. IEEE, 2015, pp. 113–118.

[68] S. Smith and G. Karypis, “SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit,” http://cs.umn.edu/~splatt/, 2016.

[69] J. Li, Y. Ma, and R. Vuduc, “ParTI! : A parallel tensor infrastructure for multicore cpus and gpus,” Oct 2018, last updated: Jan 2020. [Online]. Available: http://parti-project.org

[70] J. McAuley and J. Leskovec, “Hidden factors and hidden topics: understanding rating dimensions with review text,” in *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 2013, pp. 165–172.