ABSTRACT

MANTHENA, SAI KRISHNA TEJA VARMA. Parallel Sparse Tensor-times-Vector on Cerebras WSE-2. (Under the direction of Dr. Jiajia Li).

Sparse tensor algebra plays a crucial role in various domains such as scientific computing, machine learning, and data analytics, particularly in computations like Sparse Tensor-times-Vector (SpTTV), essential for popular tensor decompositions such as Tucker decomposition and Tensor Power Iteration. This work delves into the optimization of the SpTTV kernel specifically on the Cerebras Wafer-Scale Engine (WSE), a powerful architecture initially designed for machine learning training but increasingly recognized for its versatility across different workloads. Our study presents an optimized implementation of the SpTTV kernel, tailored specifically for the unique capabilities of the Cerebras WSE. We benchmarked this implementation against the conventional OpenMP-based PASTA library's version, highlighting the enhanced computational efficiency made possible by the Cerebras system. The findings from this research not only demonstrate the feasibility of utilizing the Cerebras WSE for high-performance sparse tensor operations. Moreover, the insights gained lay a promising foundation for extending these optimizations to the Sparse Tensor-Times-Matrix (SpTTM) kernel, potentially broadening the scope of applications and further accelerating the performance of tensor-based computations.

Parallel Sparse Tensor-times-Vector on Cerebras WSE-2

by
Sai Krishna Teja Varma Manthena

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina
2024

APPROVED BY:

Dr. Frank Mueller

Dr. Huiyang Zhou

Dr. Jiajia Li
Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

Sai Krishna Teja Varma is currently a Master's student in the Computer Science Department at North Carolina State University, specializing in high-performance computing and numerical linear algebra kernels for GPUs and AI accelerators, advised by Dr. Jiajia Li. Before this, Teja completed a Bachelor's degree in Information Technology from B. V. Raju Institute of Technology of Jawaharlal Nehru Technological University, Hyderabad, where they developed a strong foundation in the field of Computer Science.

ACKNOWLEDGEMENTS

I would like to extend my deepest gratitude to all those who made the completion of this thesis possible. First and foremost, I am immensely thankful to my thesis advisor, Dr. Jiajia Li, whose expertise, understanding, constant trust, and patience, added considerably to my graduate experience. Your guidance helped me at all times in the research and writing of this thesis. I would also like to thank the members of my thesis committee, Dr. Frank Mueller and Dr. Huiyang Zhou, for their insightful comments and encouragement, as well as for the challenging questions that incited me to widen my research from various perspectives.

My sincere thanks also goes to the Department of Computer Science at North Carolina State University for providing me with the necessary resources and environment to complete my research.

A special thank you to the technical staff of Cerebras Systems and the Developer Community, whose assistance was invaluable in conducting the experiments necessary for this thesis. Their support and willingness to share their vast knowledge have been greatly appreciated.

I must express my very profound gratitude to my parents and my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you. Finally, I would like to thank all those who were involved in any capacity in the realization of this thesis. Your contributions are sincerely appreciated and have made my graduate experience both possible and rewarding.

This project was supported in part by North Carolina State University's Faculty Startup Fund of the Computer Science Department. This project was also supported by National Science Foundation (Award Numbers 2247309 and 2316201). This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Sparse tensor algebra serves as a fundamental computational framework in numerous critical domains, including scientific computing, machine learning (Novikov et al. (2015); Abadi et al. (2016)), natural language processing, social network analysis (Papalexakis et al. (2012))and healthcare analytics (Wang et al. (2015); Ho et al. (2014); Cichocki (2013)). Within this framework, Sparse Tensor-times-Vector (SpTTV) operations stand out as essential kernels for various tensor decompositions, such as Tucker decomposition and Tensor Power Iteration. The TTV operation plays a crucial role in Tensor Power Iteration because it efficiently computes the matrix-vector multiplication involving the tensor, which is essential for updating the estimate of the dominant eigenvector in each iteration. Optimizing this kernel is imperative to enhance the efficiency and scalability of tensor-based algorithms.

In recent years, the emergence of specialized architectures has offered new avenues for accelerating sparse tensor computations. Among these, the Cerebras Wafer-Scale Engine (WSE-2) (Lie (2022)) has a powerful architecture with 850,000 processing elements (PE) structured in a 2D grid, where each PE consists of a router connected to a processor with 48KB of local memory. The absence of shared memory and the powerful memory bandwidth offers opportunities for optimization in communication models among processing elements (PEs), workload distribution strategies, and the workflow of tensor factorization algorithms to better allocate them on Cerebras PEs. Cerebras WSE-2 claims that it supports native unstructured sparsity harvesting by filtering out zeros (Lie (2022)). Although primarily used for machine learning training, Cerebras WSE-2 has seen increasing demand for other workloads (Jacquelin et al. (2022); Rocki et al. (2020)).

Research Hypothesis: This research tends to investigate the potential of Cerebras WSE-2 Architecture in performing the tensor operations via SpTTV.

We employ a COO-based approach, distribute the workload amongst the PEs, and obtain the results. The idea of choosing this kernel involves incrementally developing an optimal SpTTV kernel for the architecture leading to efficiently deploying the Sparse Tensor times Matrix (SpTTM) kernel, which is one of the most important kernels in the field of Numerical Linear Algebra for workloads such as Tucker Decomposition.

The subsequent chapters of this thesis are systematically arranged to cover the entire scope of the research, from foundational concepts to advanced implementations and evaluations:

- **Chapter 2, Background**: This chapter introduces the fundamental concepts essential to understanding the rest of the thesis. It covers tensors, the Tensor-Times-Vector (TTV) kernel, and various sparse storage formats. These foundational elements are crucial for appreciating the later discussions on specific optimizations and architectural adaptations.

- **Chapter 3, Cerebras Systems**: A detailed examination of the Cerebras WSE-2 is presented in this chapter. It delves into the architecture of the WSE-2, highlighting its novel processing elements and the efficient routing capabilities integral to its design. Additionally, the chapter explores the programming model specific to Cerebras, which leverages unique hardware features to enhance computational performance.

- **Chapter 4, Optimizing SpTTV on Cerebras WSE-2:** This chapter is dedicated to our contributions, detailing three incremental approaches to optimizing the SpTTV kernel for the Cerebras architecture. It discusses the specific sparse storage formats utilized, the distribution of the computational workload, and the algorithms developed to leverage the architecture efficiently. This chapter is crucial as it bridges the gap between the theoretical background and practical implementation.

- **Chapter 5, Preliminary Results:** Here, the outcomes of the implemented optimizations are presented. This chapter assesses the performance improvements, scalability, and efficiency gains achieved through the optimizations described in the previous chapter. It provides a critical evaluation of the results against the objectives set out at the beginning of this research.

- **Chapter 6, Conclusions and Observations:** This chapter synthesizes the findings of the thesis, offering a reflective look at the implications of the research and the insights gained from working with the Cerebras WSE-2. It discusses the broader impact of the research findings and their relevance to the field of high-performance computing.
- **Chapter 7, Future Work:** The final chapter outlines potential areas for further research and exploration based on the findings and experiences documented in the thesis. It suggests directions for future studies to build upon the groundwork laid by this research, aiming to further enhance the performance and applicability of SpTTV operations in other architectures or applications.

CHAPTER

2

BACKGROUND

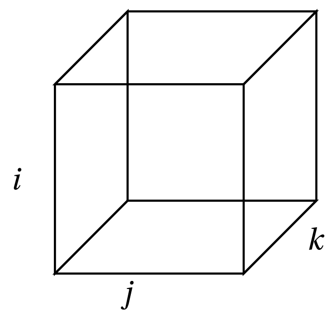
2.1 Introduction to Tensors

Tensors are multi-dimensional arrays that generalize vectors and matrices to higher dimensions, providing a powerful framework for data representation and manipulation across various scientific and engineering disciplines. The order of a tensor indicates the number of *dimensions or modes(N)* it possesses:

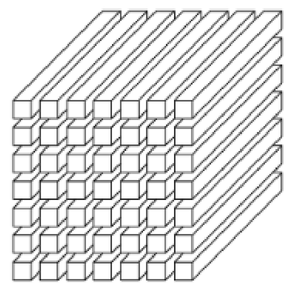
- A first-order tensor($N = 1$) is a vector($\mathbf{v}$), characterized by a single dimension.
- A second-order tensor($N = 2$) is a matrix($\mathbf{A}$), which includes two dimensions: rows and columns.
- Third-order tensors and beyond involve three or more indices($N >= 3$), which are essential for representing more complex data structures, are high-order tensors ($\mathscr{X}$).

