AMERICAN UNIVERSITY OF BEIRUT

OPTIMIZING SPARSE MATRIX MULTIPLICATION FOR SPARSE DEEP NEURAL NETWORKS ON GPUS

by

BACHIR HICHAM EL MASRY

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
May 2021

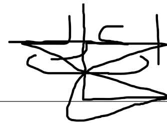
AMERICAN UNIVERSITY OF BEIRUT

OPTIMIZING SPARSE MATRIX MULTIPLICATION FOR SPARSE DEEP NEURAL NETWORKS ON GPUS

by

BACHIR HICHAM EL MASRY

Committee Members:

Dr. Izzat El Hajj, Assistant Professor Computer Science	Advisor 
Dr. George Turkiyyah, Professor Computer Science	Member of Committee
Dr. Shady Elbassuoni, Associate Professor Computer Science	Member of Committee

Date of thesis defense: May 4, 2021

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: El Masry Bachir Hicham
Last First Middle

☑ Master's Thesis ◯ Master's Project ◯ Doctoral Dissertation

☑ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after: **One ___ year from the date of submission of my thesis, dissertation or project.**
Two ___ years from the date of submission of my thesis , dissertation or project.
Three ___ years from the date of submission of my thesis , dissertation or project.

05/10/2021
Signature Date

This form is signed when submitting the thesis, dissertation, or project to the University Libraries

Abstract

Bachir Hicham El Masry for Master of Science
Major: Computer Science

Title: Optimizing sparse matrix multiplication for sparse deep neural networks on GPUs

Deep Neural Networks (DNNs) require a huge amount of computational power and memory storage. Hence, sparsifying the neural network was proposed as a technique to help reduce the computational complexities of DNNs. However, when dealing with parallelization, we face multiple challenges like load balancing, memory management, and many others. Many studies have tackled these problems, some using CPUs, and more recent studies using GPUs. Since modern GPUs, compared to the CPUs, promise a much higher peak floating-point performance and memory bandwidth, we based our study on running DNNs on GPUs. Many works have proven the efficiency of GPUs in dealing with sparse matrices. Our aim is to further explore the effects of applying a combination of various storage formats on the GPU while testing different tiling strategies. We would also be proposing a technique for better memory utilization.

Contents

Abstract **v**

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

2 Background **4**

2.1 GPU Overview 4
2.2 GEMM 5
2.3 Sparse Storage Formats 6
2.3.1 Coordinate list format (COO) 6
2.3.2 Compressed sparse row format (CSR) 7
2.3.3 Compressed sparse column format (CSC) 7
2.3.4 ELLPACK (ELL) 8
2.3.5 Doubly Compressed format (DCSR, DCSC) 8
2.4 Estimating the output size 9
2.5 Partitioning 11
2.6 SpGEMM Algorithms 11
2.6.1 CSR-CSC 12
2.6.2 CSR-CSR 12
2.6.3 CSC-CSC 14
2.6.4 CSC-CSR 14
2.6.5 CSR-ELL 15

3 Related Work **16**

3.1 Sparse Matrix Vector Multiplication 17
3.1.1 Efficient implementation of sparse matrix-sparse vector multiplication for large scale graph analytics [5] 17
3.2 General Sparse Matrix Multiplication on the GPU 18
3.2.1 CUSP [21] 18
3.2.2 cuSPARSE [22] 19

3.2.3 Optimizing Sparse Matrix—Matrix Multiplication for the GPU [6] 19
3.2.4 GPU-accelerated sparse matrix-matrix multiplication by iterative row merging [7] 20
3.2.5 A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors [9] 22
3.2.6 Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures [10] 24
3.2.7 Design Principles for Sparse Matrix Multiplication on the GPU [11] 25
3.3 SpNN: non GPU-based 26
3.3.1 Multithreaded Layer-wise Training of Sparse Deep Neural Networks using Compressed Sparse Column [13] 26
3.3.2 Scalable Inference for Sparse Deep Neural Networks using Kokkos Kernels [14] 27
3.3.3 Write Quick, Run Fast: Sparse Deep Neural Network in 20 Minutes of Development Time via SuiteSparse: GraphBLAS [15] 27
3.3.4 Accelerating Sparse Deep Neural Networks on FPGAs [16] 28
3.4 DNN: GPU-based 29
3.4.1 Performance of Training Sparse Deep Neural Networks on GPUs [17] 29
3.4.2 Accelerating DNN Inference with GraphBLAS and the GPU [18] 29
3.4.3 A GPU Implementation of the Sparse Deep Neural Network Graph Challenge [19] 30
3.4.4 Efficient Inference on GPUs for the Sparse Deep Neural Network Graph Challenge 2020 [20] 31

4 Approach **33**
4.1 General Flow 33
4.2 General Optimizations 34
4.2.1 Batching 35
4.3 Sparse Storage Formats 36
4.3.1 Combinations 36
4.4 Partitioning 37
4.4.1 Doubly-Compressed format 37
4.4.2 2D Tiling 39

5 Methodology **40**

6 Evaluation **44**
6.1 Comparing Storage Formats and Partitioning 44

6.1.1 Storage Formats 45
6.1.2 1D Vs 2D Tiling 46
6.2 Data Sets and Buffer Size 48
6.3 Comparison with other implementations 48
6.3.1 Execution time 48
6.3.2 Memory footprint 50

7 Conclusion **51**

Bibliography **52**

List of Figures

1.1 DNN representation 1
1.2 Every node is connected to all the nodes of the next layer in (a) where only some are connected to the next layer in (b) 2

2.1 COO format 7
2.2 CSR format 7
2.3 CSC format 8
2.4 Two numerical solutions 9
2.5 Two numerical solutions 10
2.6 Different Tiling schemes representations on A, B and C. 12

4.1 Work distribution using CSR 38
4.2 Work distribution using DCSR 38

5.1 SpDNN initial input representation 41
5.2 SpDNN representation 41
5.3 Partial representation of the variation in the number of non-zeros per row of our Input matrix in the 1024 neurons by 120 layers SpNN 43

6.1 Comparing the execution time of our approaches, each along both 1D and 2D partitioning, on our 9 datasets. 44
6.2 Comparing the execution time of the CSR-CSR multiplication with the total execution time. 46
6.3 Comparing the execution time of the CSR-ELL multiplication with the total execution time 46
6.4 Comparing the execution time of the CSC-CSC multiplication with the total execution time 47
6.5 Comparing the execution time of our top approach, the 2D CSR-ELL multiplication, against both Hidayetoglu et al.[20] and Bisson et al. [19] 49

List of Tables

3.1 Table listing all the different approaches done in our Related Work 16
3.2 Table listing general SpM(Sp)V approaches 17
3.3 Table listing different SpMSpM & SpMM approaches 18
3.4 Table listing SpNN approaches on devices other than the GPU (mainly CPU) 26
3.5 Table listing SpNN approaches on GPUs 29

4.1 Table listing previous work and our approaches for a SpNN implementations on GPUs 33

6.1 Experimental results of the total execution time, in seconds, of our approaches. The red values represent our best results. 45
6.2 Experimental results of the GFLOPS of our approaches. The red values represent our best results. 45
6.3 Variation of the execution time depending on the buffer size for the CSR multiplications. 47
6.4 Variation of the execution time depending on the buffer size for the ELL multiplications. 48
6.5 Variation of the execution time depending on the buffer size for the CSC multiplications. 48
6.6 Comparing our top approach with previous implementations . . . 49
6.7 Comparing the memory footprint of the input matrices of previous implementations with our own using a 10% buffer size while disregarding the weights. 50

Chapter 1

Introduction

Deep learning (DL), a subset of Machine Learning (ML), is a technique that achieves Artificial Intelligence (AI) through algorithms trained with data. It has seen great progress over the last decade and has been targeted by a wide community since it can be used in all sorts of applications such as Natural Language Processing (NLP), computer vision, speech-audio recognition and many others. A deep neural network consists of a cascade of interconnected neurons. These neurons, as shown in Figure 1.1, are organized into layers where each neuron from one layer is connected to all the neurons of the next layer. This could also be referred to as fully-connected or dense DNN.

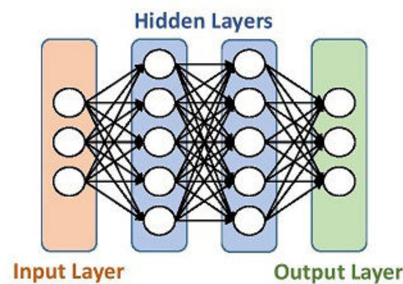


Figure 1.1: DNN representation

Increasing the amount of training data was proven to be an effective approach for improving the model accuracy over a range of machine learning tasks such as object recognition, image classifications and others. However, deep neural networks, given their structure, require a huge amount of data where a noticeable increase in the network size will result in a massive increase in both computational power and memory storage requirements. This has quickly lead the latest DNNs to outgrow the memory limitations of currently available accelerators. In order to address this issue, the DL community attempted to convert the dense

DNNs into sparse DNNs using algorithmic approaches such as pruning.

(a) Fully connected Neural Network (b) Partially connected Neural Network

Figure 1.2: Every node is connected to all the nodes of the next layer in (a) where only some are connected to the next layer in (b)

Sparse neural networks have been proposed as an approach for solving the growing computational complexities of DNN multiplications. Sparsifying the network, as shown in Figure 1.2b, mainly consists of ignoring nodes with negligible or zero weights, given that loss of accuracy can be tolerated/controlled. This will reduce inference computational complexity, increase throughput and will improve the network quality. However, this solution also presents unique scalablity challenges, like load balancing, memory management and data partitioning.

Many studies were conducted in order to tackle these problems. Some techniques involved using custom accelerators (like FPGAs [16]), but we aim to have a more general approach. Other studies utilized the CPUs, mainly using OpenMP, and running on either a single or a multi-CPU system. However, since modern graphics processing units (GPUs), compared to the CPUs, promise a much higher peak floating-point performance and memory bandwidth, the more recent researches started integrating GPUs in their work. Hence, many libraries have been created/tested that have proven their efficiency when dealing with a sparse DNN like GraphBLAS [SuiteSparse [15] , and GraphBLAST [18]] and Kokkos kernels [14].

Most of the prior approaches encountered many problems, some of which were left as future work, which opens the door for optimizations. These problems include memory bottlenecks, choosing the storage formats and problems with load balancing and data partitioning. In this thesis, and in order to further accelerate these models, we aim to test a variety of storage formats for storing and traversing the neural network layers like CSR, CSC and ELLPACK. Depending on the properties of the matrix at each layer, we might choose between storing it as a dense matrix or as a sparse matrix. We also aim to use 2D-tiling techniques along both columns and rows in order to better divide our computations across the available computational units.

Some optimization techniques that we do per iteration will include: buffering, load-balancing, memory management to reduce memory access overhead, filtering the data and pruning the edges. We will be testing different storage formats in both a 1D tiling and 2D tiling implementation in order to validate/highlight the importance of 2D tiling. It should be noted that some strategies could be device specific. Hence, throughout this study, we will be focusing on the best strategies for running a sparse DNN on the GPU. Finally, we'll compare our obtained results and conclude.

In Section 2, we elaborate on the sparse formats and some proposed problems regarding Sparse matrix multiplications. Then, in Section 3, we will be listing some related research conducted on either generally handling a sparse matrix multiplication on the GPUs or studies that has tackled running SpNN on a variety of devices, where some of which utilized the GPU. After that, in Section 4, we will divide our implementations into different steps, where we will be introducing our proposed approaches done at each step. The later Section 5, provides the specifications of the devices and the environments that are used throughout this study. Finally, we will evaluate our approaches in Section 6 and conclude in Section 7.

Chapter 2

Background

2.1 GPU Overview

First, we introduce the architecture of a modern GPU [2]. Unlike the CPU that exhibits a relatively small number of threads, the GPU accommodates tens of thousands of concurrent threads while keeping in mind that the GPU threads have higher latency than the CPU threads. In comparison to CPUs, GPUs are throughput-oriented rather than latency-oriented. This offers a theoretically higher peak floating point performance and memory bandwidth. Modern GPUs are organized into tens of Streaming Multi-processors (SMs). Each SM consists of multiple cores and is capable of executing hundreds of hardware-scheduled threads. Threads are grouped into warps which are in turn are grouped into blocks. At the next level in the hierarchy, blocks are grouped into grids. Grids are launched by a host thread and execute a specialized GPU function known as a kernel. In some cases, grids can also be launched by GPU threads [3], but we do not leverage this feature in our work.

Warps represent the finest granularity of scheduled computational units on each multiprocessor, where the number of threads per warp is defined by the underlying hardware. The GPU's threads can communicate and cooperate with each other within the memory hierarchy depending on their position/groups. This is divided into three layers (ordered from fastest to slowest): "intra-warp", "intra-block" and "intra-grid". In the first layer, the "intra-warp", the threads within a warp can communicate by accessing the adjacent threads registers. For the "intra-block" layer, threads withing a block can cooperate throughout the shared memory. On the last layer, all the threads in the GPU have access to the device global memory.

Execution across a warp of threads follows a data-parallel single instruction, multiple data (SIMD) model. Performance penalties occur when this model is

violated, as happens when threads within a warp follow separate streams of execution — divergence — or when atomic operations are executed in order — serialization. Also, some of the main issues encountered on the GPUs are control divergence, accessing memory in a un-coalesced manner, expensive write to random position in memory making it memory intensive rather than the desired computational expensive, hardware limitations like limited main memory and shared memory sizes, overhead of allocating and re-allocating memory on the GPU since the size of the resulting matrix isn't known before hand.

Within our work, we aim in properly utilizing the memory hierarchy for better performance, and to tackle these issues mentioned above by analysing and proposing different optimizations strategies.

2.2 GEMM

General matrix–matrix multiplication (GEMM) is one of the most essential operations in computational science and modeling. The operation multiplies a matrix A with a matrix B producing a resulting matrix C, where $A \in \mathbb{R}^{m \text{ x } k}$, $B \in \mathbb{R}^{k \text{ x } n}$ and $C \in \mathbb{R}^{m \text{ x } n}$. The matrix product is defined as $c_{i,j} = \sum_{k=0}^{K} a_{i,k} * b_{k,j}$ where $i \in [0, m-1]$ and $j \in [0, n-1]$. Hence, and as shown in Algorithm 1, each output $c_{i,j}$ is the result of the dot product of $a_{i,*}b_{*,j}$, where $a_{i,*}$ denotes the row-vector from A and $b_{*,j}$ denotes the column-vector from B, requiring a time complexity of $O(M * N * K)$.

Algorithm 1 GEMM multiplication

Input: $A \in \mathbb{R}^{m \text{ x } k}$,$B \in \mathbb{R}^{k \text{ x } n}$
Output: $C \in \mathbb{R}^{m \text{ x } n}$

1: **procedure** GEMM(C, A, B)
2:     **for** i in M **do**
3:         **for** j in N **do**
4:             **for** k in K **do**
5:                 $c_{i,j} \mathrel{+}= a_{i,k} * b_{k,j}$
6:     **return**

However, encountering zero-values in either the rows of A or columns of B within the computation have no effect on the resulting matrix C. Hence, wasting the computational and memory resources. This waist overhead is correlated with the sparsity level of the input matrices. A matrix is often called sparse if its number of non-zeroes is relatively small enough compared to its dimensions/storage requirements. Furthermore, an interesting dependency is shown in the dot product where the columns of A are mapped to their corresponding rows

from B. Therefore, building on that dependency, and in order to take advantage of sparsity, a SpGEMM was proposed.

