DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Porting MLEM Algorithm for Heterogeneous Systems

Mengdi Wang

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Porting MLEM Algorithm for Heterogeneous Systems

Portierung des MLEM Algorithms für heterogene Systeme

Author:	Mengdi Wang
Supervisor:	Prof. Dr. rer. nat. Martin Schulz
Advisor:	M.Sc. Dai Yang
Submission Date:	September 15th, 2019

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, September 15th, 2019 Mengdi Wang

Acknowledgments

First and foremost, I would like to give my greatest appreciation to Dai Yang for initially giving me various topic options for my bachelor's thesis. Furthermore, he has never once hesitated to provide me the help that I needed while composing and implementing. He has been always very patient with me, no matter how ignorant I was and how ambiguous my questions were.

I could also not be more thankful to Prof. Martin Schulz for supervising my thesis and giving me suggestions on colloquium.

I would also like to thank the Department of Informatics in Technical University of Munich for granting me the access to the machines. And I would not have been able to accomplish my work without the extraordinarily help documentations of NVIDIA libraries.

I am also deeply appreciated that the GPU resources in this thesis are sponsored by NVIDIA corporation. And I have to give my thanks to LRZ for providing access to the DGX-1 machine.

In the end, a big thanks goes to my girlfriend for her support to me throughout. It wouldn't be possible for me to hold on to the last without her companion.

Abstract

Positron Emission Topography Tomography (PET) is a nuclear medicine image technique. The scanning machine of PET can be characterized by a (sparse) matrix A. The result of scanning is stored in an image vector g. The image reconstruction of PET is not different from solving a linear equation system $Af = g$.

The main purpose of this thesis is to develop and compare different implementations of Maximum Likelihood Expectation Maximization (MLEM) algorithm, which can be used to solve the linear equation system $Af = g$. The target platform of all implementations are heterogeneous systems with NVIDIA GPUs that enable CUDA. The host program is written in C/C++, while the GPU kernel functions are written in CUDA.

The MLEM algorithm can be further divided into four steps, among which two are equivalent to (sparse) matrix-vector (SpMV) multiplication, respectively using the original matrix A and its transposition A^T. The main focus is then on the implementing and comparing of SpMV algorithms. In this thesis, two different SpMV algorithms are applied, namely the merge-based SpMV and the csr-vector SpMV. Apart from that, a novel parallel algorithm for matrix transposition is also covered in this thesis. The case when it is not possible to use transposed matrix is considered as well. Moreover the NCCL (NVIDIA Collective Communication Library) operations are also used as more than one GPUs may be applied to run the program.

In the last chapters, the performances of different SpMV and SpMV-T algorithms are investigated and compared. Their accuracy as well as the influence of number of GPUs are discussed in dedicated sections respectively. Lastly, the limitations of current implementation and some possible improvements are analyzed.

Contents

1 Introduction

The main purpose of this thesis is to transplant the MLEM algorithm, which is commonly used for PET reconstruction, to heterogeneous systems comprising multiple GPUs. The Positron Emission Tomography (PET) is a nuclear medicine imaging technique which is used to monitor body condition, to track blood flow or to prognosticate diseases [1]. To generate a visualization of the subject, a radioactive tracer which contains radioligands like fluorine-18 and carbon-11 is injected into the object or the body that will be examined. These radioligrands have solely short half life and will promptly undergo positron emission decay, annihilate with electrons and emit positrons. As a result, two 511 keV gamma photons are generated, traveling in opposite directions [2].

In order to track these photons, a PET scanner is used. The scanner comprises scalable detectors forming a ring and is placed around the examined body (objection) [3]. In Figure 1.1, a sample of PET scanner MADPET-II can be found. If the two emitted gamma photons are recorded by a pair of detectors in a short time interval, an annihilation event is then considered to take place somewhere along the line through which the two detectors are connected. This line is called the line of response (LOR) [1, 2] and it is possible to localize annihilation events with help of LORs. The more detected are applied, the higher accuracy and image resolution are achieved.

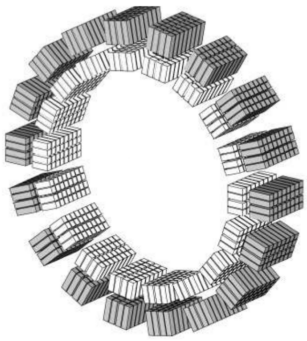


Figure 1.1: MADPET-II [3]

The space that is covered by the scanner is called field of view (FOV) [1, 2]. In order to reconstruct the image, the field of view is divided into a three-dimensional grid containing multiple grid cells, which are also called voxels. A two-dimensional matrix A of size $m \times n$ is then used to determine the properties of the scanner, where m represents the number of detector pairs and n represents number of voxels. Each element a_{ij} in the matrix measures the probability that a gamma photon emitted from inside voxel j can be detected by the pair of detectors i [3]. The detected events are recorded and formulated into a vector g of size n. The image reconstruction is equivalent to solving a linear equation system $Af = g$.

2 Related Work

How to accelerate iterative emission tomography image reconstruction, such as the Maximum Likelihood Expectation Maximization (MLEM) algorithm, has been an important research topic. Possible improvements have been investigated with respect to algorithms [4, 5], as well as to implementations [6]. This has also been a topic for Chair of Computer Architecture and Parallel Systems of the Department of Informatics for years. Previous work of the Chair includes implementations applying MPI, OpenMp as well functions in Intel MKL and NVIDIA CUSPARSE. Currently the work is oriented towards porting the MLEM algorithm to heterogeneous systems with distributed GPU clusters.