An N-way or Nth-order tensor is defined as an element formed by the tensor product of N vector spaces, where each vector space is equipped with its own coordinate system. Visualizing tensors beyond the second order involves conceptual models rather than direct graphical representations, for example, a third-order tensor can be envisioned as a cube of data, where each slice of the cube represents a matrix while, a fourth-order tensor extends this concept by adding additional cubes, each representing different layers or aspects of the data.

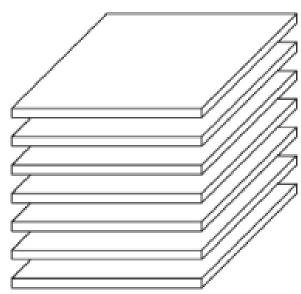
It is essential to consider their density, which refers to the proportion of non-zero entries within the tensor. A *dense* tensor indicates that either all or the majority of its entries are non-zero, while a *sparse* tensor signifies that most of its entries are zeros. The implication of sparsity is significant, as it enables the implementation of operations on the tensor using a compact data structure. This structure efficiently avoids storing and performing operations on the zero-valued entries, thereby optimizing computational resources and speeding up operations.



(a) A *third-order tensor*, $\mathcal{X}_{ijk}$



(b) *Mode-3 (tube)* fibers: $\mathbf{f_{ij:}} = \mathcal{X}_{i,j,:}$



(c) *Horizontal Slices*: $\mathcal{X}_{:,:,k}$

Figure 2.1: A third-order tensor $\mathcal{X} \in R^{I \times J \times K}$ representations

The figure 2.1 gives several representations of a third-order tensor. Within the domain of tensor algorithms, various operations are tailored to specific subsets of tensors. Among these subsets is the mode-n fiber, as depicted in figure 2.1b. This fiber represents a vector obtained by holding all indices constant except for those of mode n. For instance, the mode-3 fiber, also called as tube, of a tensor $\mathscr{X}$ is symbolized by the vector $\mathbf{f}_{ij:} = \mathscr{X}_{(i,j,:)}$, where the colon encompasses all indices of mode 3. Additionally, a slice, illustrated in figure 2.1c, refers to a two-dimensional cross-section, or matrix, of a tensor. It is extracted by fixing the indices of all modes except two, as in $\mathscr{X}_{(:,:,k)}$. Moreover, certain tensor algorithms manipulate an equivalent matrix formed by reshaping the tensor. Matricization transforms a tensor into a corresponding matrix by arranging all mode-n fibers as the columns of the matrix. For example, performing mode-3 matricization on a tensor $\mathscr{X} \in R^{4\times4\times4}$ yields a matrix $\mathscr{X}_{(1)} \in R^{4\times16}$. The survey done by Kolda and Bader (2009) gives a detailed description about the tensors.

2.2 Tensor Times Vector(TTV)

Within the framework of computations performed on tensors, product operations are foundational pillars. They constitute the tensor equivalent of matrix products, embodying the fundamental principles of multiplication inherent to classical linear algebra.

The Tensor-Times-Vector (TTV) operation in mode n multiplies a tensor $\mathscr{X}$ with a vector $\mathbf{v}$ along mode n. The resulting tensor $\mathbf{Y}$ has one less dimension than the original tensor $\mathscr{X}$. The equation can described as:

$$\mathbf{Y} = \mathscr{X} \times_n \mathbf{v}, \tag{2.1}$$

where $\mathscr{X} \in \mathbb{R}^{I_1\times\cdots\times I_n\times\cdots\times I_N}$ is a tensor, $\mathbf{v} \in \mathbb{R}^{I_n}$ is a vector, and $\mathbf{Y} \in \mathbb{R}^{I_1\times\cdots\times I_{n-1}\times I_{n+1}\times\cdots\times I_N}$ is the resulting tensor. The TTV is analogues to a matrix-vector product, thus matricizing the tensor along a mode and performing the matrix-vector product would give us the result matrix, $\mathbf{Y}$.

In the context of Tensor Power Iteration, the TTV (Tensor-times-Vector) kernel plays a central role. The significance of Tensor Power Iteration lies in its ability to efficiently compute dominant eigenvectors, which are crucial for various data analysis tasks. These tasks include dimensionality reduction, feature extraction, clustering, and anomaly detection, among others. Dominant eigenvectors capture the most significant patterns or structures within the data, making them invaluable for understanding and interpreting high-dimensional datasets. The TTV operation efficiently computes the product of a tensor and a vector, which is a core step in each iteration of the algorithm. This operation enables the algorithm to update the estimate of the dominant eigenvector iteratively. By efficiently performing TTV multiplications, Tensor Power Iteration can converge to accurate solutions even for large-scale tensors with sparse or high-dimensional data. In various real-world scenarios, datasets are often sparse, hence we dive into the Sparse Tensor-times-Vector (SpTTV) kernel.

2.3 Sparse Tensor Formats

In many real-world applications, data naturally occurs in highly sparse fashion. This is common in areas such as image processing, where large regions of uniform color result in zeros, or in machine learning, where features are often absent. Efficiently storing and processing this sparse data is crucial to optimize both storage use and computational resources. To address this, sparse tensor formats, which store only nonzero elements and their indices, are utilized, reducing both the storage requirements and the computational complexity of operations.

2.3.1 Coordinate Format(COO)

The Coordinate (COO) format is straightforward, storing each non-zero element of the tensor along with its coordinates, making it ideal for tensors with a small number of non-zero elements or where elements are dynamically added. In the COO format, each non-zero element is recorded with its full coordinate set, making it intuitive. The non-zero elements of a tensor are explicitly stored along with their coordinates in a pair of arrays: **inds** for indices and **val** for values. The **val** array is a size-M array where M is the number of non-zero entries, consisting of floating-point numbers that represent the values of these non-zero elements. Correspondingly, **inds** is a size-M array, but each entry is a tuple of integers that denote the coordinates of each non-zero element in the tensor, or it could be segregated into three arrays namely **i**, **j**, and **k** to store the corresponding indices of the non-zero value.

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

Figure 2.2: COO Format

The figure 2.2 describes an example of $4 \times 4 \times 4$ tensor in COO format, where the **i**, **j**, **k** refer to the index arrays and the **val** refers to the value array of the tensor.

2.3.2 Compressed Sparse Fiber Format(CSF)

Compressed Sparse Fiber (CSF) described in Smith and Karypis (2017) extends the principles of compressed sparse row (CSR) or column (CSC) formats to tensors, optimizing the storage by compressing along one dimension and using pointers to indicate the start of data blocks along this dimension. This method is particularly beneficial for tensors where non-zero elements are structured in a predictable pattern, facilitating faster access for many tensor operations such as slicing. CSF is designed to efficiently compress and access data along one or more dimensions of a tensor, which enhances performance for a variety of tensor operations. In CSF, the tensor's non-zero elements are stored along with their indices, similar to COO, but with a crucial difference: CSF compresses these indices across specified dimensions. This is achieved by separating the indices into distinct arrays for each dimension and introducing an additional array of pointers that indicates the starting positions of new data blocks in the next dimension's array. For each non-zero element, the **val** array stores its value, just as in COO. The indices, however, are organized into multiple arrays based on the tensor's dimensions — for example, one array for each mode **i**, **j**, **k**. The pointer arrays then guide the traversal through these index arrays, significantly reducing the number of accesses required to locate a specific slice or fiber of the tensor.

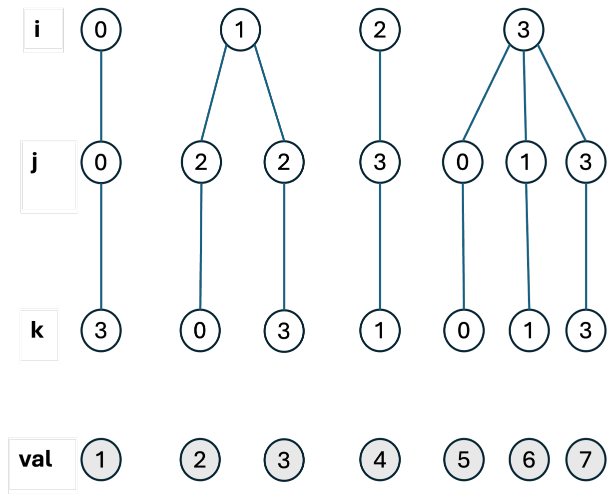


Figure 2.3: CSF Format

The figure 2.3 describes an example of $4 \times 4 \times 4$ tensor in CSF format, as described above for COO.