Algorithm 2 SpGEMM multiplication

Input: $A \in \mathbb{R}^{m \text{ x } k}$,$B \in \mathbb{R}^{k \text{ x } n}$
Output: $C \in \mathbb{R}^{m \text{ x } n}$

1: **procedure** SpGEMM(C, A, B)
2:     **for** i in M **do**
3:         **for** k in $a_{i,*}$ **do**
4:             **for** j in $b_{k,*}$ **do**
5:                 $c_{i,j} \mathrel{+}= a_{i,k} * b_{k,j}$
6:     **return**

The SpGEMM, as shown in Algorithm 2, is adjusted to only take into consideration the non-empty values within A and B while computing C. An early description of this algorithm was given by Gustavson [1]. This algorithms iterates over the rows of A and maps the non-empty values column indices to the non-empty values in the corresponding row indices from B which could explain why the CSR format is the most commonly used format. However, this algorithm could be tailored depending on the storage formats used which open the door for many optimizations.

2.3 Sparse Storage Formats

Upon dealing with sparse matrices, storing them as Dense can be very wasteful in terms of the amount of unused allocated memory and computational resources. Hence, many formats were created in order to tackle these issues, from which we will be considering the following: $(D)CSR$, $(D)CSC$ and ELL. We will also be using a form of Doubly-compressed storage formats discussed below. It should be noted that the storage requirements will also differ depending on the type of the values stored [float, double,...]. Our examples will be based on given a sparse matrix A of size MxN.

2.3.1 Coordinate list format (COO)

The Coordinate list format (COO), as shown in Figure 2.1, is composed of 3 arrays: $RowIdxs$, $ColIdxs$ and $Values$ that store the triplets (i, j, v) respectively. These values represent the coordinates and the value of an element from a matrix where i is the row index, j is the column index and v is the value of $a_{i,j}$. Each array has $nnz(A)$ elements, hence require $O(3 * nnz(A))$ total storage space. The

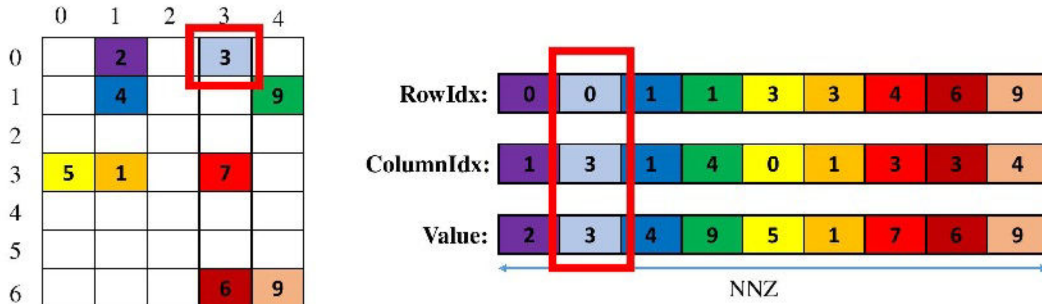


Figure 2.1: COO format

elements are sometimes sorted by row or column. By default, we assume that the elements aren't sorted unless stated otherwise.

2.3.2 Compressed sparse row format (CSR)

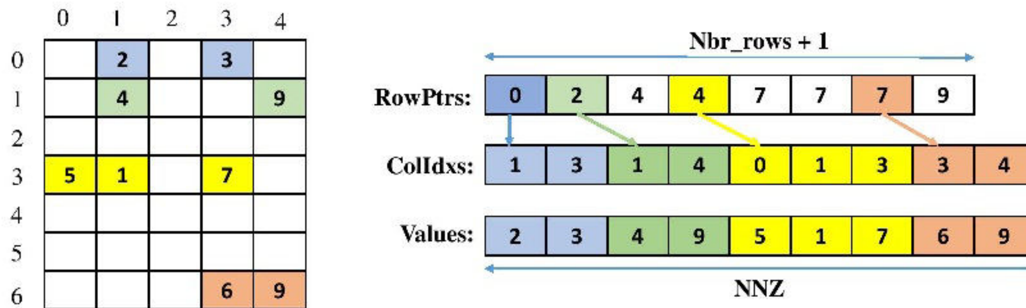


Figure 2.2: CSR format

The compressed sparse row format (CSR), as shown in Figure 2.2, is composed of 3 arrays: $RowPtrs$, $colIdxs$ and $values$. The $RowPtrs$, of size $O(NbrRows+1)$, keeps track of the number of non-zero values at each row where the pairs (col, val) of a_{i*} are stored in an incremental manner at $ColIdxs$ and $Values$ respectively starting at the index $RowPtrs[i]$. Hence, to get the number of nonzero-elements on row r we calculate $nnz(a_{r*}) = RowPtrs[r+1] - RowPtrs[r]$. Therefore, CSR require $O((NbrRows+1)+2nnz(A))$ total storage space. It should be noted that the elements within a row don't have to be sorted by column unless specified otherwise.

2.3.3 Compressed sparse column format (CSC)

The compressed sparse column format (CSC), as shown in Figure 2.3, is composed of 3 arrays: $ColPtrs$, $RowIdxs$ and $Values$. The $ColPtrs$, of size $O(NbrCols+1)$, keeps track of the number of non-zero values at each column where the pairs

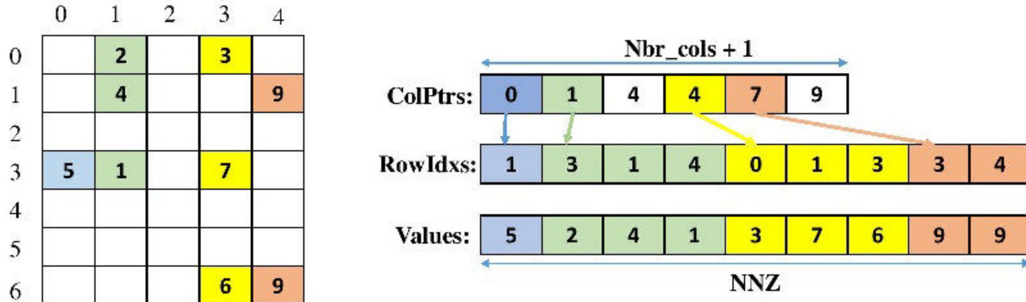


Figure 2.3: CSC format

(row, val) of a_{*j} are stored in an incremental manner at $RowIdxs$ and $Values$ respectively starting at the index $ColPtrs[i]$. Hence, to get the number of nonzero-elements on column c we calculate $nnz(a_{*c}) = ColPtrs[c+1] - ColPtrs[c]$. Therefore, CSC require $O((NbrCols + 1) + 2nnz(A))$ total storage space. It should be noted that the elements within a column don't have to be sorted by row unless specified otherwise.

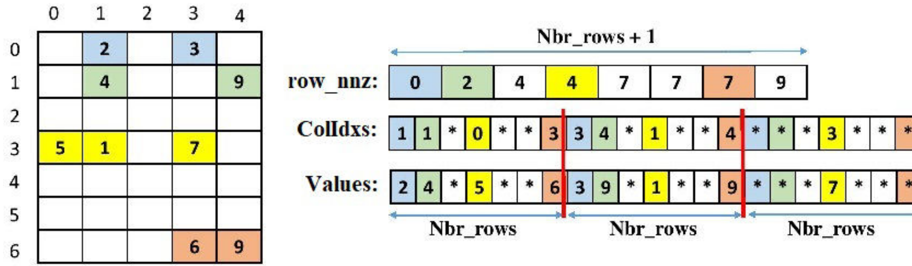
2.3.4 ELLPACK (ELL)

ELLPACK (ELL), as shown in Figure 2.4, is composed of 3 arrays: row_nnz, $ColIdxs$ and $Values$. The row_nnz, of size $O(NbrRows + 1)$, keeps track of the number of non-zero values at each row where the pairs (col, val) of a_{i*} are stored at $ColIdxs$ and $Values$ respectively starting at the index i. However, unlike CSR, the elements within the same row aren't stored successively. Rather, the elements within the same row are stored Nbr_rows apart each other as if they are stored in column major matrix. Hence, ELL require $O(NbrRows * (max(a_i) + 1) + 1)$ storage space. It should be noted that in some cases where some of the rows are (somewhat) dense, ELL could end-up allocating as much space as a Dense matrix.

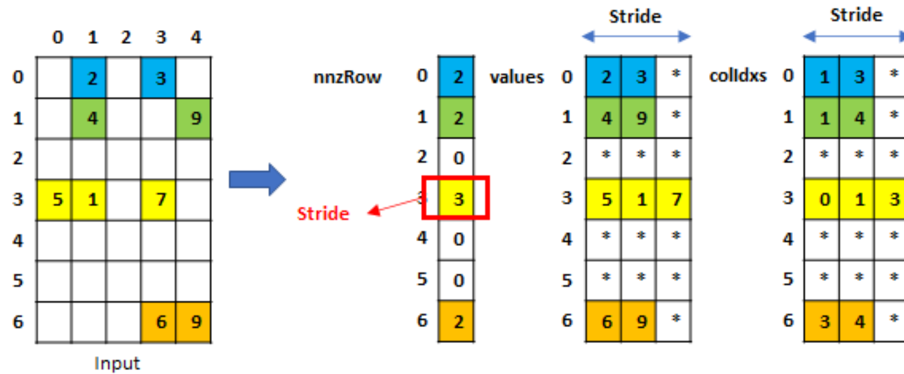
2.3.5 Doubly Compressed format (DCSR, DCSC)

In our implementation, we applied a form of compression to CSR and CSC denoted as DCSR and DCSC by adding a 4th array that keeps track of the non-empty row and column indices respectively.

This doubly compressed format will be used in order to distribute the GPU's computational units across the non-empty (rows | columns) to attain better load balancing. Further details are discussed in Section 4.4.1.



(a) ELLpack 1D representation



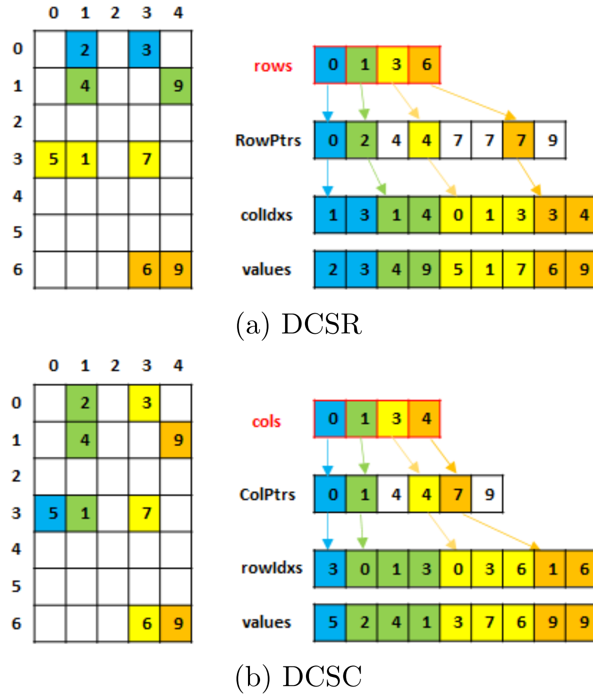
(b) ELLpack 2D representation

Figure 2.4: Ellpack sparse format representation.

2.4 Estimating the output size

In a sparse matrix multiplication, and depending on the characteristics of the input matrices, storing the results as a Dense matrix offers a significant memory overhead that could lead to poor performance and it can also be a main factor in preventing the computations to be done on some devices with limited on-chip memory such as the GPUs. Hence, one of the main issues posed by a SpGEMM is adequately estimating the size of the resulting matrix C. As a solution, the common practices for tackling this issue are listed below, each offering some advantages and disadvantages.

- **Precise method:** pre-computes the sparse matrix multiplication, that is usually done in boolean form instead of actual arithmetic operations, to generate the precise size of C. This method is consider expensive since it dictates the necessity of executing the SpGEMM twice.
- **Probabilistic method:** generates an inaccurate estimation of the $nnz(C)$ based on some probabilistic analysis done on the input matrices. Since this method isn't practical, it was mainly used in order to estimate the execution time of the multiplication.



(a) DCSR

(b) DCSC

Figure 2.5: Representation of both a Doubly compressed CSR and CSC.

- **Upper bound method:** computes the upper bound of $nnz(C)$ such that $nnz(C) = \sum_{i=0}^{m} \sum_{j=0}^{nnz(a_{i*})} nnz(b_{j*})$ where b_{j*} is the corresponding row of $a_{i,j}$. However, this method doesn't take into account zero values generated by the arithmetic operations and almost always allocate excessive amount of memory.

- **Progressive method:** allocates memory of a certain size for C and re-allocates it as the $nnz(C)$ increases. This is the most commonly used method on CPUs but isn't supported on GPUs.

- **Dense method:** allocates the resulting matrix C as a dense matrix. This method eliminates the need to conduct any computations to estimate or re-allocate C. However, this is the most expensive method in terms of memory and is primarily used for focusing on execution time.

These techniques offers a non-negligible overhead. Therefore, we will be splitting our computations into batches and allocate a single buffer. Before flushing the buffer to our output matrix, we will be re-allocating our matrices accordingly. This technique is discussed in details in Section 4.2.1.

2.5 Partitioning

Upon dealing with parallel computing, one of the main focuses is achieving good load-balancing. This is done by equally distributing the work-load across all the available computational units. The partitioning strategy could be considered an essential factor in assessing the overall efficiency of an algorithm. Our work will cover some common sparse matrix computation's partitioning schemes. Given T computational units, the most common partitioning strategies used are:

- **1D partitioning:** the computational units are each assigned to a single row/col per iteration.
- **1D tiling:** the computational units are each assigned to T_{size} consecutive rows/cols per iteration.
- **2D partitioning:** the computational units are each assigned to a single element per iteration.
- **2D tiling:** the computational units are each assigned to a single sub-matrix with T_{row} and T_{col} consecutive rows and columns respectively per iteration.

Note that 2D tiling of sparse matrices has also been applied in the context of graph algorithms such as k-truss decomposition [12].

For further illustration of tiling, assume that we are tackling a sparse matrix multiplication of the form $C_{M,N} = A_{M,K} * B_{K,N}$.

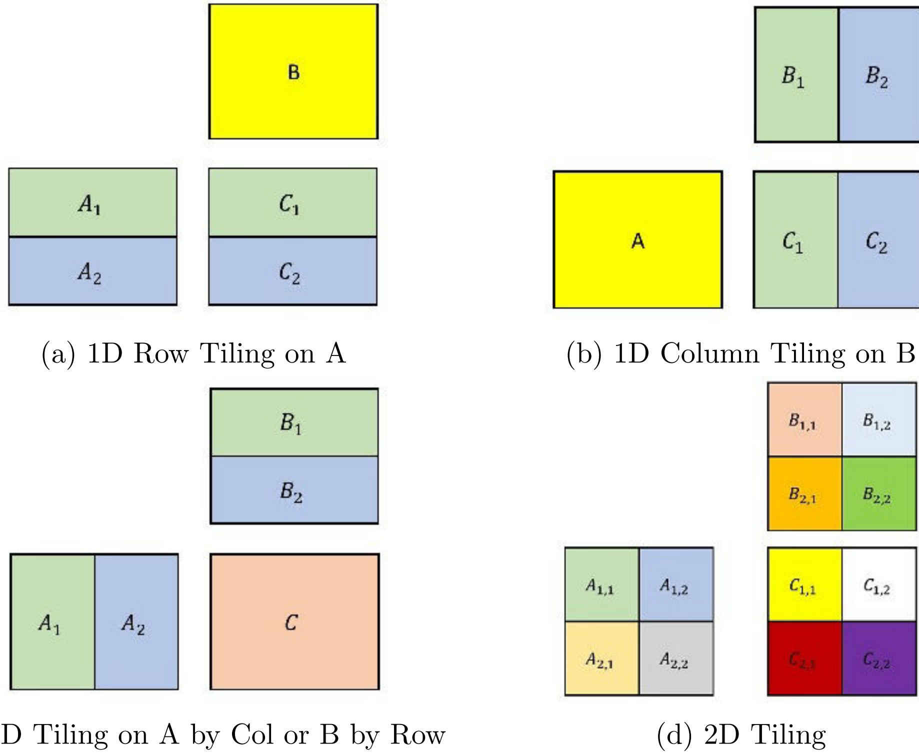
Tiling is necessary to reuse on-chip memory space and improve the processing efficiency. The combination of tiling along multiple dimensions at multiple levels enables high flexibility of the design. Assuming that the matrices are divided into K tiles, the resulting matrix is calculated by:

$$C_{i,j} = \sum_{k=0}^{K} A_{i,k} B_{k,j}$$

The example shown in Figure 2.6 illustrate the partitioning for $K = 2$. It should be noted that for 1D partitioning, either i or j is 0.

2.6 SpGEMM Algorithms

Many storage formats combinations could be utilized upon tackling a SpGEMM multiplication. Each combination offers different advantages and disadvantages that might also depend on the underlying hardware. Throughout this section, we will be listing some of the most commonly used storage format combinations with their corresponding serial pseudo-code implementation. Also, it should be noted that some combinations could dictate certain preferred storage formats for the resulting sparse matrix C.



(a) 1D Row Tiling on A

(b) 1D Column Tiling on B

(c) 1D Tiling on A by Col or B by Row

(d) 2D Tiling

Figure 2.6: Different Tiling schemes representations on A, B and C.

2.6.1 CSR-CSC

As illustrated in the Algorithm 3, a direct conversion of the GEMM Algorithm 1, would store the input matrix A as CSR and the weight matrix B as CSC. This offers some improvements in term of time complexity shifting from $O(M * N * K)$ to $O(M * N * max(nnz(A), nnz(B)))$. However, although this approach is embarrassingly parallel, it still suffers a high overhead since each row needs to iterate over all the columns of B in order to produce a resulting row.

2.6.2 CSR-CSR

The most commonly used algorithm for tackling sparse-sparse matrix multiplication is the CSR-CSR algorithm. The input and weight matrices are both stored in CSR format. As illustrated in Algorithm 4, and following the SpGEMM mentioned in Algorithm 2, this implementation will iterate over the rows of A and maps the column indices to their corresponding row indices from B. Each non-empty value $a_{i,k}$ will iterate over the elements of $b_{k,*}$ to compute the results of $c_{i,*}$. Hence, it is preferred to store the output matrix C as CSR. This implementation requires a single access to the rows of A and multiple access to the rows of B.

Algorithm 3 CSR-CSC multiplication

Input: A_{mxk} in CSR, B_{kxn} in CSC
Output: C_{mxn}

```
1:  procedure SpGEMM(C, A, B)
2:      for r in A.rowPtrs do
3:          for c in B.colPtrs do
4:              a ← A.rowPtrs[r]
5:              b ← B.colPtrs[c]
6:              while a < A.rowPtrs[r + 1] & b < B.colPtrs[c + 1] do
7:                  ka ← A.colIdxs[a]
8:                  kb ← B.rowIdxs[b]
9:                  if ka < kb then
10:                     a + +
11:                 else if ka > kb then
12:                     b + +
13:                 else
14:                     C[r][c] += A.values[a]*B.values[b]
15:                 a + +, b + +
16:     return
```

Algorithm 4 CSR-CSR multiplication

Input: A_{mxk} in CSR, B_{kxn} in CSR
Output: C_{mxn}

```
1:  procedure SpGEMM(C, A, B)
2:      for r in A.rowPtrs do
3:          nnza ← A.rowPtrs[r + 1]−A.rowPtrs[r]
4:          for i in nnza do
5:              idxA ← A.rowPtrs[r] + i
6:              k ← A.colIdxs[idxA]
7:              va ← A.values[idxA]
8:              nnzb ← B.rowPtrs[k + 1]−B.rowPtrs[k]
9:              for j in nnzb do
10:                 idxB ← B.rowPtrs[k] + j
11:                 c ← B.colIdxs[idxB]
12:                 vb ← B.values[idxB]
13:                 C[r][c] += va ∗ vb
14:     return
```

2.6.3 CSC-CSC

Similarly to the CSR-CSR multiplication, the CSC-CSC multiplication is considered to be the transpose of the later. The main difference here, is that instead of iterating over the rows of A, we are iterating over the columns of B and mapping them to their corresponding columns of A. As shown in Algorithm 5, each non-empty value $b_{k,j}$ will iterate over the elements of $a_{*,k}$ to compute the results of $c_{*,j}$. Hence, it is preferred to store the output matrix C as CSC. This implementation requires a single access to the columns of B and multiple access to the columns of A.

Algorithm 5 CSC-CSC multiplication

Input: A_{mxk} in CSC, B_{kxn} in CSC
Output: C_{mxn}

```
1:  procedure SpGEMM(C, A, B)
2:      for c in B.colPtrs do
3:          nnzb ← B.colPtrs[c + 1]−B.colPtrs[c]
4:          for i in nnzb do
5:              idxB ← B.colPtrs[c] + i
6:              k ← B.rowIdxs[idxB]
7:              vb ← B.values[idxB]
8:              nnza ← A.colPtrs[k + 1]−A.colPtrs[k]
9:              for j in nnza do
10:                 idxA ← A.colPtrs[k] + j
11:                 r ← A.rowIdxs[idxA]
12:                 va ← A.values[idxA]
13:                 C[r][c] += va ∗ vb
14:     return
```

2.6.4 CSC-CSR

Inspired by the dependency between the columns of A and the rows of B, this approach, illustrated in Algorithm 6, stores the input matrix A in CSC format and the weight matrix B in CSR format. The main advantage offered by this implementation is that it require a single access to the elements in both A and B. However, the main disadvantage is the random write-memory accesses to C which could drastically hinder performance if not dealt with properly, especially on a GPU.

Algorithm 6 CSC-CSR multiplication

Input: A_{mxk} in CSC, B_{kxn} in CSR
Output: C_{mxn}

```
 1: procedure SpGEMM(C, A, B)
 2:     for k in A.colPtrs do
 3:         nnza ← A.colPtrs[k + 1]−A.colPtrs[k]
 4:         for i in nnza do
 5:             idxA ← A.colPtrs[k] + i
 6:             r ← A.rowIdxs[idxA]
 7:             va ← A.values[idxA]
 8:             nnzb ← B.rowPtrs[k + 1]−B.rowPtrs[k]
 9:             for j in nnzb do
10:                 idxB ← B.rowPtrs[k] + j
11:                 c ← B.colIdxs[idxB]
12:                 vb ← B.values[idxB]
13:                 C[r][c] += va ∗ vb
14:     return
```

2.6.5 CSR-ELL

As an optimization of the CSR-CSR approach, this technique, illustrated in 7, is used to ensure coalesced and (semi-)coalesced memory access pattern to A and B respectively.

Algorithm 7 CSR-ELL multiplication

Input: A_{mxk} in CSR, B_{kxn} in ELL
Output: C_{mxn}

```
 1: procedure SpGEMM(C, A, B)
 2:     for r in A.numRows do
 3:         nnza ← A.rowPtrs[r + 1]−A.rowPtrs[r]
 4:         for i in nnza do
 5:             idxA ← A.rowPtrs[r] + i
 6:             k ← A.colIdxs[idxA]
 7:             va ← A.values[idxA]
 8:             for s in B.row_nnz[k] do
 9:                 idxB ← (B.numRows ∗ s)+j
10:                 c ← B.colIdxs[idxB]
11:                 vb ← B.values[idxB]
12:                 C[r][c] += va ∗ vb
13:     return
```

Chapter 3

Related Work

In this thesis, we aim to build an optimized GPU based implementation that solves sparse deep neural networks. This challenge consists of optimizing the sparse matrix-matrix multiplications performance on the underlying architecture. Hence, we will be analyzing previous implementations that handles running general sparse matrix-matrix multiplication on the GPU. Also, we will be viewing some of the previous work that tackled running a sparse neural network on different devices (CPUs, GPUs and accelerators). We summarize these works in Section 3.1, then cover them throughout the rest of the chapter.

Device	Appl	Sparse matrix format				Partition	Ref
		Input	Weight	Buffer(s)	Output		
CPU	SpMSpV	CSC	bitVec	Buk(xThds)	Vec	1D col,row	[5]
	SpNN	CSC	CSC	-	CSC	1D Tiling	[13]
		CSR	CSR	-	CSR	1D row	[14]
		CSR	CSR	-	CSR	1D row	[15]
FPGA	SpNN	Dense	CSC	-	Dense	1D Tiling (x3)	[16]
GPU	SpMSpM	CSR	CSR	COO_s	CSR	1D row	[6]
		CSR	CSR	CSR(x2)	CSR	1D row	[7]
		CSR	CSR	CSR	CSR	1D row	[9]
		CSR	CSR	-	CSR	1D row	[10]
	SpMM	CSR	Dense	COO	Dense	1D Tiling	[11]
	SpNN	CSR	CSR	CSR	CSR	1D row	[17]
		CSR	CSR	CSR	CSR	1D row	[18]
		Dense	CSR	-	Dense	1D tiling (x2)	[19]
		Dense	CSR	-	Dense	2D partition	[20]
		Dense	ELL	-	Dense		

Table 3.1: Table listing all the different approaches done in our Related Work

This chapter is divided into four sections. Section 3.1 serves as a general introduction to sparse matrix vector multiplications. Section 3.2 will cover some prior research and libraries that deal with running a general sparse matrix multiplication on the GPU. This will give us a feel of the common issues faced with sparse matrices and the common strategies used to deal with them. Section 3.3 and Section 3.4 will cover prior works that have tackled running a SpNN on both the CPU and GPU respectively. The final Section, in order to assess our work, we will be using some of the mentioned/created libraries as base cases for comparison.

3.1 Sparse Matrix Vector Multiplication

Before diving in the main task of solving sparse matrix multiplications, we start with an overview of a general sparse matrix-vector multiplications. Our aim is highlighting some correlations between both methods that could be useful in solving our problem efficiently.

Algorithm	Sparse matrix format				Partition
	Input	Weight	Buffer(s)	Output	
J. Serrano [5]	CSC	bitVec	Buckets(xThreads)	Vec	1D col,row

Table 3.2: Table listing general SpM(Sp)V approaches

3.1.1 Efficient implementation of sparse matrix-sparse vector multiplication for large scale graph analytics [5]

This work covers a general optimized SpMSpV implementation on the CPU. Throughout their work, they address memory access efficiency, partitioning efficiency, and synchronization, with the goal of exploiting the maximum memory bandwidth that a system provides. It follows a 1D column-wise partitioning scheme followed by a 1D row-wise tiling scheme discussed below. The input sparse matrix A is stored as CSC while the sparse weight vector B is stored as bitVector producing a Dense vector C. This algorithm also provided a SpMV approach and employ some heuristic, that are based on the number of nonzeros involved in the operation, in order to decide between applying a SpMSpV or SpMV multiplication.

The proposed algorithm follows a two-phased approach that requires private and global buckets/bins. Before launching the algorithm, the data is preprocessed in order to determine the upper bound number of nonzeros involved. Based on that and some given heuristics, it chooses between 2 partitioning strategies across sockets (NUMA aware), decides whether to run a SpMSpV or SpMV

algorithm, and allocates the needed sizes for the global bins in the global bucket. Each thread gets assigned a private fixed size bucket partitioned into row-wise bins. The first phase, the scaling phase, distributes the threads across the nonzeros of B where they compute and store the values in the private bins and copy them to the corresponding global bins when they are either full or when the first phase terminates. The second phase, the aggregation phase, re-distributes the threads across the global bins to aggregate and to store the values in the resulting vector C in a synchronous manner.

From their approach, they presented an interesting notion of choosing between sparse or dense formats given some heuristics. However, their methods are CPU specific that focus on the CPU's memory hierarchy. Hence, they cannot be implemented efficiently on GPUs.

3.2 General Sparse Matrix Multiplication on the GPU

This section covers some work that tackled running a sparse matrix-matrix multiplication on the GPU. In most of these papers, a wide variety of matrices were used for testing, each with different shapes/sizes and sparsity structures. This section allows us to have a better overview of the GPU's architecture, from threads to warps to blocks. It also sheds the light on different key features/issues that affects performance that should be taken into consideration such as: memory hierarchy, memory management, data management and load-balancing.

Algorithm	Application	Sparse matrix format				Partition
		Input	Weight	Buffer(s)	Output	
ESC_{op} [6]	SpMSpM	CSR	CSR	COO_s	CSR	1D row
RMerge [7]	SpMSpM	CSR	CSR	CSR(x2)	CSR	1D row
*bh*SPARSE [9]	SpMSpM	CSR	CSR	CSR	CSR	1D row
*kk*SpGEMM [10]	SpMSpM	CSR	CSR	-	CSR	1D row
Yang et al. [11]	SpMM	CSR	Dense	COO	Dense	1D Tiling

Table 3.3: Table listing different SpMSpM & SpMM approaches

3.2.1 CUSP [21]

Cusp is a library for sparse linear algebra and graph computations based on Thrust. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. In comparison to the other methods,

Cusp achieves very constant performance when dealing with sparse matrix multiplications and isn't affected by their irregular sparsity structures. This can be explained by the ESC (Expansion, Sorting, Compression) approach. However, it pre-allocates an intermediate buffer with a size equal to the number of multiplications needed. The size is computed by using the "upper bound method". The array is then sorted in O(n) using radix sort which dominates the computational cost. Since the algorithm doesn't ignore the zero entries generated by arithmetic operations, the resulting matrix is normally larger than the input matrices. Hence, in some cases the intermediate matrix $\widehat{C}$ maybe too large to fit in the device global memory. Further explanation is done in the work discussed below [6] since it follows the same approach with some optimizations.

3.2.2 cuSPARSE [22]

cuSPARSE is a CUDA library used for handling sparse matrices. It pre-computes a simplified SpGEMM [boolean matrices] in the same computational pattern in order to allocate sufficient memory for the output matrix C before the actual computation takes place. The matrices could be stored in a bunch of supported formats following a 1D scheme done on the GPU. Even though this method generates precise size to the nnz(C) it is expensive. It also utilizes the GPU hash-table for the insert operations. So time complexity of this approach is O(flops) on average and O(flops nnzr(C)) in the worst case, where nnzr(C) is defined as the average number of nonzero entries in the rows of the matrix C. Because the algorithm allocates one hash table of fixed size for each row of C, the space complexity is O(nnz(A)+nnz(B)+n+nnz(C)).

3.2.3 Optimizing Sparse Matrix—Matrix Multiplication for the GPU [6]

This work covers a fully general optimized SpGEMM implementation on the GPU. Their approach is based on the CUSP library with some improvements by introducing some bandwidth saving operations. It follows a 1D row-wise partitioning scheme on a single GPU architecture while handling double precision values. The matrices are stored in CSR format and a sorted COO format was also used as an intermediate buffer in the computation. The approach used is an optimization of the expand-sort-construct (ESC) algorithm that consist of 3 main phases: expansion, sorting and contraction. In general, the ESC algorithm distills the computation into a small set of data-parallel primitives whose performance characteristics are readily understood in order to eliminate any complexity or unpredictable data-access patterns.