CHAPTER

3

CEREBRAS WSE-2

3.1 The Chip

The Cerebras WSE-2 comprises 850,000 Processing Elements structured in a 2D grid. Each PE contains a general-purpose compute element (CE), a fabric router, and 48kB of local SRAM memory with single-cycle read/ write access latency. PE-to-PE communication latency is also one cycle. The PE contains a network router with links to the CE and to the routers of the four nearest PEs in the north, south, east, and west directions. Communication is integrated into the instruction set, at single 32-bit word granularity, and is accordingly as fast as arithmetics.

Uniquely, the WSE is composed of a 7×12 array of identical "dies," each containing thousands of PEs. Unlike traditional chips, which are fabricated by slicing a wafer into individual dies, the WSE maintains continuous interconnects across these dies. This innovative approach allows the WSE to extend the inter-die connections, creating a single, unified wafer-scale processor. This design substantially increases the processor size, making it several times larger than the largest processors available on the market at its introduction. This extensive integration and scale enable unparalleled processing power and efficiency

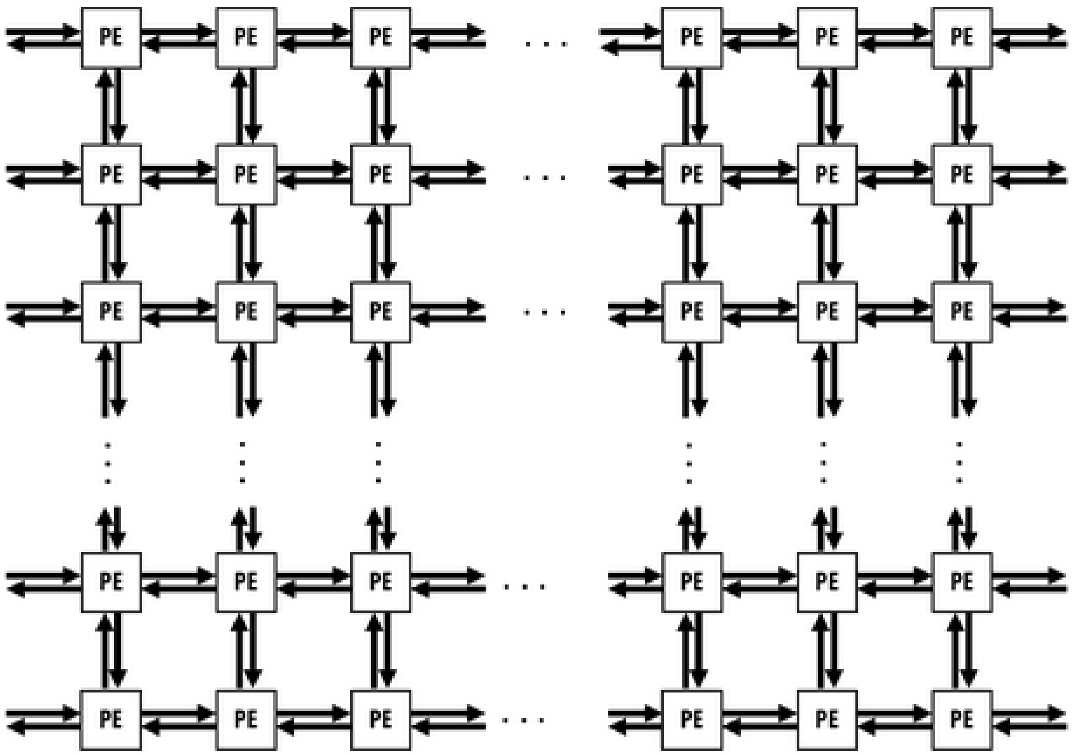


Figure 3.1: A 2D mesh of Processing Elements

for complex computational tasks.

The WSE (Wafer Scale Engine) represents a novel approach to parallel computing by incorporating the entire distributed memory machine within a single wafer, breaking away from traditional multi-chip architectures. This design eliminates the need for a cache hierarchy or shared memory, which are staples in conventional computing systems. Each processing element (PE) within the WSE has its own local memory, ensuring rapid data access and minimizing latency typically associated with hierarchical memory systems. the WSE's architecture ensures that each PE can operate independently yet communicate efficiently through a directly connected mesh network. This setup is akin to having a massive multi-core processor where each core is directly connected to its neighbors without the intermediation of shared caches or memory.

The lack of shared memory and cache hierarchies in the WSE simplifies the architecture but requires that data management be handled more explicitly at the software level,

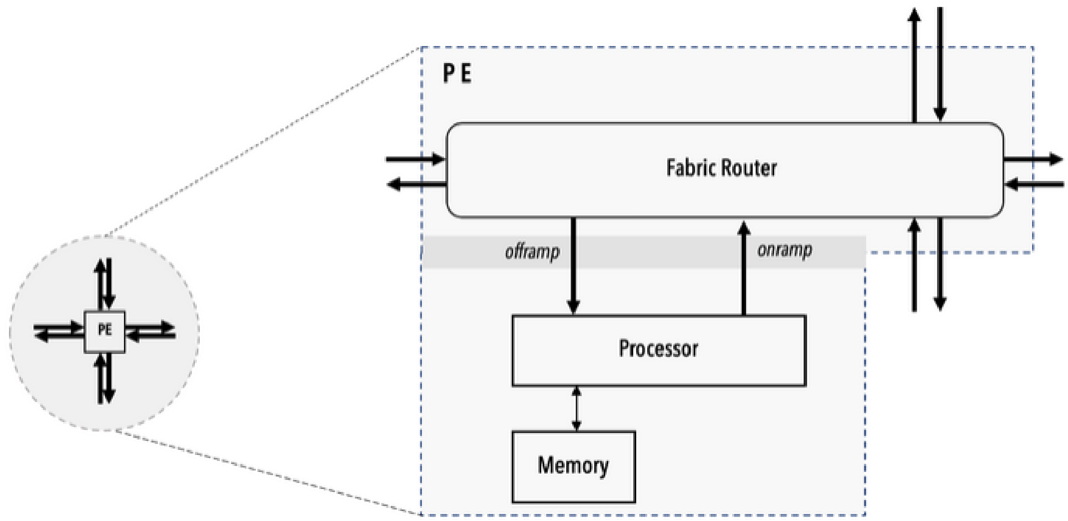


Figure 3.2: PE and its components

promoting a model where data locality is paramount. The direct connectivity among PEs in a 2D mesh mimics the behavior of a tightly coupled distributed system but without the traditional overhead associated with complex memory management protocols and inter-chip communications. This leads to a significant reduction in the data access times and increases the efficiency of data processing, making the WSE particularly well-suited for tasks requiring high parallelism and data throughput, such as deep learning computations and large-scale scientific simulations. The figures 3.1 and 3.2 give an overview of the Wafer-Scale Engine 2 and the Processing Elements, originally described in the whitepaper (Selig (2022)).

3.2 Routing

The Cerebras WSE employs a sophisticated packet routing mechanism that utilizes a color-coded system for managing data flow across its network of routers. Each packet, referred to as a wavelet, is tagged with a 5-bit virtual channel identifier known as a color. This identifier is crucial for directing the wavelet through the network, guiding it along predefined paths to its intended destination.

Routers within the WSE are capable of handling wavelets in five possible directions: NORTH, EAST, SOUTH, WEST, and RAMP, the latter being directly associated with the processor itself. During the compilation phase, routing configurations are established that specify which directions can accept and propagate wavelets of each specific color. These configurations ensure that wavelets travel through the network efficiently; however, if a wavelet arrives from a direction that is not configured to accept its color, it will stall, temporarily halting its progression until the correct path becomes available.

Each router holds up to four different configurations for handling wavelets of any given color, allowing for flexibility in routing paths based on network conditions or task requirements. This setup permits each color to be associated with any subset of the router's five outgoing links, facilitating a highly customizable network topology. Routes are typically configured statically at compile time, but they can also be dynamically altered during program execution through the use of control wavelets. This dual capability of static and dynamic reconfiguration provides significant adaptability in managing data flows, especially under varying computational loads or when network conditions change.

Furthermore, for the efficient communication of sparse data, the system employs a method where each wavelet encapsulates sparse vector information in a compact format—specifically, a 16-bit index and a 16-bit data field. This design allows each wavelet to carry both the index $\mathbf{i}$ and the value $\mathbf{x}(\mathbf{i})$ of a nonzero element, optimizing the transmission of sparse data across the network. This approach is particularly beneficial in applications dealing with large-scale sparse datasets, where maintaining data compactness and ensuring efficient routing is paramount.

3.3 Programming Model

The Cerebras WSE operates on a dataflow architecture, where tasks are triggered by incoming wavelets at processing elements (PEs). The sequence and timing of these wavelets determine the order of task execution, which can vary depending on their arrival sequence. Tasks can also be triggered at compile time or activated by other tasks, adding a layer of dynamic execution that depends on the computational workflow. To streamline operations, the WSE employs Data Structure Descriptors (DSDs). DSDs are crucial for representing segments of memory or sequences of wavelets, either incoming or outgoing. By abstracting complex operations into DSDs, the system can condense repetitive tasks into single hardware instructions, enhancing performance significantly. For example, a task involving the reception of 'B' wavelets where each ith wavelet is stored at an address offset by i×100i×100 can be efficiently handled by one DSD, rather than iteratively through a traditional for loop. This method not only speeds up processing but also simplifies coding. Moreover, DSDs support asynchronous operations, enabling the launch of microthreads that execute these descriptors independently. Once a microthread completes its task, it can activate subsequent tasks, facilitating a chain of operations. This feature allows for the concurrent execution of multiple DSDs—meaning they operate sequentially across different cycles, but not simultaneously—enhancing throughput by allowing continuous task activation without idle cycles.

The instruction set of the WSE is specifically designed to manipulate vectors and objects of higher dimensionality. This functionality is enabled by the DSDs, which define how to access and manipulate specific data structures, incorporating parameters like address, length, and stride. These descriptors ensure that each data interaction is optimized for the architecture, allowing for high-performance computation across the diverse and complex tasks typical in large-scale AI applications. This strategic use of DSDs underpins the WSE's ability to handle vast datasets and complex models, pushing the boundaries of what's possible in advanced computational environments.

CSL (Cerebras Systems Language) is used for programming the WSE-2. It merges low-level access to the hardware's instruction set and structure with high-level features like structured control flow and automatic register allocation, making it suitable for both direct hardware manipulation and more abstract programming tasks. Notably, CSL supports compile-time code execution for constant inputs, a feature derived from Zig. The language is statically typed, with a syntax familiar to C/C++ programmers, and supports a range of data types along with common programming constructs like loops and conditional statements. CSL introduces the concept of "tasks," which are triggered by data wavelets and function similarly to interrupts but without suspending ongoing tasks, thus maintaining efficiency. It also employs Data Structure Descriptors (DSDs) and Data Structure Registers (DSRs) to facilitate complex data operations on tensors, reducing runtime costs and enhancing performance for data-intensive computations.

CHAPTER

4

PARALLEL SPTTV ON CEREBRAS WSE-2

4.1 Introduction

We delve into the optimization of the Sparse Tensor-Times-Vector (SpTTV) operation, a crucial component for tensor computations, specifically tailored for the high-performance capabilities of the Cerebras WSE-2. The unique architecture of the Cerebras WSE-2, featuring a vast array of interconnected processing elements (PEs), presents both opportunities and challenges for implementing efficient tensor operations. The primary focus here is to exploit the architecture's strengths—such as its massive parallelism and specialized memory hierarchy—while mitigating limitations related to data sparsity and communication overhead. Recognizing the importance of SpTTV in various high-dimensional data applications, from deep learning to data analytics, this chapter outlines three incremental strategies employed to optimize this kernel. Each approach is designed to leverage the aspects of the Cerebras system, from its memory management to its processing capabilities, ensuring that SpTTV operations are not only faster but also more energy-efficient and scalable.

The culmination of these strategies aims not only to refine SpTTV operations but also to set a robust groundwork for extending these optimizations to the more complex Sparse Tensor-Times-Matrix (SpTTM) kernel. The optimizations for SpTTV are expected to translate effectively to SpTTM, considering the similar computational and data movement challenges involved.

4.2 COO-based Implementation

This work uses a COO-based approach to perform SpTTV. The Coordinate (COO) format is fundamentally straightforward: it stores each non-zero element of a sparse tensor along with its multidimensional coordinate. For instance, if a tensor element at coordinates $(\mathbf{i}, \mathbf{j}, \mathbf{k})$ holds a value (v), COO stores this as $(\mathbf{i}, \mathbf{j}, \mathbf{k}, \mathbf{v})$ in different arrays. This method provides several distinct advantages:

- As we are trying to not to address any structure in the sparsity pattern. The adaptability of COO to various sparsity patterns without the need for restructuring the data format is a significant advantage. The WSE-2's design, which supports flexible and dynamic data processing, aligns well with COO's adaptability, enabling it to handle diverse and changing sparsity patterns effectively.

- COO's structure allows for direct access to the values and their coordinates without the need for additional computations to determine their positions which is beneficial for the Cerebras WSE-2, as its architecture can leverage direct access to streamline the processing of non-zero elements, thereby minimizing latency and maximizing throughput. Each processing element (PE) on the WSE-2 can retrieve and compute operations directly from the COO stored data without navigating through zeros or unnecessary data points.

- The COO format's indexing system is inherently simple because it directly maps each non-zero value to its coordinates in the tensor. This simplicity is advantageous for the WSE-2, which can utilize its powerful routing capabilities to quickly transfer and process data. The absence of complex indexing calculations reduces the computational load on the system, allowing the PEs to focus on performing meaningful computations rather than deciphering tensor layouts.

- In the WSE-2, memory efficiency is crucial due to the absence of shared memory and the localized memory within each PE. COO format, by storing only non-zero elements, ensures that the system's memory is used efficiently. The data transferred between PEs predominantly consists of meaningful information (non-zeros and their coordinates), which optimizes the use of the interconnect bandwidth and reduces the overhead of processing large volumes of sparse data. This flexibility ensures that as tensor operations evolve or as the sparsity pattern of the data changes, the underlying data handling and processing architecture does not need significant adjustments.

- Since each entry in COO is independent of the others, the WSE-2 can process multiple entries in parallel without needing synchronization mechanisms to manage dependencies. This capability is essential for scaling up operations and efficiently utilizing the extensive array of PEs on the WSE-2, thereby enhancing overall performance and reducing computation times for large-scale tensor operations.

Thus, the COO format's straightforward, efficient, and flexible characteristics make it highly suitable for the Cerebras WSE-2's architecture, enabling it to perform sparse tensor operations with reduced computational overhead and improved efficiency. This alignment between the data format and hardware architecture is crucial for achieving high performance in the processing of large and complex tensor-based computations.

4.3 Parallel SpTTV Implementation

The implementation is developed in Cerebras Software Language (CSL) of the Cerebras SDK, first described in Selig (2022).

4.3.1 Strategy 1: NNZ distribution with reduction of Y

The primary strategy for optimizing SpTTV on the Cerebras WSE-2 takes advantage of the parallel processing power by distributing the workload across the processing elements (PEs). In this approach, the tensor $\mathscr{X}$, stored in the COO format, is partitioned along with its indices vectors $\mathbf{i}, \mathbf{j}, \mathbf{k}$ and the value vector **val**. These partitions are then disseminated across the PEs, ensuring an even distribution of computational tasks, i.e. the Non-zeroes are equally distributed among the PEs. Simultaneously, the vector $\mathbf{v}$, with which the tensor is to be multiplied, is copied to the first PE. Leveraging the high-speed inter-PE communication capabilities of the WSE-2, it is then broadcasted to the other PEs. This process is optimized on the WSE-2 due to its single-cycle communication latency, which facilitates rapid and synchronous data transfer across the processor's mesh network. Each PE independently computes its segment of the resultant tensor(second-order, matrix) **Y** against the broadcasted vector **v**. These computations exploit the local SRAM within each PE, which provides single-cycle access latency, enabling each PE to swiftly fetch and store the necessary elements for computation. This is particularly advantageous given the WSE-2's absence of shared memory, emphasizing the importance of local memory in reducing communication overhead and thereby accelerating the computation. After the PEs compute their respective segments of the resultant tensor **Y**, the partial results are efficiently reduced into the final PE, culminating in the full resultant matrix **Y**. This reduction step is inherently parallel and benefits from the fabric's low-latency, ensuring that the final aggregation is both timely and efficient. The final result **Y**, now in its dense form, is transferred to the host, concluding the operation. This method capitalizes on the WSE-2's strengths: the high-density PE array, the low-latency local memory access, and the fabric's efficient data routing. These capabilities allow for a high-performance implementation of the SpTTV operation that is well-suited to the architecture of the WSE-2.