A high level overview of this algorithm follows 5 consecutive steps: "slice, expand, sort, contract, construct". Initially, "slice" will partition the input CSR

matrix A into M sub-matrices A_k in a 1D row-wise where M is the number of rows in A [each A_k represent a single row from A]. It will then iterate over the M sub-matrices while applying the 3 intermediate steps. Each sub-matrix will be called in "expand" where the generated product between A_k and B will be stored in a Buffer C_k in COO format that was allocated using upper bound method. The values in C_k will be sorted in the "sort" call by row indices and then by column indices. Next, the "contract" will eliminate any duplicates by accumulating their values. Finally, after iterating, "construct" will merge all the C_k producing the output matrix C.

In their optimization, the paper highlighted the importance of utilizing shared memory in order to reduce data-movement operations in global memory. Following the memory hierarchy, each of the 3 intermediate steps were adjusted to better utilise shared memory and register when possible. Due to the extreme variation of the workload requirement per row, assigning a static computational unit per row could lead to extreme deviation in load-balancing and poor performance. Hence, as a solution, and since dynamic assignment of computational units per row is difficult to implement, they proposed an interesting notion of reordering the matrix rows by the amount of computational work. This permutation strategy allows the grouping of rows given their workload and memory requirement. These permutations were used to distribute the work across threads, Warps and Blocks, depending on the number of non-zero elements per row, offering some improvements in load balancing.

It should be noted that this work takes into consideration big row sizes that don't fit into the GPU's shared memory. Upon dealing with this issue, the computation of the rows that don't fit into shared memory was done in the main memory. Also, it emphasised on reducing the cost of the sorting phase. They introduce an interesting notion of assigning computational units in a seemingly dynamical fashion rather than statically that has proved to be an efficient approach for reducing load balancing. Hence, we decide to adopt a similar version of this technique in our implementation. However, instead of deciding what memory to use depending on the nnz per row we wish to reduce the use of the GPU's main memory in our computation as much as possible.

3.2.4 GPU-accelerated sparse matrix-matrix multiplication by iterative row merging [7]

This work presents a GPU-accelerated method for general sparse matrix-matrix multiplication (SpGEMM). It follows a 1D row-wise partitioning scheme on a single GPU architecture, where each row is assigned to a sub-warp of threads. The matrices are stored in CSR format, with the assumption that the column indices

are sorted, while handling double precision values. Similar to "Merge Sort", the method used, denoted by "RMerge", computes and merges the rows on the GPU. However, the elements with duplicate column indices are aggregated on the fly. This early merger will reduce global memory access and is referred to in this work as the "compression effect". The compression factor, defined as the ratio of the number of nonzero multiplications to nnz(result), was later calculated and shown to have an interesting correlation with the performance rate.

The proposed algorithm "RMerge" require that the input matrix A has atmost W non-zeros per row before performing the sparse matrix multiplication. In order to satisfy this condition, the input matrix A is split into K matrices $A = G_k G_{k-1} ... G_1$ such that each matrix G_i satisfy the condition. Each split will reduce the maximum row length by a factor of W, hence, $K = \left\lceil \frac{max(nnz(a))}{W} \right\rceil$ splits are needed where a is a row from A. This split will transform our original $C = AB$ into a chain of sparse matrix multiplications $C = G_k G_{k-1} ... G_1 B$. However, in order to reduce memory consumption, the splitting was done iteratively K times. At each iteration i, the current input matrix A_i will be split into $A_i = A_{i+1} G_i$ where A_{i+1} doesn't satisfy the condition and can be further split. The computation is done from right to left resulting in an intermediate result $B_{i+1} = G_i B_i$ per iteration. Therefore at most 2 intermediate buffers are needed. Only the first merging level require multiplication. Furthermore, the values of $G2, ..., Gk$ are guaranteed to be one which is used to reduce memory consumption and to avoid trivial multiplications.

The GPU-accelerated function MulLimited() is called for computing the product $C = AB$. This function is divided into 3 steps. It starts with computing the structure of the result matrix C by calculating the rows lengths. Then, in order to have an estimate of the required memory, the rowPtrs will be computed using prefix-sum before allocating C. Finally, the values and column indices of C will be computed and stored. It should be noted that the first and third step are similar. As discussed above, A will consist of a limited number of non-zeros per row W, where W represent the sub-warp size, which is passed as argument to the function. The sub-warp size template of W could be either [2, 4, 8, 16, 32]. Each sub-warp will be responsible for a single output row following $c = aB$ where a and c are corresponding rows of A and C. Each thread withing the sub-warp will cover a single element $a_{r,j}$ that correspond to the row $b_{j,*}$ from B. Hence, this approach allows the merger of up to W rows of the right-hand side B that were selected and weighted by a.

This work focuses on increasing the number of arithmetic operations than that of the memory transfer from and to global memory. Although they presented a good use of computational resources with good caching and high occupency

was achieved, their approach presents many restrictions and could be considered somewhat wasteful. One of the main drawbacks is the high logarithmic dependency on the sparsity structure of the input matrix A where the value of K, number of splitting and memory overhead were all affected by the characteristics of A. Also, throughout their work, the sub-warp size effected performance and needed to be tailored. As an optimization, in the last multiplication, the lowest sufficient sub-warp size was used. The work done only focused on registers and didn't use shared memory to fully benefit form the GPU's memory architecture. When dealing with irregular input matrices, load imbalance and large memory allocation make it inefficient. In our implementation, we will be implementing a somewhat similar approach in terms of sub-warps. However, we will be avoiding splitting and will be utilizing shared memory. Our aim is having a dynamically assigned thread distribution strategy that scales well from assigning sub-warps to assigning full Blocks.

3.2.5 A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors [9]

This work is an extended version of the paper "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data [8]". Their algorithm, denoted by "bhSPARSE", mainly focuses on improving the GPU SpGEMM performance for matrices with arbitrary irregular sparsity structures. It follows a 1D row-wise partitioning scheme where a non empty row from A is either assigned a single thread or a Block of threads depending on the upper bound number of nonzeros in the corresponding resulting row in C. The matrices are stored in CSR format, with the assumption of sorted column indices, and can be in single or double precision. The proposed framework is implemented on either a single GPU or a CPU–GPU heterogeneous processor architecture. The later is composed of CPU cores, GPU cores and shared virtual memory and promotes the use of reallocatable shared virtual memory. The SpGEMM posed three main problems: (1) not knowing the size of the resulting matrix C in advance, (2) parallel insert operations at random positions in C hinder performance and (3) Load balancing when dealing with sparse data. These problems were handled in a unique manner and are described below.

Their SpGEMM approach is implemented in four successive stages: "upper bound", "binning", "computing" and "arranging memory". The first stage, generates the upper bound number of nonzero entries in each row of the resulting matrix C and stores them in an array U [1 GPU thread per row]. The second stage, executed on the CPU, is responsible for allocating an intermediate matrix $\widehat{C}$ using a hybrid allocation method of both progressive and upper bound

method. It allocates 38 bins and put them in five groups: $G_1 : [bin_0]$, $G_2 : [bin_1]$, $G_3 : [bin_2, ..., bin_{32}]$, $G_4 : [bin_{33}, ..., bin_{36}]$ and $G_5 : [bin_{37}]$. Each row is assigned to a corresponding bin according to its number of nonzero entries $nnz(u_r)$. The size for the intermediate matrix $\widehat{C}$ is computed as: $\sum_{n=0}^{36} nnz(bin_i) * nnz(u_r) + \sum nnz(bin_{37}) * 256$ where u_r represent the row indices belonging to bin_i. Each group has a different thread distribution strategy and computation method that eliminates any duplicates. It should be noted that every bin, except for bin_0 and bin_1, has its own kernel depending on its own characteristics and group.

The third stage consists of computing and storing the results in $\widehat{C}$, and updating the number of nonzeros of the resulting matrix C. In $G3$, a single thread was assigned per row and it uses a heap method. A single heap is created for every row in this group and is located in the shared memory. In $G4$, a thread block was assigned per row and it uses a bitonic ESC method. This method collects the nonzeros entries in shared memory, sorts them using bitonic sort and compresses the duplicate indices using prefix-sum scan. In both $G3$ and $G4$, after the computation, the values were copied form shared memory to main memory. However, in $G5$, since the expected number of nonzeros per row exceeded the on-chip memory capacity, the computation was done by utilizing both the main memory and the shared memory. The last group, $G5$, assigned a thread block per row and used merge path algorithm. The proposed merge method is originally done on shared memory and is split into four sub-steps: (1) binary search, (2) prefix-sum, (3) copying and (4)merging. When the row exceeds the allocated shared memory size, the kernel records the current position, dumps the values to global memory, re-allocates global memory size [x2 each time] and re-lunch with 2x shared memory. In the event that a row is too large to fit in shared memory, the merger is done on 2 level: the shared memory and then on global memory.

The fourth and final stage sums the number of nonzeros from $\widehat{C}$ and allocates the resulting matrix C. It then copies the data to C, while ignoring $G1$, using a single thread per row for $G2$, and using a thread block per row for the remaining groups $G3$,$G4$ and $G5$. This work's approach outperforms other known CPU and GPU SpGEMM methods [including RMerge]. However, it is better suited for heterogeneous CPU-GPU processors, especially when re-allocation is needed. Their algorithm is build on the assumption of sorted sequences and present some control divergence for $G4$ and $G5$. In our implementation, we divided the matrices into 2D tiles, where each tile is guaranteed to fit in the GPU. Hence, pre-allocating the resulting sparse matrix step could be completely ignored. We might consider some similar tactics, while taking into account the event of non-sorted sequences, with some of their merger methods depending on the number of nonzeros per row. However, our computational thread distribution strategy doesn't require many kernels and kernel lunches.

3.2.6 Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures [10]

This work focuses on building an efficient muti-platform/portable SpGEMM implementation. They aim on minimizing the need for revisiting algorithmic designs upon shifting between different high performance computing architectures. Their work builds on top of Kokkos [23], a C++ library providing an abstract data and task parallel programming model. In their implementation, they developed many algorithms for sparse matrix multiplication, each using different data structures for the accumulator. They created a meta-algorithm denoted by "kkSpGEMM" that, based on some heuristics, chooses the optimal algorithm between them depending on the problem being tackled. It follows a 1-D row-wise partitioning scheme where each row is assigned a computational unit. The matrices are stored in CSR format, and can be in single or double precision. The proposed framework is implemented on either a single CPU, KNL or GPU architecture.

Their core implementation uses a two-phase approach with two thread-scalable data structures: a memory pool and an accumulator. The first phase, denoted as "symbolic phase", finds the structure of C by using an optimized "precise method" where the weight matrix B can be compressed. The second phase, denoted as "numeric phase", computes and stores the values in the resulting matrix C. It should be noted that the following terminologies, referenced in this work, that we will be using: "Team, Thread, Vector" refers to "thread block, warp, thread" in GPUs and "hyperthreads, thread, thread vectorization" in CPU/KNL respectively. Both the Thread-Sequential and the Thread-Flat-Parallel scheme were used on GPUs and the Thread-Sequential scheme was used on CPUs/KNLs. The Thread-Sequential scheme follows a Thread per row approach that traverses the corresponding rows from B sequentially while exploiting vector parallelism. On the other hand, the Thread-Flat-Parallel scheme follows a Team per row approach that flattens the needed multiplications across the Threads in a Team while exploiting vector parallelism.

Throughout their work, three different algorithms were proposed: KKMEM, KKDENSE and KKLP. The kkSpGEMM meta-algorithm chooses between either "KKMEM" or "KKLP" or between either "KKMEM" or "KKDENSE" when running on the GPU or CPU/KNL respectively. The first algorithm, "KKMEM", uses a Thread-Sequential approach with a two-level Linked-list based HashMap accumulators (LL). It aims at minimizing the memory use and localizing memory accesses at the cost of increased hash operations/collisions. The second algorithm, "KKDENSE", uses a Thread-Sequential approach with dense accumulators. Hence, it eliminates the overhead of hashing but was limited to CPUs and KNLs only. The last algorithm, "KKLP", uses the Thread-Flat-Parallel partitioning with two-level Linear probing HashMap accumulators (LP) and is limited

to the GPUs. The two-level hash-maps used dictates the necessity of memory pools in order to reduce the allocation and de-allocation overhead. Also, as an optimization, a compression technique was proposed for the weight matrix B in order to reduce the number of times it was accessed.

Throughout this work, cuda-streams weren't used since they still aren't supported in Kokkos. They listed different computational units distribution strategies while presenting different accumulator data-structures and highlighted the effects of each choice. However, running a two-phase approach with a "precise method" could be very costly and will most probably outweigh the benefits despite performing some optimization. In our implementation, and since we are allocating dense Buffers, we no longer have to worry about estimating C and can completely ignore this step. We might also consider using pools in an effort of reducing the number of re-allocation needed throughout the computation. Finally, we might consider editing our dense accumulator to better utilize the limited shared memory on the GPUs by using hashMaps and compression.

3.2.7 Design Principles for Sparse Matrix Multiplication on the GPU [11]

Unlike the previous work, this work tackles a sparse-dense matrix multiplications (SpMM) on the GPU. Their approach was the result of the combination between a key memory access pattern and recent advances in the SpMV. The input sparse matrix A is stored in CSR and multiplied with a dense matrix B producing a dense resulting matrix C. Their implementation is composed of 2 separate SpMM algorithms: "Row-splitting" and "Merge-based". Both algorithms follow a 1D tiling scheme. Their design focused on "latency hiding" with thread- and instruction- level parallelism (TLP and ILP), and efficient load-balancing. The thread-level parallelism (TLP) was done by lunching a sufficient number of warps to achieve high occupancy while the instruction-level parallelism focused on "threat coarsening".

The first algorithm, Row-splitting SpMM, assigns a warp per row. The warp iterates over the nonzeros of its row in batches of size 32 (WARPSIZE). In order to access B in a coalesced manner, each thread will be responsible for loading a column from B. The key component is warp broadcasts ("shuffle" warp intrinsic –shfl) where each thread broadcasts the row index to read from B corresponding to the column index read from A. The values are computed in registers before written to C in a coalesced manner. On the other hand, the second algorithm, Merge-based SpMM, evenly distributes the work across the blocks by equally distributing the nonzeros across the threads. This require flattening the CSR to COO. However, due its heavy register usage it can't benefit from ILP. Also,

it presents 2 memory access overhead when compared to Row-splitting: 1) the additional kernel call for calculating the starting index of each block and 2) CTAs writing to the same rows in C (boundaries).

Depending on the average nonzeros per row withing the matrix, and given some heuristics, the most suitable algorithm from these 2 is chosen. However, in their assumption, B was considered to be a 'tall-skinny' dense matrix compared to the sparse input matrix A. This work provided a very interesting approach to utilizing registers to better benefit from the GPUs memory hierarchy and to reduce constraints on shared memory while introducing the notion of ILP. Our approach will aim in utilizing the warp intrinsics and the thread registers in a different manner in the hope of reducing memory write access to C.

3.3 SpNN: non GPU-based

This section starts tackling our proposed problem. It sheds the light on the most commonly used strategies for dealing with sparse neural networks. The papers listed utilizes CPUs and FPGAs in their implementations.

Algorithm	Device	Sparse matrix format				Partition
		Input	Weight	Buffer(s)	Output	
Mofrad et al. [13]	CPU	CSC	CSC	-	CSC	1D Tiling
KKSpDNN [14]	CPU	CSR	CSR	-	CSR	1D row
GraphBLAS [15]	CPU	CSR	CSR	-	CSR	1D row
Huang et al. [16]	FPGA	Dense	CSC	-	Dense	1D Tiling (x3)

Table 3.4: Table listing SpNN approaches on devices other than the GPU (mainly CPU)

3.3.1 Multithreaded Layer-wise Training of Sparse Deep Neural Networks using Compressed Sparse Column [13]

The proposed sparse matrix multiplication algorithm in this work follows a 1D column-wise tiling strategy. The weight matrix B is split into vertical tiles where each tile is assigned to a thread. Their work is based on a CPU architecture and the matrices are stored in CSC format. At each layer, the storage space for C is estimated using "precise method" followed by a numerical multiplication. Also, in their implementation each thread was assigned a sparse accumulator (SPA). However, they didn't use double-buffering. Hence, at each layer, they re-adjusted C, computed the results and stored them in C and had to re-adjust and copy the results to A. This presented a significant overhead while relying on many

synchronization calls between threads. It should be noted that they didn't take into account load-balancing across the threads. We might consider creating an approach that computes B instead of A.

3.3.2 Scalable Inference for Sparse Deep Neural Networks using Kokkos Kernels [14]

This work developed a meta-algorithm, denoted as "$kkSpDNN$", that builds on top of the "kkSpGEMM [10]" algorithm from the Kokkos library. The proposed meta-algorithm stores the matrices in CSR while running on either single or multiple CPU nodes. It introduced a 1D row-wise tiling across sockets where each sockets receives the same number of rows from Y and a copy of the weight. However, they came across some load balancing issues which may prevent scalability. Hence, they showed the necessity of implementing dynamic load balancing and demonstrated the benefits of data-parallelism.

In their approach, the bias was added after the sparse multiplication in a separate function call that could introduce zero values. These zero values could result in wasting the computational resources and could transform the sparse input into dense. Therefore, they implemented a trimming function that was called on some specified heuristics. This algorithm could be somewhat wasteful due to the overhead created by the symbolic phase and the trimming. Our study wishes to further explore the performance on the GPU kernels while tackling the load balancing issues and testing other storage formats.

3.3.3 Write Quick, Run Fast: Sparse Deep Neural Network in 20 Minutes of Development Time via SuiteSparse: GraphBLAS [15]

This work demonstrated that GraphBLAS can be an efficient library that allows end users to write simple yet fast code. By default, all matrices are held in CSR format and can be executed on a single or multiple CPU architecture. It follows a 1D row-wise partitioning scheme where the work is divided to a single task per thread. The sparse multiplication in this library, called using "GrBmxm", support three different forms of matrix-matrix multiply: Gustavson's method, a heap-based method, and a dot-product based method. The algorithm selects automatically the algorithm given the input's characteristics and can also be specified at the user end. In this work, the sparse deep neural network solution was done in the parallel Gustavson matrix-matrix multiplication.

Similar to Matlab, allocating the resulting matrix C was done in a "progressive method" which isn't practical on GPUs. It should be noted that the sparse matrix

computation, adding the bias, removing the zeros and adjusting the values to the upper bound were done separately, each in an independent function call in that order per iteration. Hence, this could be wasteful and might hinder performance. The performance scales up well to the max number of hardware threads per node, beyond which performance scales more modestly. This work also stated that they are currently building libraries that support GPUs. Our study will tackle a GPU-based parallel algorithm, a feature that isn't yet supported in GraphBLAS.

3.3.4 Accelerating Sparse Deep Neural Networks on FPGAs [16]

This work showed the impact and importance of FPGAs as a platforms for DNN acceleration. The input matrices are treated as dense while the DNN parameters are stored in CSC formats. It follows a 3-level partitioning scheme. The first level, groups the input feature vectors by rows of size T_images following 1D row tiling. The second level, propagates each group throughout T_layers layers of the DNN at each iteration in a manner similar to a 1D column Tiling across layers. This was applicable since the computation are independent across the rows. The final level, is applied on the computational level where the weights are divided in a 1D column partition of size $T_neurons$.

In their implementation, the groups are distributed across the pools of accelerators in the FPGA-chip. Each accelerator is composed of 2 Buffers of size $T_neurons$ and $T_neurons$ PEs are instantiated. The computations are done as sparse vector dot products where each PE processes the vector dot product of the same input feature vector, stored in one buffer, with one different column in the parameter matrix. After the computation, the results are stored in the second buffer that become the input to the second layer. Evaluation of the results showed that the proposed design achieved x4.7 better energy efficiency compared to CPU. However, the FPGA on-board memory bandwidth became the bottleneck of the whole system.

Although this work's algorithm is applied on FPGAs, it presented some interesting ideas on multi-level tiling. However, as we as we propagate throughout the layers, more and more empty-rows are generates/appear that could hinder performance. Also, they presented poor access patterns that should be avoided when dealing with GPUs. If after each layer, we re-group the input while ignoring the empty rows, we can dramatically reduce the number of tiles and hence reducing the total number of iterations needed. Therefore, we will disregard the tiling across layers process. We also aim to have a better dynamic distribution strategy.

3.4 DNN: GPU-based

This is the most relevant section for our work. It covers some papers that have tackled running a sparse neural network on the GPUs. The work done can highlight some key features that should be taken into consideration.

Algorithm	Sparse matrix format				Partition
	Input	Weight	Buffer(s)	Output	
Wang et al. [17]	CSR	CSR	CSR	CSR	1D row
GraphBLAST [18]	CSR	CSR	CSR	CSR	1D row
Bisson et al. [19]	Dense	CSR	-	Dense	1D tiling (x2)
Hidayetoglu et al.[20]	Dense	CSR	-	Dense	2D partition
	Dense	ELL	-	Dense	2D partition

Table 3.5: Table listing SpNN approaches on GPUs

3.4.1 Performance of Training Sparse Deep Neural Networks on GPUs [17]

This work proposed a Fine-tune Structured Sparsity Learning (FSSL) method to regularize the structures of DNNs and accelerate the training of DNNs. The matrices are stored in CSR format running on a single GPU architecture. Their sparse matrix multiplication is based on the ”CuSparse” library. They implemented two baseline versions by utilizing ”CusparseDcsrgemm” on the first and ”cusparseDcsrgemm2” API on the second. They also introduced a filtering step that filters the newly created zeros after each iteration. They concluded the effectiveness of a well structured sparsity pattern especially when dealing with large data.

3.4.2 Accelerating DNN Inference with GraphBLAS and the GPU [18]

This work provided a GPU implementation of the GraphBLAS library that was approximately two times faster. The matrices are stored in CSR format on a single GPU architecture following a 1D partitioning scheme. Their algorithm, denoted as GraphBLAST, executed the sparse multiplication using the ”cuSPARSE” library. However, due to its limitations and the limited on-chip GPU memory, they weren’t able to run large matrices. As future work, they proposed some solutions to the limited memory by either implementing data-parallelism, dividing the task among multiple GPUs or developing their own kernels.

Some steps, like adding bias, pruning, and filtering, were adjusted to better suit the tackled problem that resulted in a noticeable speedup. They also introduced an interesting notion of rank promotion (Numpy-style broadcasting). Also, throughout their work they compared $W^T Y^T$ vs YW. The former gave better results but this was due to the equal number of nonzeros per row in the weights that assured load-balancing. We might build on this finding in order to assess the most suitable storage formats for the matrices and deciding on either executing the SpGEMM as $W^T Y^T$ vs YW.

3.4.3 A GPU Implementation of the Sparse Deep Neural Network Graph Challenge [19]

This work was the 2019 Sparse Deep Neural Network Graph Challenge champion. It proved the suitability of GPUs to accelerate Deep Learning workloads and reduced redundant memory traffic. Their code is capable of running the computation in either single or double precision. The architecture tackled in this work consists of multiple GPUs connected via NVLink and NVSwitch to a single node (CPU). The input feature vectors Y is stored as a Dense matrix and the weights W matrices were stored in CSR format. The input was partitioned as 1D row-wise tiling (horizontal slabs) across the GPUs, each having the same number of non-empty rows. Since the number of empty-rows varies after each iteration, and since this study deals with a multi-GPU system, after each iteration they re-calculated and re-partitioned the non-empty rows equally across the available GPUs for better load-balancing [this is done by the OpenMP threads].

As for the weights, they were partitioned in a 1D column-wise tiling (vertical slabs). One of the main reason behind this partitioning was to be able to fit the resulting values in the limited GPU's shared memory capacity. Double buffering was used on the weights where the memory copy times were completely hidden by the inference computations having the kernel and the memory copy each being executed on a different stream. All the layer matrices are read into pinned host memory and allocate device memory on each GPU only for two of them. The inference function is called by one OpenMP thread per GPU. Each GPU executes two kernels, one for the multiplication+ReLU of its own part of Y and one to compute the non-empty row indices in the resulting matrix.

In the inference computations, 2D Blocks were lunched of [dimX, dimY] dimensions that varies depending on the number of neurons. The number of 2D Blocks lunched per GPU at each iteration is equal to the number of non-empty rows of that GPU's corresponding Y slab. Hence, each 2D Block will cover a single non-empty row. The Blocks will be divided into dimY sub-CTAs each consisting of dimX consecutive threads. Each sub-CTA will be responsible for

the output of a single element. However, one of the drawbacks of this approach was having to read each row number of slabs time. Therefore, in order to reduce the number of access to the input matrix, the optimization made was reading the row one time while storing the values in the threads registers where each thread was responsible for NN/threads per block.

Our study aims in testing 2D-based partitioning along both row and column while trying to overcome the size of the networks, which was discussed as the limiting factor to the scaling. We will be dynamically dividing the work of the threads in the Blocks unlike the work done in this work that was done statically with the sub-CTAs. We will be using 2 layers of double-buffering (having more than 2 streams) and we will be handling register-tiling in a different manner.

3.4.4 Efficient Inference on GPUs for the Sparse Deep Neural Network Graph Challenge 2020 [20]

This work was the 2020 Sparse Deep Neural Network Graph Challenge champion. It presents a GPU performance optimization and scaling results for SpDNN on a single and multiple GPU architecture [up to 6 GPUs per node]. It solves a $sparse$x$dense$ matrix multiplication $W^T Y^T$ instead of YW where the dense matrices are stored in column major order. It presents two different approaches: a base-case and an optimized approach. The first method stores the weights as CSR and follows a 2D partitioning approach where each thread is responsible for a single output element of the resulting matrix C. On the other hand, the second optimized method stores W as ELL(sliced) while following a 2D tiling approach where each thread is responsible for $MINIBATCH$ output elements of C.

The purposed of their initial base-case method was pinpointing the main deficiencies from which they built their main contribution in the optimized method. After analysing the first method, the main causes that hindered performance were due to random memory access to Y and redundant memory access to W. Upon tackling these issues, they proposed three performance optimizations:"register tiling", "shared memory tiling", and "compact index representation", and two memory optimizations: "Out-of-Core Storage and Overlapping Strategy" and "Compact Representation and Batching".

The tiling mainly targeted data reuse where both register tiling and shared memory tiling help in reusing elements from W and Y respectively. This also helps in reducing memory write access to C. Also, the W elements were sorted and adjusted using maps, and stored in ELL(slices) with $WARPSIZE$ in order to assure warp granularity and efficient memory access to the weights. The memory optimizations used double-buffering to load the weights that was completely

hidden by the computations. Finally, they used compact representation of the maps for batching to reduce the memory footprint.

This work presented some interesting thread distribution approach while reformulating the weights to assure (semi-)coalesced [efficient] memory access and memory reuse patterns. They respected and properly utilised the GPUs memory hierarchy. However, some minor overheads was detected by reformulating the weights and adding zero values to the ELL representation to assure warp granularity. We aim to have a similar approach in terms of memory hierarchy while utilising some 2D tiling representations. However, we wish to have a more dynamic thread distribution strategy.

Chapter 4

Approach

In our work, we propose some optimizations and highlight their theoretical effects along the way. Throughout this chapter, and as shown in Table 4.1, we will be comparing the performance with different data-partitioning strategies, storage formats, and some proposed optimization approaches. We will also be referring to previous work done for comparison and better understanding of some underlying problems.

<table>
<tr><th colspan="4">Sparse matrix format</th><th rowspan="2">Partition</th><th rowspan="2">Approach</th></tr>
<tr><th>Input</th><th>Weight</th><th>Buffer(s)</th><th>Output</th></tr>
<tr><td rowspan="2">CSC</td><td rowspan="2">CSC</td><td rowspan="2">Dense</td><td rowspan="2">CSC</td><td>1D Tiling</td><td rowspan="2">1</td></tr>
<tr><td>2D Tiling</td></tr>
<tr><td rowspan="2">CSR</td><td rowspan="2">CSR</td><td rowspan="2">Dense</td><td rowspan="2">CSR</td><td>1D Tiling</td><td rowspan="2">2</td></tr>
<tr><td>2D Tiling</td></tr>
<tr><td rowspan="2">CSR</td><td rowspan="2">ELL</td><td rowspan="2">Dense</td><td rowspan="2">CSR</td><td>1D Tiling</td><td rowspan="2">3</td></tr>
<tr><td>2D Tiling</td></tr>
<tr><td rowspan="2">Dense</td><td>ELL</td><td>-</td><td>Dense</td><td rowspan="2">2D partition</td><td rowspan="2">[20]</td></tr>
<tr><td>CSR</td><td>-</td><td>Dense</td></tr>
<tr><td>Dense</td><td>CSR</td><td>-</td><td>Dense</td><td>1D tiling (x2)</td><td>[19]</td></tr>
<tr><td rowspan="2">CSR</td><td rowspan="2">CSR</td><td rowspan="2">CSR</td><td rowspan="2">CSR</td><td rowspan="2">1D row</td><td>[17]</td></tr>
<tr><td>[18]</td></tr>
</table>

Table 4.1: Table listing previous work and our approaches for a SpNN implementations on GPUs

4.1 General Flow

Our work is focused on the created sparseNN function and its kernel calls. The naive sparseNN function, as shown in Algorithm 8, takes the input feature vec-

tors $Y0$ and the array of Weights $W[L]$ with a *bias* as input and returns a vector as output. This vector will contain the non-empty row indices of the resulting matrix after propagating over the Neural Network. We will then compare the produced output vector with the provided Categories vector to validate our results.

Algorithm 8 Naive sparseNN function

Input: $Y0_{mxn}$, $W_{nxn}[L]$, $bias$
Output: v_m

```
 1: procedure SPARSENN(Y0, W[L], bias)
 2:     YMAX ← 32
 3:     Y ← Y0
 4:     for l in L do
 5:         nnz ← SpGEMM_symbolic(Y, W[l])
 6:         if nnz ≥ sizeof(Z) then
 7:             resize(Z, nnz), resize(Y, nnz)
 8:         Z ← SpGEMM(Y, W[l])
 9:         Y ← addBias(Z, bias)
10:         boundary_check(Y, 0, YMAX)
11:     v ← MatrixToVector(Y)
12:     return v
```

Each iteration l could produce 0 or negligible values that should be pruned, values greater than 32 (our upper limit) that should be adjusted, empty rows and rows that could become more and more dense. Also, the number of non-empty values could increase, by which we might have to increase the allocated size of our matrices. Some implementations, like GraphBLAS, created function calls to handle each of these issues separately. Similarly, our naive base-case implementation starts by calculating the upper-bound size needed using the SpGEMM_symbolic kernel, and resizing the matrices accordingly. It will then compute $Y * W[l]$ and store our results in the temp buffer Z with the SpGEMM kernel call. After the computation, a *bias* is added to the values of Z and copied back to Y. Finally, the boundary_check kernel call will remove the negative and zero-values and adjust the values to be bounded by the upper limit YMAX.