Algorithm 1 SpTTV on Cerebras

Input: Vector **v**, Tensor $\mathscr{X}$ in COO format using vectors $\mathbf{i}, \mathbf{j}, \mathbf{k}, \mathbf{val}$
Output: Resultant tensor **Y**
Partition $\mathscr{X}$: Divide $\mathscr{X}$ into P partitions, $\{\mathscr{X}_p\}_{p=1}^{P}$
Copy $\mathscr{X}$: Distribute the equal partitions of $\mathscr{X}$ in each PE
Copy v: Copy Vector **v** from host to PE1.
Broadcast v: Distribute a copy of **v** to each PE from PE1.
for $p = 1$ **to** P **do**
    Partial $\mathbf{Y}_p$ **calculation**: Compute the partial SpTTV of PE p to **Y** using $\mathbf{X}_p$.
end for
Reduce Y: Combine partial contributions $\{\mathbf{Y}_p\}_{p=1}^{P}$ into final **Y** in the last PE.
Copy Y to Host: Copy **Y** from the last PE to the host.

As detailed in Algorithm 1, the Tensor $\mathscr{X}$ in COO format with index vectors and the value vector $\mathbf{i}, \mathbf{j}, \mathbf{k}, \mathbf{val}$ respectively are first partitioned and distributed among all the PEs. Each PE then receives a copy of the entire vector **v**. The key step involves each PE making its partial contribution by performing the SpTTV kernel. These partial contributions are subsequently reduced to the final PE to form the final result, which is then transferred to the host. In the figure 4.1, vector **v** sits atop the diagram and is initially copied to the first of four horizontally aligned Processing Elements (PEs) on the Cerebras WSE-2; it is then broadcasted to the subsequent PEs. Each PE is allocated an equal segment of the sparse tensor $\mathscr{X}$ indices $\mathbf{i}, \mathbf{j}, \mathbf{k}$ and values **val**, using the COO format for storage. The PEs compute

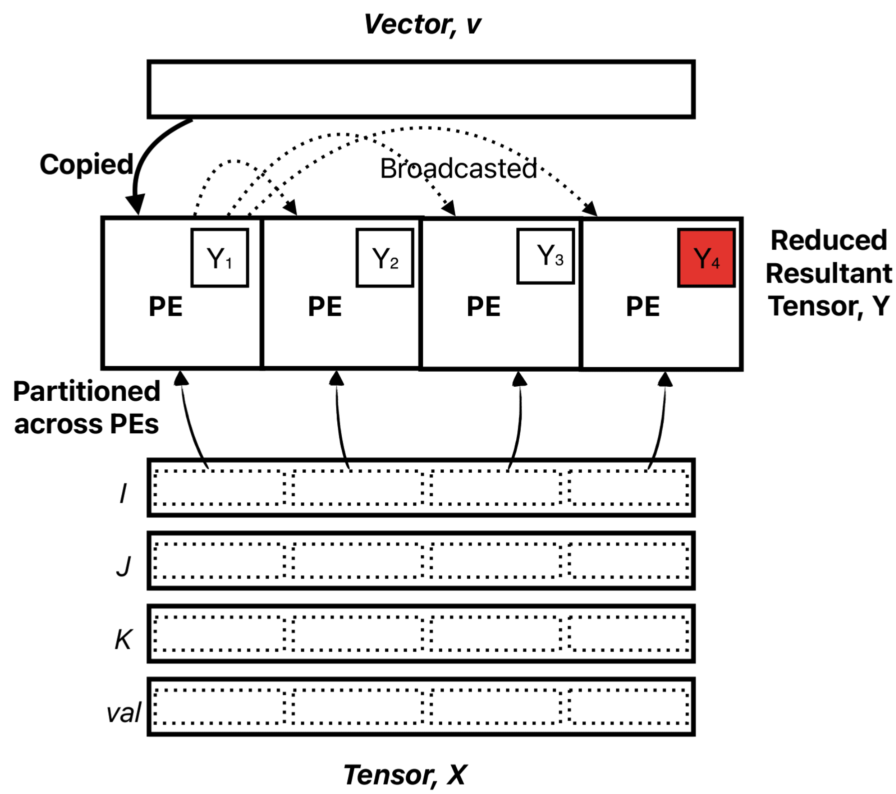


Figure 4.1: An Example of four PEs for Startegy 1.

parts of the resultant tensor **Y** independently, leveraging the WSE-2's local memory for rapid access, with the computation paralleled by the distribution of $\mathscr{X}$ and the broadcasting of **i**, **j**, **k**. The final PE in the sequence is emphasized to indicate its role in the reduction step, aggregating the segments into the dense form of **Y**, completing the SpTTV operation. This figure encapsulates the synergy between the WSE-2's architectural strengths and the SpTTV operation's parallel computation, resulting in an optimized execution path.

4.3.2 Strategy 2: NNZ distribution with partial reduction of Y

In the second strategy, an evaluation of the sparsity of the resultant tensor **Y** leads to a crucial adjustment: instead of storing **Y** in a dense format, it is now stored in the Coordinate (COO) format. This shift acknowledges that the tensor **Y**, while reduced in dimensionality through the TTV operation, still retains a level of sparsity that requires an efficient storage approach. Leveraging the COO format for **Y** enables a more efficient distribution of the computational workload across the PEs on the Cerebras WSE-2. The PEs are now tasked with calculating only the non-zero elements of the resultant tensor, significantly reducing unnecessary computations. Each PE computes a segment of $\mathbf{Y}$, with its non-zero elements' indices and values corresponding to the portion of tensor $\mathscr{X}$ and vector $\mathbf{v}$ it has been assigned. The sparse nature of $\mathbf{Y}$ in the COO format also streamlines the process of copying the results back to the host. Only the non-zero elements, along with their indices, are communicated back, rather than an entire dense tensor. This not only minimizes the data transfer volume but also aligns with the WSE-2's architecture, which is adept at handling such sparse data distributions and transfers. This strategy of using the COO format for both the input tensor $\mathscr{X}$ and the resultant tensor $\mathbf{Y}$ leverages the sparsity-oriented design of the WSE-2, ensuring that the PEs' processing power is directed towards meaningful computations. It optimizes both the computational efficiency on the PE array and the communication efficiency with the host system, which are paramount for accelerating sparse tensor operations in high-performance computing environments Furthermore, We introduce uniformity to the computation of the sparse resultant tensor YY on the Cerebras WSE-2 by standardizing the length of the segments calculated by each PE. This is achieved by padding the segments with zeros to match the length of the longest segment, thereby ensuring that all PEs work with and produce output tensors of equal size. This padding not only streamlines data aggregation but also simplifies data transfer to the host, leveraging the WSE-2's efficient handling of sparse data and optimizing overall computational resources.

Algorithm 2 Initialize and Compute Matrix Y in COO Format Across PEs

Input: Tensor $\mathscr{X}$ in COO format using vectors $\mathbf{i}, \mathbf{j}, \mathbf{k}, \mathbf{val}$, number of PEs **P** and vector **v**
Output: Resultant matrix Y in COO format
Partition $\mathscr{X}$: Divide $\mathscr{X}$ into P partitions, $\{\mathscr{X}_p\}_{p=1}^{P}$
Initialize Local Y**:** Initializes local COO format for each PE segment of **Y**.
for $p = 1$ to P **do**
    Identify relevant elements for $\mathbf{Y}_p$ based on $\mathscr{X}_p$ assigned to PE p.
    Initialize arrays $\mathbf{Y}_i$, $\mathbf{Y}_j$, $\mathbf{Y}_{val}$ for each PE p.
    Pad $\mathbf{Y}_i$, $\mathbf{Y}_j$, $\mathbf{Y}_{val}$ to maximum length among PEs.
end for
Copy $\mathscr{X}$: Distribute the equal partitions of $\mathscr{X}$ and **Y** in each PE
Copy v: Copy Vector **v** from host to PE1.
Broadcast v: Distribute a copy of **v** to each PE from PE1.
for $p = 1$ **to** P **do**
    Partial $\mathbf{Y}_p$ **calculation**: Compute the partial SpTTV of PE p to $\mathbf{Y}_p$ using $\mathbf{X}_p$.
end for
Copy Results to Host: Transfer computed segments of $\mathbf{Y}_p$ from each PE to the host.
Aggregate Results: Combine segments to form complete matrix $\mathbf{Y} inthehost$.