4.2 General Optimizations

At first glance, our naive base-case implementation suffers from excessive kernel calls that could be merged and lots of unnecessary memory transfers that

could be avoided. Hence, our implementation is further refined by removing both the addBias and boundary_check kernel calls and integrating their functionality within the SpGEMM kernel, reducing the total number of kernel calls per iteration. Furthermore, when propagating in the Neural Network, and as illustrated in Figure 5.2, the output of each iteration l will become the input of the next iteration $l+1$. Therefore, as shown in Algorithm 9, we utilized double-buffering by creating 2 buffers and swapping between them, eliminating the copy overhead of line 9 in Algorithm 8.

Algorithm 9 sparseNN function

Input: $Y0_{mxn}$, $W_{nxn}[L]$, $bias$
Output: v_m

```
 1: procedure SPARSENN(Y0, W[L], bias)
 2:     YMAX ← 32
 3:     Y_input ← Y0
 4:     Y_output ← ∅
 5:     batchSize ←< value >              ▷ Set number of cols|rows per batch
 6:     B_uffer ← initBuffer(Y|W, batchSize)
 7:     for l in L do
 8:         numBatchs ← Y_input/batchSize
 9:         for b in numBatchs do
10:             empty(B_uffer)
11:             B_uffer ← SpGEMM(Y_input, W[l], b, bias, YMAX)
12:             scan(B_uffer)
13:             expand(Y_output, nnz(B_uffer))
14:             flush(Y_output, B_uffer, b)
15:         swap(Y_output, Y_input)
16:     v ← MatrixToVector(Y_input)
17:     return v
```

The sparse matrix algorithm used withing the SpGEMM function will obviously vary depending on our desired storage formats used for each of the input, weights and output matrices. Also, it should be noted that the *expand*() kernel is only called when needed and it re-allocates the sparse matrix to the exact storage requirement and doesn't follow the convention of doubling its size.

4.2.1 Batching

Unlike CPUs, the process of re-allocating memory on the GPU is rather expensive. Hence, and as mentioned in section 2.4, a general approach would be conducting

a symbolic sparse matrix multiplication in order to estimate and allocate the required size for the resulting matrix. However, this approach isn't optimal since we are essentially calling the sparse matrix multiplication 2 times at each layer. In order to mitigate this bottleneck, and taking into consideration the limited on-chip memory capacity of the GPU, we decided to allocate a Dense Buffer of size $BatchSize$. Depending on the storage formats used, $BatchSize$ could either refer to a number of columns per batch $colsPerBatch$ when dealing with Approach 1 or to a number of rows per batch $rowsPerBatch$ when dealing with Approaches 2 and 3. Hence, $BatchSize = (rowsPerBatch * Y.numCols)|(colsPerBatch * Y.numRows)$ where the value of $rowsPerBatch|colsPerBatch$ is provided by the user.

4.3 Sparse Storage Formats

One of the major improvements that could be done when tackling sparse matrix multiplications is choosing the most suitable sparse storage formats for our input, weight and output matrices. In this section we will be proposing and discussing different sparse storage combinations while assessing their impact. The storage formats that will be used or discussed in this study include (D)CSR, (D)CSC and ELL.

4.3.1 Combinations

As listed in Table 4.1, we present three different storage format combinations, each with two different tiling strategies (1D and 2D tiling). We aim at finding the best approach among them by assessing and comparing them with each other.

- **Approach 1 – CSC-CSC:** Following the dependency between the columns of the inputs with their corresponding rows from the weights, this implementation distributes the blocks across the columns of the weights in CSC within the batch. The threads will iterate over the rows of the columns from the weight matrix in a coalesced manner. Each thread will then map to its corresponding column from the input matrix in CSC (not coalesced) to perform the computation. The computations are done in shared-memory before flushing to the Buffer (coalesced). In the 1D tiling, each blocks will be in-charge of filling its corresponding columns unlike the 2D tiling where columns could be filled by multiple blocks.

- **Approach 2 – CSR-CSR:** Similar to Approach 1, and following the dependency between the columns of the inputs with their corresponding rows from the weights, this implementation distributes the blocks across the rows of the input stored in CSR within the batch. The threads will iterate over

the columns of the rows from the input matrix in a coalesced manner. Each thread will then map to its corresponding row from the weight matrix stored in CSR (not coalesced) to perform the computation. The computations are done in shared-memory before flushing to the Buffer (coalesced). In the 1D tiling, each block will be in-charge of filling its corresponding rows unlike the 2D tiling where rows could be filled by multiple blocks.

- **Approach 3 – CSR-ELL:** An optimization to Approach 2, this implementation distributes the blocks across the rows of the input stored in CSR within the batch. The threads will iterate over the columns of the rows from the input matrix in a coalesced manner. Each thread will then map to its corresponding row from the weight matrix stored in ELL ((semi-)coalesced depending on the input rows) to perform the computation. The computations are done in shared-memory before flushing to the Buffer (coalesced). In the 1D tiling, each blocks will be in-charge of filling its corresponding rows unlike the 2D tiling where rows could be filled by multiple blocks.

4.4 Partitioning

Parallelizing any application on the GPU involves distributing the work across the number of available computational units, where the overall execution time of the application will be dependent on the time taken by the last unit to terminate. By ensuring load-balancing, such that the workload is properly/equally distributed among these units, we can increase scalability and improve performance. When dealing with a dense matrix, load-balancing could be easily achieved by simply distributing the units following various partitioning strategies since the workload will almost be the same.

However, when dealing with sparse matrices, we face 2 main problems leading to poor load-balancing. The first problem is encountering empty rows and columns, resulting in control divergence. The second problem is the extreme variation of the workload distribution across the computational units, resulting in inactive threads. These problems are caused by the matrix's random sparsity patterns. If not handled properly, they could hinder performance and waste computational resources. Hence, in order to tackle these 2 problems, we proposed 2 approaches: doubly-compressed row or column format, and 2D tiling. Theses approaches will be carried out with our proposed various sparse storage formats combinations discussed above.

4.4.1 Doubly-Compressed format

Assuming an input sparse matrix consisting of M rows and N column where we wish to assign T rows per B blocks. As shown in Figure 4.1, each block

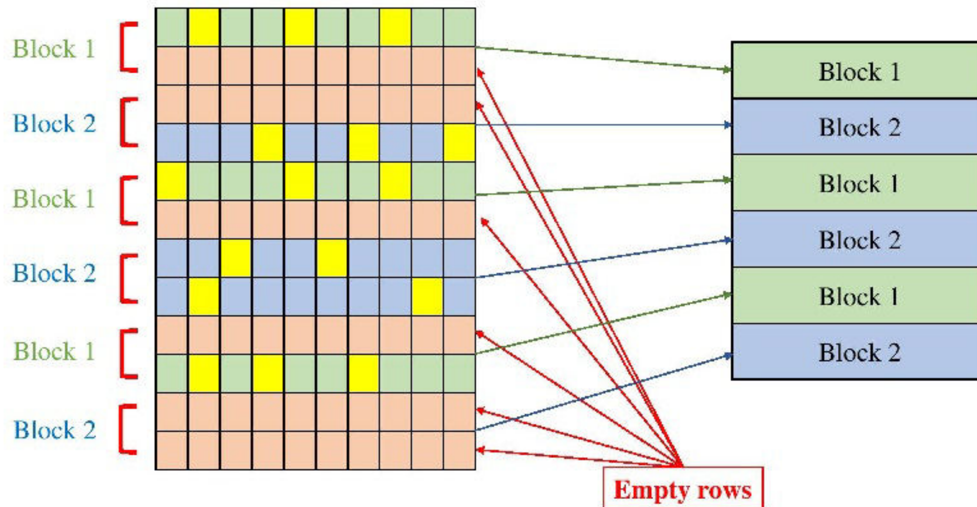


Figure 4.1: Work distribution using CSR

could encounter different number of empty rows. The number of iteration needed per block will be $(M/T)/B$. Also, tacking into consideration the possibility of newly formed empty-rows when propagating in a Neural Network, this will lead to extreme load-imbalances across the blocks. Hence, as shown in Figure 4.2, we decided to disregard the empty-rows and only partition the non-empty rows among the blocks. By this reasoning, we shifted from $(M/T)/B$ iterations/block to $(nnz(M)/T)/B$ iterations/block where $nnz(M)$ represents the non-empty rows of our input matrix.

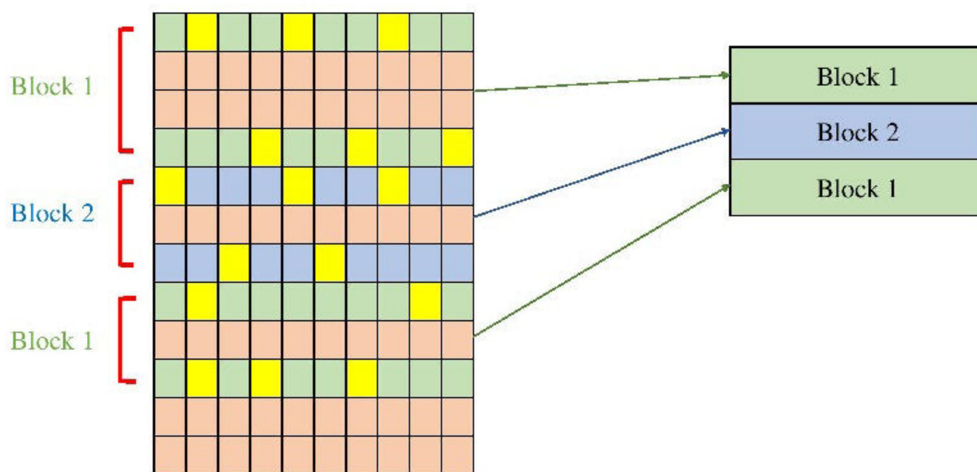


Figure 4.2: Work distribution using DCSR

In our implementation, and since some rows could become empty after each

iteration, we decided to re-identify the non-empty rows and re-distribute them accordingly across the blocks after each iteration. This is also applied to the columns when dealing with CSC. It should be noted that this strategy isn't only limited to the input matrix and could also be implemented on the weight matrix resulting in a 2-level compression effect. However, this could also offer some overhead that could negate its benefits since we have different weights per iterations. This strategy could dictates either the use of the DCSR||DCSC formats or by simply creating an array that keeps track of the non-empty row||column indices respectively. This is applied at line 8 in our Algorithm 9 where the $numBatchs$ is calculated by dividing the number of non-empty (rows | columns) over the $batchSize$.

4.4.2 2D Tiling

Our 1D-tiling used initially helps in attaining better partitioning of the input matrices row-wise or column-wise depending on our approach. However, the workload done per block is also correlated to the number of non-zero elements within its assigned tile. Hence, in order to limit the extreme workload variation, we decided to add another layer of partitioning to attain a 2D tiling representation of our input and weight sparse matrices as shown in Figure 2.6d. Therefore, each block would get assigned to a T_{row} by T_{column} tile in Y denoted as a submatrix of Y.

This could add a non-negligible overhead, however we wish to prove that the advantages offered out-weights the disadvantages. Assuming that our input matrix Y is divided into R by K tiles and that the weight matrix is divided into K by C tiles. Having B blocks, each block would need to access on average $(R * K)/B$ tiles from Y presenting more work per block. Furthermore, each block would only need to access a portion of W, by accessing the corresponding C tiles from W. For further illustration, in Figure 2.6d, the block assigned to $A_{1,1}$ would only need to access the tiles $B_{1,1}$ and $B_{1,2}$ from B and only writes to the tiles $C_{1,1}$ and $C_{1,2}$ from C.

The number of blocks launched will be equal to the number of tiles within our batch, where each block will be responsible of computing its own tile. This will increase parallelism by increasing the number of blocks needed while limiting the overall work required per block depending on the tile dimensions.

Chapter 5

Methodology

The datasets [4] used in this work originates from the MNIST (Modified National Institute of Standards and Technology) database. This database consists of a large collection of handwritten images that are widely used for training and testing DNN image processing systems. Our initial input matrix $Y0$ is composed of a constant 60,000 handwritten digit images, of $28x28$ pixel each. All the images have been resized to 3232(1024 neurons), 6464 (4096 neurons), 128128 (16384 neurons), and 256256 (65536 neurons). As visualized in Figure 5.1, the input $Y0$ was produced by flattening each image to a single row and stacking them together where each row Y_{i*} represents an image. It should be noted that the images have been thresholded so that all values are either 0 or 1.

The provided Weight matrices were modeled following the images sizes, where the number of neurons is either $NN = [1024, 4096, 16384, 65536]$. In this work, we will be only using $NN = [1024, 4096, 16384]$ since they are enough to give a good estimate of our overall performance. The non-zero values of the matrices are written as triples (i, j, v) representing the row index, column index and the value respectively. These values are stored in a .tsv file with a single non-zero value per line. The input was propagated across L layers where $L = [120, 480, 1920]$. Hence, this amounts to 12 different DNNs $[(1024, 4096, 16384, 65536)x(120, 480, 1920)]$. At each layer l, the resulting matrix was computed along the following inference computation:

$$Y_{l+1} = ReLU(Y_l W_l + b_l)$$

where Y_l and Y_{l+1} are MxN matrices of M input features of length N, respectively, W_l is an NN matrix of activation weights, b_l is an $Mx1$ bias vector for each output, and $ReLU$ is the activation function defined as

$$ReLU(x) = max(0, min(x, 32))$$

.

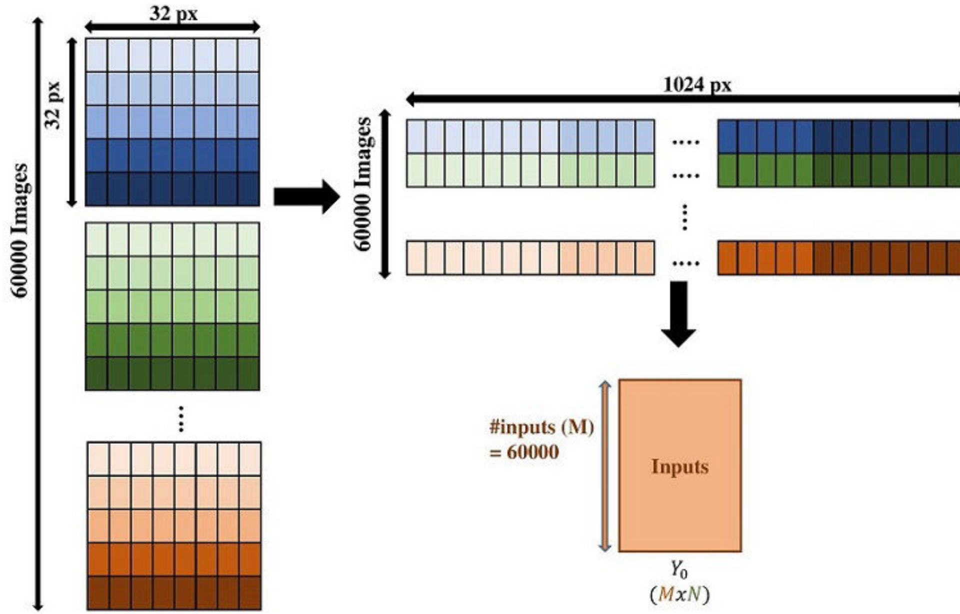


Figure 5.1: SpDNN initial input representation

Resizing our images dictates different storage requirements that varies from 176MB (for the 1024 neurons) up to 16.3GB (for the 65536 neurons) of storage. Thus, depending on our GPU version, we might encounter problems while fitting the data on the GPU due to the limited hardware (limited main memory space).

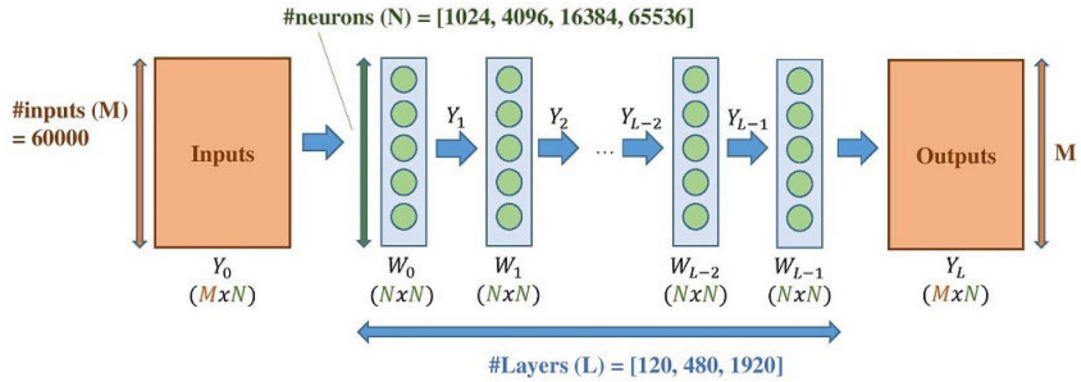


Figure 5.2: SpDNN representation

Throughout the evaluation, we will be testing our approaches on 9 different sparse neural networks [(NN: 1024, 4096, 16384)x(NL: 120, 480, 1920)]. The large sparse DNNs are synthetically generated by a RadiX-Net sparse DNN generator [24], where the number of connections per neuron is uniformly distributed such that the number of non-zeros per row is always 32 in all the weights.

Also, upon propagating in the neural network, we notice a huge decrease in the number of non-empty rows of our input matrix where the number of non-empty rows decreases from 60000 to 1812 (an approximate 97% decrease). By addressing this, we decided to store our inputs in the doubly compressed CSR format in some of our approaches. This format will only target the non-empty rows hence offering a considerable optimization.

Furthermore, and as depicted in Figure 5.3, the remaining non-empty rows are becoming fully dense after the first few layers. Therefore, and since we are only distributing our blocks on the non-empty rows, the computations are shifting from a sparse-sparse matrix multiplication to a dense-sparse matrix multiplication. For each layer, the non-zeros tend to be evenly distributed across the non-empty rows. As such, we also consider storing our inputs in the ELL format that is better suited for handling this when compared with other sparse storage formats. An interesting workaround would be switching between a SpGEMM and GEMM depending on a threshold, however this wasn't addressed withing the scope of this study.

Our kernels are implemented using CUDA and our experiments are performed on a single Nvidia V100 GPU. The number of blocks launched and their corresponding dimensions and distributions within the grid varies depending on the chosen $TILE_SIZE$, the amount of shared memory available in the Device and the provided batch size. Throughout our experiments, we allocated a Dense buffer of size equal to 10% of that of a fully dense buffer such that $sizeof(B_{uffer}) = 10\%(Y.numRows * Y.numCols)$. The $TILE_SIZE = 512$ resulting in an average of 16 blocks launched per SM with dimensions (64 x 2) threads each. Different parameters yields different results and hence opens the door for further optimizations with hyper-parameter tuning. However, this wasn't tackled since it is outside the scope of this study and is added to future works.

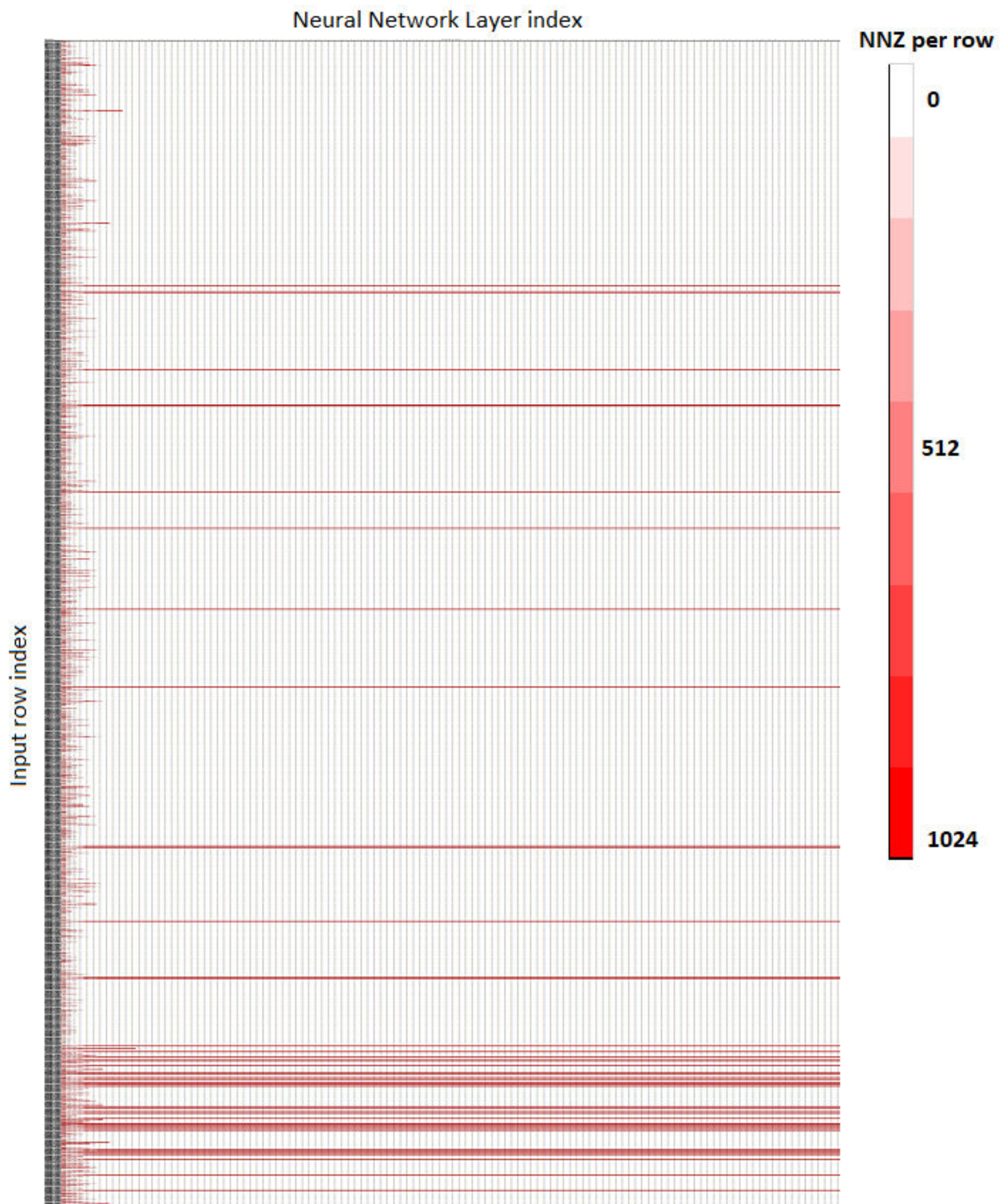


Figure 5.3: Partial representation of the variation in the number of non-zeros per row of our Input matrix in the 1024 neurons by 120 layers SpNN

Chapter 6

Evaluation

Throughout this chapter we will be comparing our proposed approaches in terms of the storage formats and partitioning strategy used. We will then show the effects of different buffer sizes before finally comparing our top approach with other works tackling the same problem and running within the same environment. It should be noted that our evaluations are specific/limited to the dataset used and could vary accordingly.

6.1 Comparing Storage Formats and Partitioning

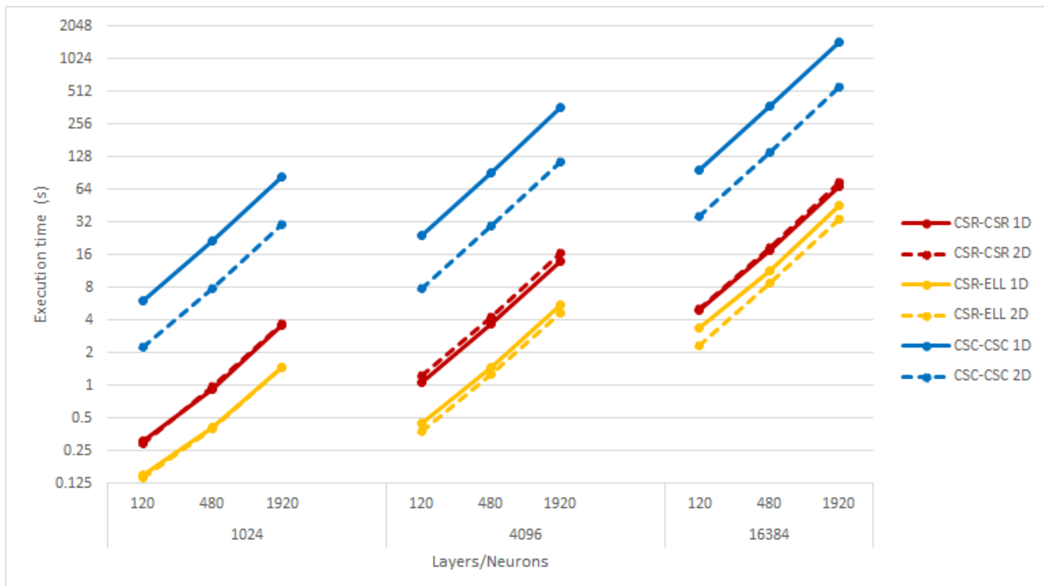


Figure 6.1: Comparing the execution time of our approaches, each along both 1D and 2D partitioning, on our 9 datasets.

		CSC*CSC->CSC		CSR*CSR->CSR		CSR*ELL->CSR	
Neurons	**Layers**	**1D**	**2D**	**1D**	**2D**	**1D**	**2D**
1024	**120**	5.94	2.25	0.31	0.29	0.149	0.14
		21.64	7.9	0.93	0.97	0.41	0.4
		84.58	30.74	3.56	3.64	1.47	1.47
4096	**120**	24.37	7.9	1.07	1.22	0.44	0.38
		91.3	29.1	3.67	4.3	1.45	1.26
		360.13	114.05	14.1	16.76	5.49	4.7
16384	**120**	96.8	35.95	4.9	5	3.35	2.35
		370.35	141.11	17.56	18.72	11.5	8.64
		1467.28	561	68.35	73.5	45.44	33.8

Table 6.1: Experimental results of the total execution time, in seconds, of our approaches. The red values represent our best results.

		CSC*CSC->CSC		CSR*CSR->CSR		CSR*ELL->CSR	
Neurons	**Layers**	**1D**	**2D**	**1D**	**2D**	**1D**	**2D**
1024	**120**	1.45	3.827	27.78	29.7	57.8	61.5
		1.42	3.88	32.96	31.6	74.77	76.64
		1.4	3.86	33.37	32.645	80.83	80.83
4096	**120**	1.34	4.15	30.7	26.92	74.64	86.83
		1.32	4.14	32.82	28.02	83.1	95.62
		1.31	4.13	33.4	28.1	85.8	100.2
16384	**120**	1.37	3.7	27.14	26.6	39.7	56.6
		1.36	3.6	28.8	27.04	44.03	58.6
		1.36	3.56	29.25	27.2	44	59.16

Table 6.2: Experimental results of the GFLOPS of our approaches. The red values represent our best results.

6.1.1 Storage Formats

From Figure 6.1 and Table 6.1, and although the CSC-CSC multiplication is considered as the inverse of the CSR-CSR multiplication, having similar advantages and disadvantages, it offers very poor performance in comparison where the CSR-CSR approach is up to (6x) faster. This is due to the huge difference in dimensions between the number of rows and columns of our resulting matrix where the number of rows (60000) is up to (58.5x) larger than that of the number of columns.

On the other hand, the CSR-ELL multiplication offered the best results throught all the 9 datasets where it gave up to (25.6x) and (2.7x) speedup in comparison with that of Approaches 1 and 2 respectively. This is due to the (semi-) coalesced data access pattern offered while accessing the weights in ELL

format. Therefore, it is safe to deduce that Approach 3 is the best approach out of the three.

6.1.2 1D Vs 2D Tiling

From Figure 6.1 and Table 6.1, the 2D tiling approach offers, in most of the cases, some (noticeable) improvements in the sparse matrix multiplications execution time. This is especially apparent within the CSC-CSC approach, were the 2D strategy offers between (4x) up to (9x) speedup over that of 1D. However, this speedup isn't very prominent in the CSR-ELL approach, were the speedup reaches up to (1.18x) speedup over that of the 1D partitioning. Also, the 2D partitioning actually resulted in sligthly worse performance for the CSR-CSR multiplication approach. Again, as previously stated, since our input matrix is becoming dense as we are propagating within the neural network, it might be negating the benefits of 2D tiling.

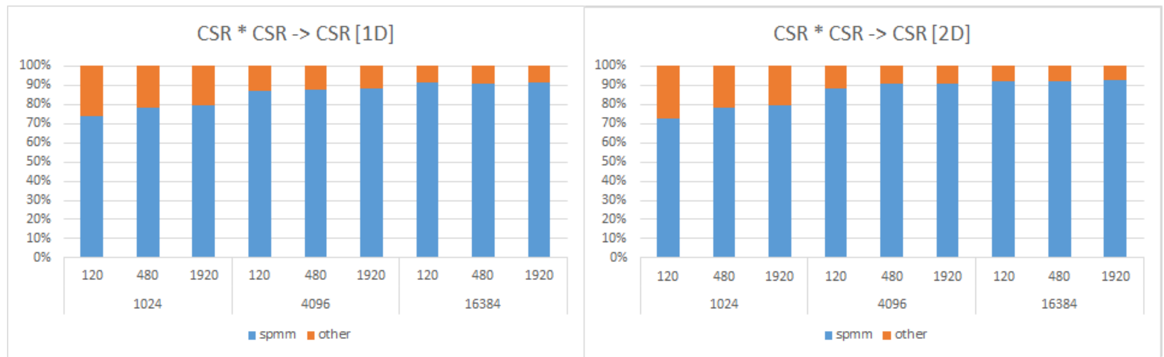


Figure 6.2: Comparing the execution time of the CSR-CSR multiplication with the total execution time.