The algorithm described in 2 is designed to efficiently compute the Sparse Tensor-Times-Vector (SpTTV) on multiple processing elements (PEs) using a distributed approach. Initially, the tensor XX in COO format, consisting of indices $\mathbf{i}, \mathbf{j}, \mathbf{k}$ and values **val**, is divided equally among **P** PEs to ensure load balance. Each PE is initialized with a local segment of the resultant matrix **Y** in COO format, preparing for computation. The process involves copying the vector **v** from the host to PE1 and then broadcasting it to all other PEs to ensure each has the necessary vector for multiplication. Subsequently, each PE computes its portion of **Y** based on its partition of $\mathscr{X}$. This partial computation is optimized by padding the local **Y** arrays $\mathbf{Y_i}, \mathbf{Y_j}, \mathbf{Y_v}$**al**to uniform lengths across all PEs, enhancing memory alignment and computational efficiency. Finally, the segments of **Y** computed at the host each PE are gathered to form the complete resultant matrix **Y**, completing the distributed computation.

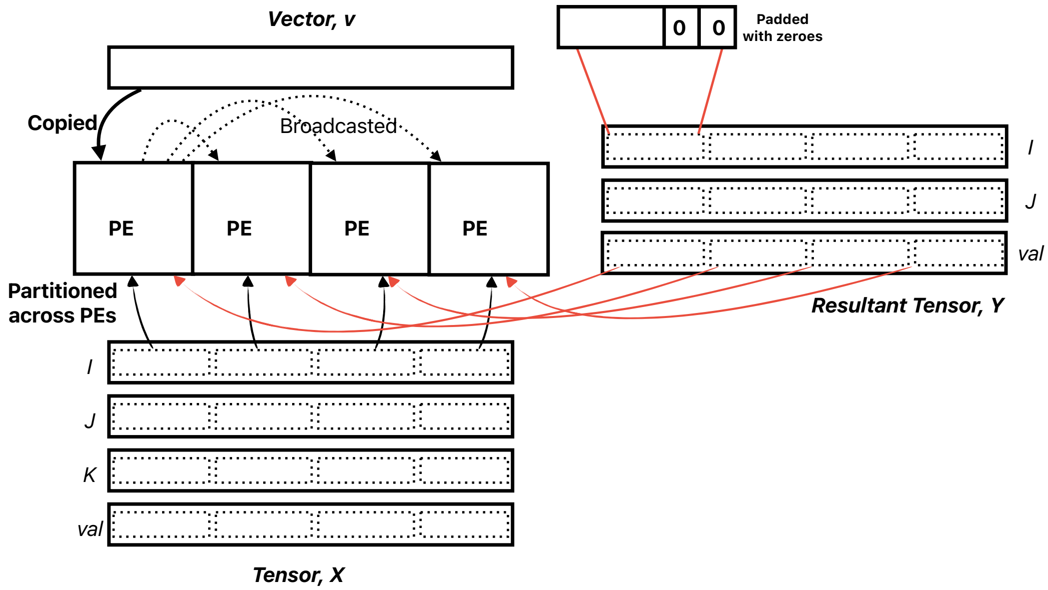


Figure 4.2: An Example of four PEs for Startegy 2.

The algorithm is exemplified in a scenario involving four PEs, illustrated in Figure 4.2, where each PE's responsibilities are clearly delineated, from receiving tensor partitions to contributing to the final output. This figure likely shows the step-by-step data flow through the PEs, including partitioning, computation, and aggregation phases, which visually supports the described computational process.

4.3.3 Strategy 3: Mode-3 fiber distribution

The third strategy for optimizing SpTTV operation on the Cerebras WSE-2 focuses on a mode-3 fiber distribution approach. In this refined method, the tensor $\mathscr{X}$ is partitioned by its M and N dimensions into blocks. Each block corresponds to a fiber of the tensor along mode-3, which is the dimension of the tensor that is not being multiplied by the vector **v**. These blocks are then allocated to the PEs for partial computation. This partitioning strategy ensures that each PE is responsible for a distinct subset of the tensor, facilitating parallel computations across the PEs without overlap. It leverages the WSE-2's inherent parallel processing capabilities and high-bandwidth interconnects, enabling simultaneous computations that contribute to the overall result. Once the computations are complete, the blocks of the resultant matrix are gathered. This collection involves accumulating the partial results from each PE and assembling them into the complete resultant tensor **Y**. By performing this gather operation, the sparse nature of the resultant tensor is preserved, and the COO format's efficiency is maintained. This approach contrasts with previous methods by focusing on reconstructing the entire result matrix in a distributed manner, thereby optimizing the final phase of the SpTTV operation where the computed results are synthesized into a cohesive output. The key to this strategy is the orchestration of data distribution and collection to maintain the benefits of sparsity while also exploiting the full computational power of the WSE-2 architecture. This holistic approach results in a highly efficient and scalable solution for SpTTV operations, paving the way for handling even larger tensors and more complex tensor operations.

Algorithm 3 Mode-3 Fiber Distribution

Input: Sparse tensor $\mathscr{X}$ in COO format using vectors **i**, **j**, **k**, **val**, vector **v**, number of PEs **P**
Output: Resultant tensor **Y** in COO format
Partition $\mathscr{X}$: Divide the M and N dimensions of $\mathscr{X}$ into P blocks.
Initialize Local Y: For each PE, initialize local COO format arrays $\mathbf{Y}_i$, $\mathbf{Y}_j$, $\mathbf{Y}_{val}$.
Determine Maximum Length for $\mathscr{X}$: Identify the maximum length of $\mathscr{X}$ segments across all PEs.
Determine Maximum Length for Y: Identify the maximum length of **Y** segments across all PEs.
for $p = 1$ to P **do**
    Pad $\mathbf{Y}_i$, $\mathbf{Y}_j$, $\mathbf{Y}_{val}$ with zeros to match the maximum length.
    Pad **i**, **j**, **k**, **val** arrays of Tensor $\mathscr{X}$ with zeros to match the maximum length.
end for
Distribute $\mathscr{X}$ **and Y Blocks**: Assign blocks of $\mathscr{X}$ to each PE for partial computation.
Copy v: Copy vector **v** from host to PE1.
Broadcast v: Distribute a copy of **v** to each PE from PE1.
for $p = 1$ **to** P **do**
    Partial $\mathbf{Y}_p$ **Computation**: Perform partial SpTTV computation on PE p using the padded $\mathscr{X}_p$.
end for
Gather Partial Results: Collect partial **Y** segments from each PE.
Construct Final Y: Assemble the gathered segments to form the complete resultant tensor **Y** in COO format.

As detailed in Algorithm 3, we enhance the process by focusing on the equitable distribution of computational loads and data uniformity across the processing elements (PEs). This strategy involves partitioning the sparse tensor $\mathscr{X}$ along its M and N dimensions into manageable blocks, which are then assigned to the PEs. Each PE initializes its segment of the resultant tensor **Y** in COO format and prepares to handle its designated block of $\mathscr{X}$. To ensure consistency and efficiency across all PEs, both $\mathscr{X}$ and **Y** are padded to the maximum segment length identified among the PEs. This padding standardizes the data size of each PE process, facilitating synchronized operations and reducing the complexity of managing variable data lengths.

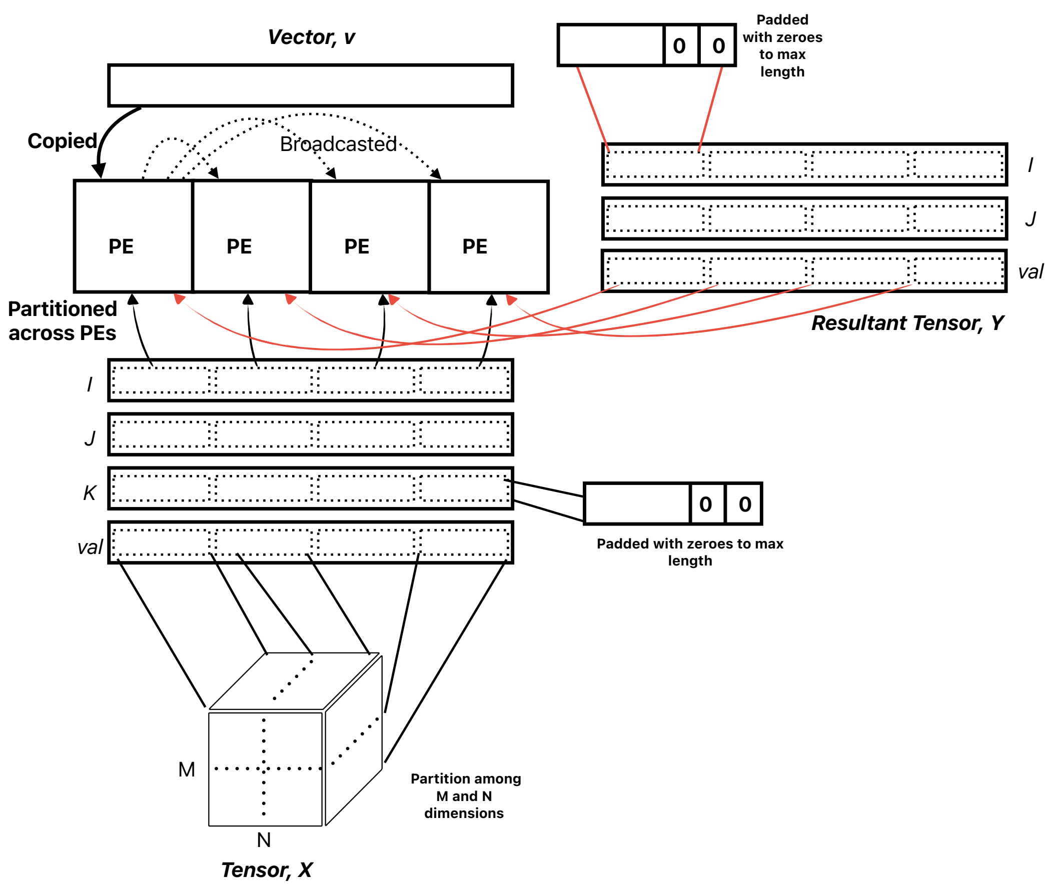


Figure 4.3: An Example of four PEs for Strategy 3.

After distributing $\mathscr{X}$ and copying and broadcasting the vector vv across the PEs, each unit performs its part of the SpTTV computation. The computations are executed on padded data to ensure that each PE operates under similar conditions, maximizing the hardware's computational efficiency. Following the computation, the partial results from each PE are collected and assembled into the complete resultant tensor **Y**. This final gathering step is critical as it consolidates the individual outputs into a unified result while maintaining the sparse tensor format, as visually illustrated in Figure 4.3. This figure effectively depicts the entire process from data partitioning and distribution through to the final assembly of the resultant tensor, highlighting the strategic use of the Cerebras WSE-2's architecture to optimize tensor computations.

CHAPTER

5

RESULTS

The evaluation was performed on a 13th Gen Intel(R) Core(TM) i9-13900F processor running at 2000 MHz clock speed for the PASTA'sLi et al. (2019) OpenMP implementation of SpTTV, and the Cerebras WSE-2 Simulator utilizing the CSL compiler at 850MHz clock frequency for the CSL-SpTTV Implementation. The evaluation covered synthetic tensors generated using pyttbBader et al. (2017) with dimensions Case 1: 50×50×50 with non-zeros ranging from 128 to 32768, number of PEs ranging from 2 to 256, and Case 2: 68×68×68 with non-zeros from 128 to 16384 and number of PEs from 64 to 128. We observed that the largest tensor that could run was 68×68×68 in the simulator.

We evaluated based on the compute cycles of the SpTTV kernel. We notice that the performance of Cerebras-SpTTV increases with increasing the number of PEs as the computation per PE reduces. For case 1, it performs 4× on average faster than the PASTA SpTTV for non-zeroes till 2048 for 256 PEs and it performs comparably for non-zeroes till 8192 but significantly fails as the number of non-zeroes increases as it runs out of memory for the simulation. We observe a similar trend for Case 2 where we achieve 3.5× speedup until 8192 nonzeroes for 128 PEs and later runs out of memory as we increase the number of nonzeroes for the simulation. The figures 5.1 and 5.3 describe the results of the Case 1.

Furthermore, we evaluated the three strategies against each other for tensor 64×64×64

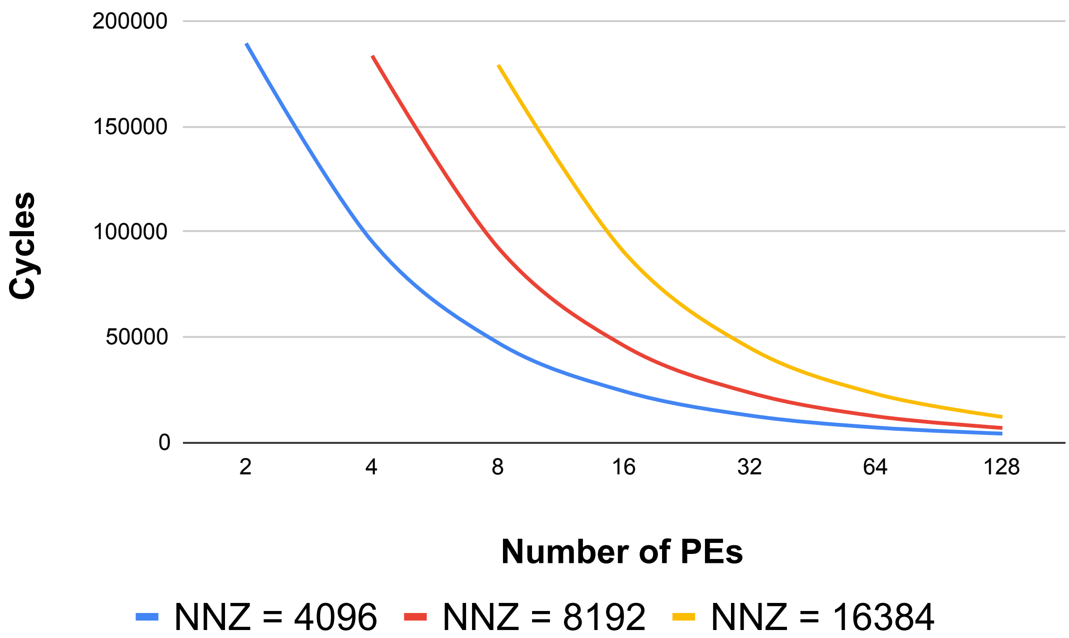


Figure 5.1: Number of PEs VS Compute Cycles for Different NNZ Values for 50×50×50

for 512 nonzeroes as the tensor dimension could not exceed 68, and nonzeros beyond 512 would not run in the simulator for strategy 3. The figure 5.1 describes the results for the above evaluation, where we observe that strategy 1 performs better than the other two strategies that is because the majority of cycles in strategies 2 and 3 come from the memcpy of Tensor $\mathscr{X}$, vector $\mathbf{v}$, and the resultant matrix $\mathbf{Y}$ as we selectively load the data onto the device for these strategies while in a single cycle, data is loaded onto the device in strategy 1. To elaborate, in Strategy 1, the Tensor $\mathscr{X}$ is loaded in a single cycle where the data is contiguous, onto the PEs, and resultant tensor $\mathbf{Y}$ is initialized in the PE. But for strategy 2 and 3, the Tensor $\mathscr{X}$ and Tensor $\mathbf{Y}$ undergo preprocessing on the host and are not contiguous to be loaded onto the PEs in single cycles. This creates more cycles for strategies 2 and 3.