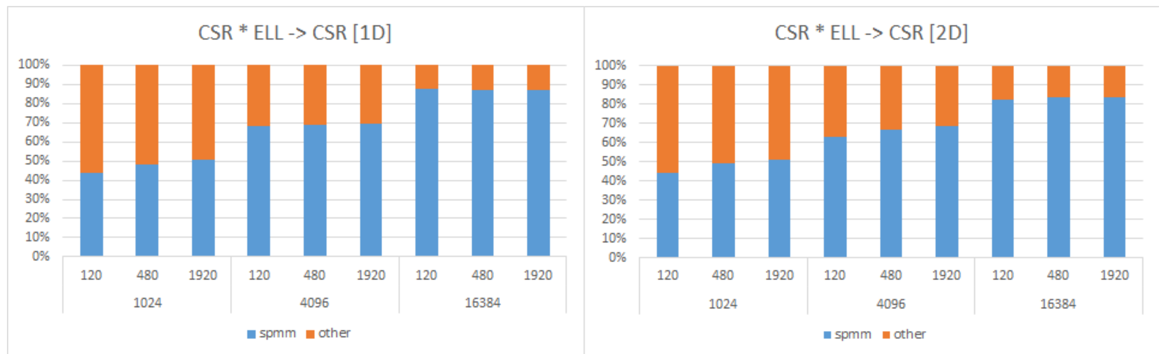


Figure 6.3: Comparing the execution time of the CSR-ELL multiplication with the total execution time

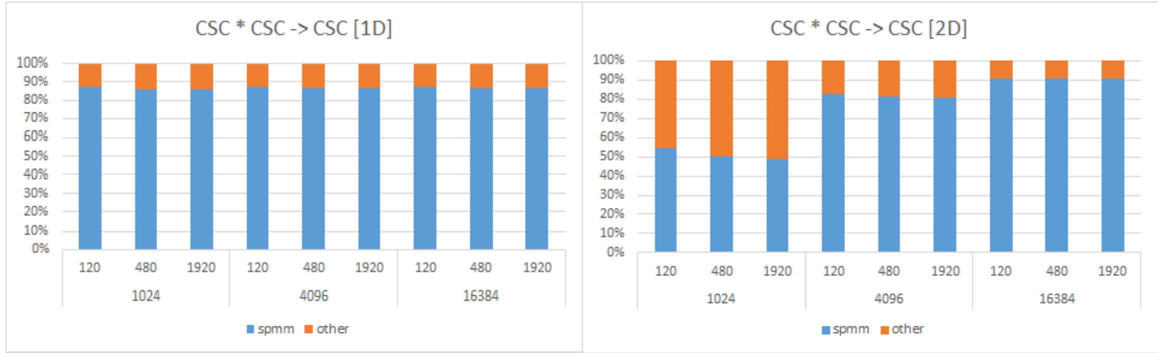


Figure 6.4: Comparing the execution time of the CSC-CSC multiplication with the total execution time

Finally, and as shown in the Figures 6.2, 6.3 and 6.4, the 2D partitioning slightly decreases the overall execution percentage of the sparse multiplications [This is especially apparent in 6.4 with 1024 neurons]. Therefore, the benefits of 2D tiling out-weights that of 1D tiling since it increases the parallelism by providing more work in smaller quantities (limiting the negative effects of load-imbalances).

Neurons	**Layers**	**CSR*CSR->CSR [1D]**						**CSR*CSR->CSR [2D]**					
		10% buffer		**25% buffer**		**50% buffer**		**10% buffer**		**25% buffer**		**50% buffer**	
		spmm	**total**	**spmm**	**total**	**spmm**	**total**	**spmm**	**total**	**spmm**	**total**	**spmm**	**total**
1024	**120**	0.23	0.31	0.21	0.27	0.21	0.27	0.21	0.29	0.23	0.3	0.23	0.28
	480	0.73	0.93	0.73	0.93	0.73	0.91	0.76	0.97	0.77	0.97	0.75	0.93
	1920	2.83	3.56	2.82	3.53	2.81	3.52	2.9	3.64	2.89	3.6	2.89	3.6
4096	**120**	0.93	1.07	0.92	1.04	0.9	1.02	1.08	1.22	1.07	1.19	1.07	1.2
	480	3.23	3.67	3.22	3.65	3.23	3.66	3.9	4.3	3.92	4.32	3.92	4.32
	1920	12.44	14.1	12.43	14	12.43	14	15.2	16.76	15.2	16.74	15.22	16.79
16384	**120**	4.47	4.9	4.38	4.79	4.36	4.76	4.6	5	4.64	5.04	4.63	5.03
	480	16	17.56	21.4	24.65	15.97	17.44	17.3	18.72	17.3	18.7	17.3	18.7
	1920	62.57	68.35	62.44	68.25	62.4	68.2	68	73.5	68.17	73.84	68.13	73.6

Table 6.3: Variation of the execution time depending on the buffer size for the CSR multiplications.

Our approaches exhibits linear scaling behaviour as our data gets larger.This is apparent from the Table 6.1, where increasing our dataset size by 4 resulted in a (4x) increase of our execution time. Therefore, our implementation could be considered as a good benchmark for comparison between different data storage formats.

Neurons	Layers	CSR*ELL->CSR [1D]						CSR*ELL->CSR [2D]					
		10% buffer		25% buffer		50% buffer		10% buffer		25% buffer		50% buffer	
		spmm	total	spmm	total	spmm	total	spmm	total	spmm	total	spmm	total
1024	**120**	0.065	0.149	0.063	0.124	0.062	0.11	0.062	0.14	0.063	0.13	0.062	0.119
	480	0.198	0.41	0.196	0.38	0.19	0.38	0.198	0.4	0.2	0.4	0.198	0.379
	1920	0.75	1.47	0.74	1.45	0.74	1.42	0.75	1.47	0.75	1.46	0.76	1.46
4096	**120**	0.3	0.44	0.289	0.41	0.288	0.4	0.24	0.38	0.24	0.37	0.23	0.35
	480	1	1.45	1	1.44	0.99	1.42	0.84	1.26	0.83	1.25	0.83	1.23
	1920	3.83	5.49	3.83	5.49	3.8	5.44	3.23	4.7	3.23	4.79	3.23	4.76
16384	**120**	2.93	3.35	2.69	3.1	2.67	3	1.93	2.35	1.93	2.34	1.93	2.32
	480	10	11.5	10.18	11.6	10.4	11.9	7.22	8.64	7.22	8.63	7.23	8.67
	1920	39.69	45.44	41	46.8	42.45	48.16	28.3	33.8	28.4	33.82	28.39	33.8

Table 6.4: Variation of the execution time depending on the buffer size for the ELL multiplications.

Neurons	Layers	CSC*CSC->CSC [1D]						CSC*CSC->CSC [2D]					
		10% buffer		25% buffer		50% buffer		10% buffer		25% buffer		50% buffer	
		spmm	total	spmm	total	spmm	total	spmm	total	spmm	total	spmm	total
1024	**120**	5.17	5.94	2.51	2.94	1.26	1.52	1.22	2.25	1.18	1.63	1.17	1.47
	480	18.63	21.64	9	10.7	4.52	5.52	3.95	7.9	3.8	5.5	3.8	4.95
	1920	72.5	84.58	35.22	41.89	17.56	21.52	14.95	30.74	14.29	21.18	14.13	18.69
4096	**120**	21.29	24.37	3.93	4.6	-	-	6.54	7.9	6.54	7.32	-	-
	480	79.26	91.3	14.6	17.2	-	-	23.68	29.1	23.78	26.86	-	-
	1920	312.1	360.13	57.32	67.67	-	-	92.32	114.05	92.55	104.65	-	-
16384	**120**	84.38	96.8	-	-	-	-	32.72	35.95	-	-	-	-
	480	321.6	370.35	-	-	-	-	128.23	141.11	-	-	-	-
	1920	1272.24	1467.28	-	-	-	-	510	561	-	-	-	-

Table 6.5: Variation of the execution time depending on the buffer size for the CSC multiplications.

6.2 Data Sets and Buffer Size

We aim at showing the effects of different buffer sizes over the overall execution time of the SpDNN. Upon testing, changing the buffer size showed negligible speedups for both the CSR-CSR and the CSR-ELL approaches as shown in Tables 6.3 and 6.4. However, the buffer size had an extreem effect in the CSC-CSC approach, where we obtained up to (2x) speedups. By analysing the input data, the total number of non-empty rows drop from 60000 to 1812 (a 97% drop) after a few layers. The remaining number of non-empty rows is less than our smallest buffer size (10%) which could explain the negligible speedups.

6.3 Comparison with other implementations

6.3.1 Execution time

In order to properly evaluate our work we decided to compare our best approach with other implementations covering the same datasets running within the same environment. The 2 previous implementations chosen were to most prominent having top results. Upon comparing our results and as shown in Figure 6.5, we

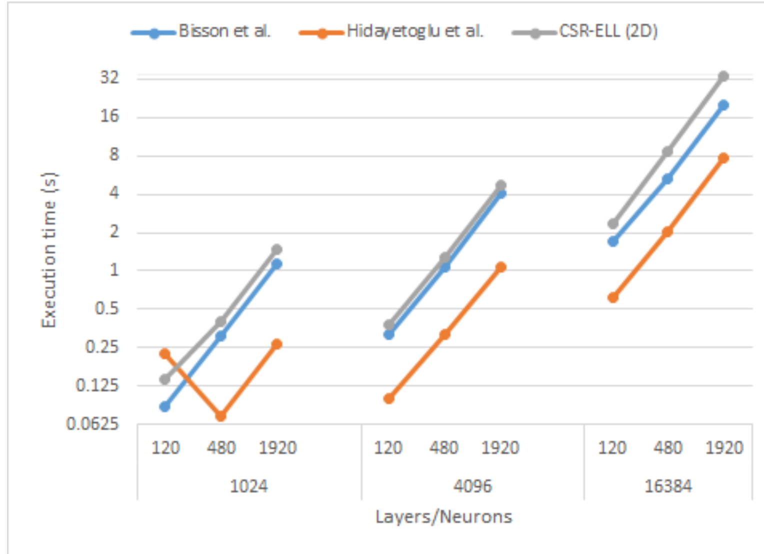


Figure 6.5: Comparing the execution time of our top approach, the 2D CSR-ELL multiplication, against both Hidayetoglu et al.[20] and Bisson et al. [19]

Neurons	Layers	Bisson et al. [19]	Hidayetoglu et al.[20]	Approach 3 [2D]
1024	**120**	0.086s	0.225s	0.14s
	480	0.306s	0.073s	0.4s
	1920	1.143s	0.264s	1.47s
4096	**120**	0.321s	0.1s	0.38s
	480	1.068s	0.322s	1.26s
	1920	4.069s	1.08s	4.7s
16384	**120**	1.695s	0.614s	2.35s
	480	5.352s	2.027s	8.64s
	1920	20.012s	7.704s	33.8s

Table 6.6: Comparing our top approach with previous implementations

were approximately (1.5x) slower than Bisson et al. [19] and (4.3x) slower than that of Hidayetoglu et al.[20]. Their work stored the input matrices as Dense and tailored their implementations accordingly, therefore eliminating the overheads of maintaining a sparse matrix. Furthermore, the output of their computations is directly flushed to their output buffer unlike our implementations that needs to pass through an intermediate buffer dictating an increase in the total amount of needed memory transfer. However, our strategy enabled us to only utilize 25% memory footprint for the inputs as we show in the next section.

6.3.2 Memory footprint

Neurons	Bisson et al. [19] & Hidayetoglu et al.[20]	Our Implementation
1024	491.52 MB	123.36 MB
4096	1966.08 MB	492 MB
16384	7864.32 MB	1966.56 MB

Table 6.7: Comparing the memory footprint of the input matrices of previous implementations with our own using a 10% buffer size while disregarding the weights.

Depending on the sparsity of the inputs and the weights matrices, the process of batching with certain batch-sizes allows us to potentially obtain a significantly less memory footprint than that of previous work done that allocated 2 fully dense buffers ([20] and [19]). The amount of storage required by the previous implementations, while disregarding some vectors due to their negligible memory footprint in comparison, is approximately equal to $2 * (Y.numRows * Y.numCols) + sizeof(W[l]) + sizeof(W[l+1])$. In contrast, our implementation requires approximately $sizeof(Y_{input}) + sizeof(Y_{output}) + sizeof(W[l]) + sizeof(W[l+1]) + sizeof(B_{uffer})$. As shown in Table 6.7, while disregarding the memory allocated for the weights since they are almost the same, for a 10% buffer size our implementation only utilizes 25% of the memory footprint allocated by other implementations. These values were calculated using the max number of non-zeros obtained within the neural network with single precision data type.

Chapter 7

Conclusion

In this thesis we implemented and compared the performance of multiple sparse matrix multiplications within a sparse deep neural network on a GPU using different combinations of sparse storage formats. From our experiments, we were able to deduce the best suited format to our given problem. We also evaluated the effects of 2 partitioning strategies where we proved the advantages of 2D partitioning over that of 1D. Finally, we compared our top approach with other implementations running on the same environment and using the same datasets. Our comparison showed slower execution time but offered better memory efficiency, demanding less storage requirements.

Our evaluation showed the advantages of having coalesced memory access withing the GPU. Therefore, as future work, we can focus on testing other storage formats that offer better coalesced memory access patterns. It should be noted that, since our implementations have many parameter dependencies, applying some parameter tuning could increase our overall execution efficiency. Also, since our work focuses on a specific dataset, we aim in testing a variety of datasets in order to obtain a more proper evaluation of our approaches. Finally, an interesting feature to add is shifting from a SpMSpM (sparse-sparse) matrix multiplication to SpMM (sparse-dense) matrix multiplication given a certain threshold of the input sparsity.

Bibliography

[1] F. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978. cited By 190.

[2] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[3] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "Klap: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[4] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 9 2019.

[5] M. J. Serrano, "Efficient implementation of sparse matrix-sparse vector multiplication for large scale graph analytics," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 9 2019.

[6] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—matrix multiplication for the gpu," *ACM Trans. Math. Softw.*, vol. 41, Oct. 2015.

[7] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C54–C71, 2015.

[8] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 370–381, 2014.

[9] W. Liu and B. Vinter, "A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 47 – 61, 2015. IPDPS 2014 Selected Papers on Numerical and Combinatorial Algorithms.

[10] M. Deveci, C. Trott, and S. Rajamanickam, “Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures,” *Parallel Computing*, vol. 78, pp. 33–46, 2018.

[11] C. Yang, A. Buluç, and J. D. Owens, “Design principles for sparse matrix multiplication on the gpu,” in *Euro-Par 2018: Parallel Processing* (M. Aldinucci, L. Padovani, and M. Torquati, eds.), (Cham), pp. 672–687, Springer International Publishing, 2018.

[12] S. Diab, M. G. Olabi, and I. El Hajj, “KTrussExplorer: Exploring the design space of k-truss decomposition optimizations on gpus,” in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2020.

[13] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud, “Multithreaded layer-wise training of sparse deep neural networks using compressed sparse column,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, 9 2019.

[14] J. A. Ellis and S. Rajamanickam, “Scalable inference for sparse deep neural networks using kokkos kernels,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 9 2019.

[15] T. A. Davis, M. Aznaveh, and S. Kolodziej, “Write quick, run fast: Sparse deep neural network in 20 minutes of development time via suitesparse:graphblas,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, 9 2019.

[16] S. Huang, C. Pearson, R. Nagi, J. Xiong, D. Chen, and W. Hwu, “Accelerating sparse deep neural networks on fpgas,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 9 2019.

[17] J. Wang, Z. Huang, L. Kong, J. Xiao, P. Wang, L. Zhang, and C. Li, “Performance of training sparse deep neural networks on gpus,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–5, Sep. 2019.

[18] X. Wang, Z. Lin, C. Yang, and J. D. Owens, “Accelerating dnn inference with graphblas and the gpu,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, Sep. 2019.

[19] M. Bisson and M. Fatica, “A gpu implementation of the sparse deep neural network graph challenge,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, 9 2019.

[20] M. Hidayetoglu, C. Pearson, V. S. Mailthody, E. Ebrahimi, J. Xiong, R. Nagi, and W. mei W Hwu, “Efficient inference on gpus for the sparse deep neural network graph challenge 2020,” in *Arxiv pre-print*, 2020.

[21] S. Dalton, N. Bell, L. Olson, and M. Garland, “Cusp: Generic parallel algorithms for sparse matrix and graph computations,” 2014. Version 0.5.0.

[22] NVIDIA, “Cusparse.”

[23] H. Carter Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[24] J. Kepner and R. Robinett, “Radix-net: Structured sparse matrices for deep neural networks,” *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019.