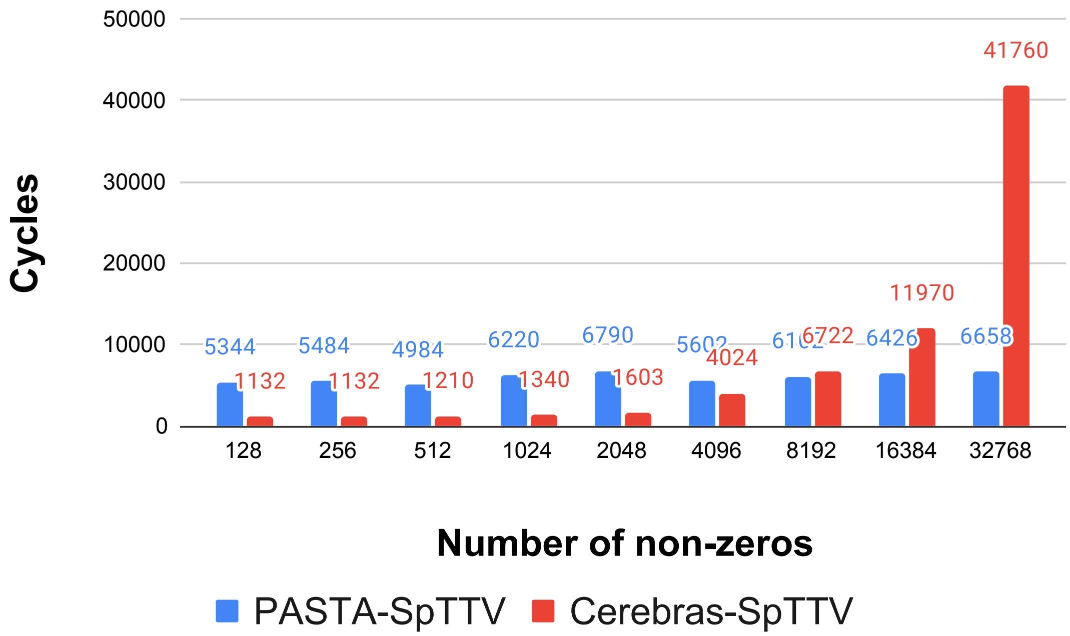


Figure 5.2: PASTA OpenMP-SpTTV Vs Cerebras-SpTTV

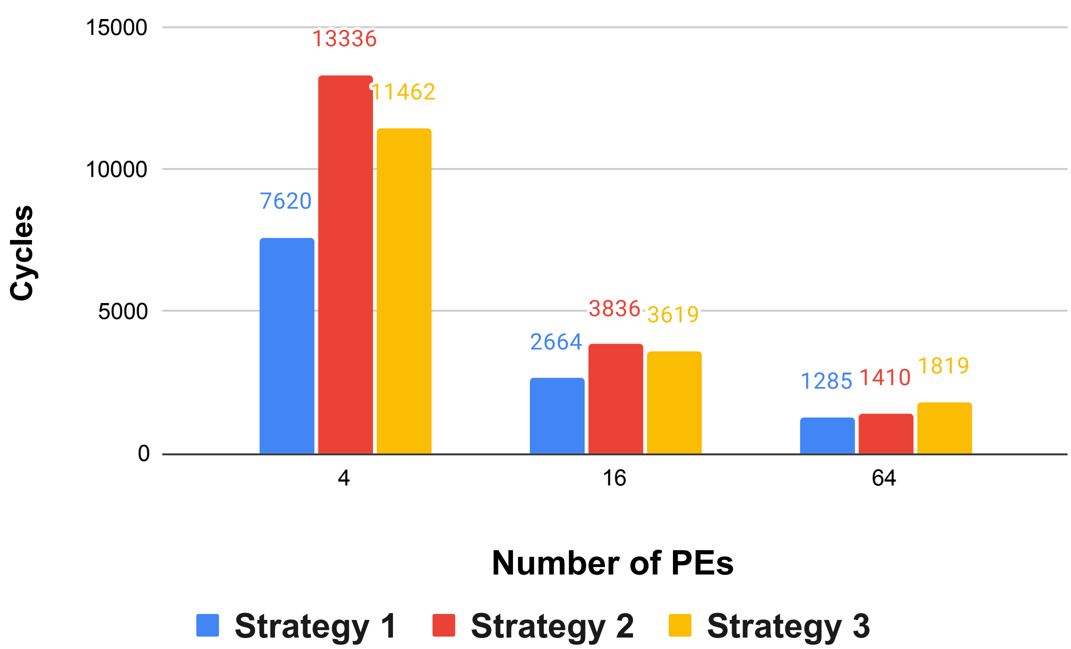


Figure 5.3: Cycles vs Number of PEs for 512 NNZs for the three strategies for 64x64x64

CHAPTER

6

CONCLUSIONS

In this work, we successfully implemented a COO-based SpTTV algorithm on the Cerebras WSE-2 platform, demonstrating its capability to accelerate Sparse Tensor Computations and revealing significant performance improvements. However, we encountered memory limitations during the simulation. Hence, our future work will focus on refining the workload distribution among PEs and enhancing the communication model by delving deeper into the communication primitives.

CHAPTER

7

FUTURE WORK

In our future work, we plan to focus on is improving the efficiency of the memcpy operation, particularly for Strategies 2 and 3 by incorporating techniques such as reordering and combining individual units for a single memcpy onto the fabric. Furthermore, we plan to analyze different communication patterns among the PEs to identify the most efficient communication strategy. Understanding the communication patterns will allow us to optimize data movement, reduce communication overhead, and improve overall system performance. Additionally, we intend to extend the implementation to a 2D topology by utilizing MPI collectives along a 2D mesh. This extension will enable us to leverage the communication capabilities of MPI to facilitate efficient data exchange between PEs arranged in a 2D grid.

Overall, our future work will focus on refining the workload distribution among PEs, optimizing the memcpy operation, enhancing the communication model, and leveraging MPI collectives in a 2D topology. These efforts aim to overcome memory limitations, improve system scalability, and further accelerate Sparse Tensor Computations on the Cerebras WSE-2 platform.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

Bader, B. W., Kolda, T. G., et al. (2017). Matlab tensor toolbox version 3.0-dev. Available online.

Cichocki, A. (2013). Tensor decompositions: A new concept in brain data analysis? *CoRR*, abs/1305.0395.

Ho, J., Ghosh, J., and Sun, J. (2014). Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *KDD 2014 - Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 115–124, United States. Association for Computing Machinery. 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2014 ; Conference date: 24-08-2014 Through 27-08-2014.

Jacquelin, M., Araya-Polo, M., and Meng, J. (2022). Scalable distributed high-order stencil computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’22. IEEE Press.

Kolda, T. G. and Bader, B. W. (2009). Tensor decompositions and applications. *SIAM Review*, 51(3):455–500.

Li, J., Ma, Y., Wu, X., Li, A., and Barker, K. J. (2019). PASTA: A parallel sparse tensor algorithm benchmark suite. *CoRR*, abs/1902.03317.

Lie, S. (2022). Cerebras architecture deep dive: First look inside the hw/sw co-design for deep learning : Cerebras systems. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–34, Los Alamitos, CA, USA. IEEE Computer Society.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. P. (2015). Tensorizing neural networks. *CoRR*, abs/1509.06569.

Papalexakis, E., Faloutsos, C., and Sidiropoulos, N. (2012). Parcube: Sparse parallelizable tensor decompositions. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2012, Proceedings*, number PART 1 in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and

Lecture Notes in Bioinformatics), pages 521–536. Copyright: Copyright 2012 Elsevier B.V., All rights reserved.; 2012 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, ECML-PKDD 2012 ; Conference date: 24-09-2012 Through 28-09-2012.

Rocki, K., Essendelft, D. V., Sharapov, I., Schreiber, R., Morrison, M., Kibardin, V., Portnoy, A., Dietiker, J., Syamlal, M., and James, M. (2020). Fast stencil-code computation on a wafer-scale processor. *CoRR*, abs/2010.03660.

Selig, J. (2022). The cerebras software development kit: A technical overview.

Smith, S. and Karypis, G. (2017). Accelerating the tucker decomposition with compressed sparse tensors. In Rivera, F. F., Pena, T. F., and Cabaleiro, J. C., editors, *Euro-Par 2017: Parallel Processing*, pages 653–668, Cham. Springer International Publishing.

Wang, Y., Chen, R., Ghosh, J., Denny, J. C., Kho, A., Chen, Y., Malin, B. A., and Sun, J. (2015). Rubik: Knowledge guided tensor factorization and completion for health data analytics. *KDD*, 2015:1265–1274.