For SpMV algorithms on GPU, Bell, Garland and Merrill from NVIDIA are pioneers in this area. Both two SpMV algorithms applied in thesis (merge-based SpMV and csr-vector SpMV) originally come from their research [7–9]. Both algorithms are already compared against NVIDIA CUSPARSE functions in [7, 10] and in [11] respectively. In this bachelor's thesis, those two algorithms are compared to each other with respect to *Forward Projection* and *Backward Projection* in different cases.

The algorithm for matrix transposition applied in this thesis (ScanTrans) is developed by students in Virginia Tech [12, 13]. They have actually developed two novel parallel matrix transposition algorithms (ScanTrans and MergeTrans) and investigated the difference in performance of both algorithms against other existing algorithms as well as the matrix transposition function provided in Intel MKL library. As matrix transposition is not the main focus of this thesis, no comparison on performance among matrix transposition algorithms is carried out here.

The case when it is impossible to use transposed matrix for *Backward Projection* is also covered in this thesis. Instead of using any existing functions provided in CUSPARSE library, an algorithm developed last year by a master student Gupta of TUM is applied [11]. This algorithm is to some extent based on the csr-vector SpMV algorithm developed by Bell and Garland [9]. Gupta has already compared this algorithm to the algorithm provided in NVIDIA CUSPARSE library and an implementation in OpenMP, which will not be redone in this thesis. Instead, the performance of this algorithm is

compared to the two SpMV algorithms applied in this thesis, namely the case when the use of transposed matrix for *Backward Projection* is feasible.

The influence of pinned memory on performance does not belong to the focuses of this thesis and is hence not investigated. A similar analysis is carried out by the master student Gupta in [11].

One of the platforms used in this bachelor's thesis is NVIDIA DGX-1 V100. In comparison, the chosen platform for Gupta's master's thesis [11] is NVIDIA DGX-1 P100, which is the previous generation of DGX-1 V100. Because of some technical problems, the access to DGX-1 P100 is not granted this year. Therefore the performance in practice is not compared between these two platforms.

Some helper functions, including the functions for reading matrix file, converting matrix into CSR format and calculating norms, are provided by the Department of Informatics of TUM. The department has also provided other versions of MLEM implementations as well as a program to visualize matrices. The Figures 4.2 and 4.3 are generated with help of this program.

3 MLEM Algorithm

To solve the linear equation system $Af = g$, the Maximum Likelihood reconstruction (ML) using Expectation Maximization (EM) algorithm is used in this thesis. MLEM is one of the mostly used algorithms for PET image reconstruction and it works iteratively, starting with an initial grey image [3, 14]. The algorithm can be represented as follow (3.1):

$$f_j^{(q+1)} = \frac{f_j^q}{\sum_{l=1}^{m} a_{lj}} \times \sum_{i=1}^{m} \left(a_{ij} \left(\frac{g_i}{\sum_{k=1}^{n} a_{ik} f_k^q} \right) \right) \tag{3.1}$$

In this equation, f_j^q represents the j-th element of vector f in q-th iteration and $f_j^{(q+1)}$ is the corresponding value for next iteration. Before iterations start, two vectors are required: one is called *norm* which stores the summations of elements in each column (3.2); another is the initial grey image f^0 which the iterations will start with. Normally all elements in f^0 are set to the quotient of the summation of vector g and the summation of vector *norm* (3.3).

$$norm_j = \sum_{l=1}^{m} a_{lj} \tag{3.2}$$

$$f_k^0 = \frac{\sum_{i=1}^{m} g_i}{\sum_{j=1}^{n} norm_j} \quad \text{for all } k \text{ from 1 to } n \tag{3.3}$$

Each iteration of MLEM can be broken down to four smaller steps. The first step is called *Forward Projection*(FP). It is equivalent to a matrix-vector multiplication and results in a vector called $fwproj$ of size m (3.4).

$$fwproj_i = \sum_{k=1}^{n} a_{ik} f_k^q \tag{3.4}$$

The next step is a scaling step called *Correlation*. This step takes the result from *Forward Projection* as input and correlates it to the actual measurement (3.5). A vector *correl* of size m is generated in this step.

$$correl_i = \frac{g_i}{fwproj_i} \tag{3.5}$$

The following step is called *Backward Projection*(BP) and is equivalent to a matrix-vector multiplication using the transposed matrix A^T (3.6). The result of this step is a vector of size n called $update$.

$$update_j = \sum_{i=1}^{m} a_{ij} correl_i \tag{3.6}$$

The last step is a scaling step called $Update$. In this step, the image vector of current iteration f^q is scaled with the result from *Backward Projection* and the *norm* vector (3.7). This step generates the image vector $f^{(q+1)}$ as input for next iteration.

$$f_j^{(q+1)} = \frac{f_j^q}{norm_j} \times update_j \tag{3.7}$$

In conclusion, *Correlation* and $Update$ are simply scaling steps, whereas *Forward Projection* and *Backward Projection* are matrix-vector multiplications. Compared to scaling, matrix-vector multiplication requires considerably more resources and is also more difficult to implement. Hence it is reasonable to develop some dedicated kernels for matrix-vector multiplication, which can be invoked by both *Forward projection* and *Backwardprojection*. However, for *backwardprojection*, the transposition of matrix A is required, which costs not only additional computation, but also more memory space. Therefore it is not always possible to carry out *Backward Projection* using transposed matrix because of the limited memory space of platforms, which will be discussed later.

4 Matrices

For any given voxel, the dectector pairs in which the detectors are closely 180 degrees apart from each other are almost the only candidates to record non zero entries (nnz). Therefore matrices used in PET reconstruction are commonly very sparse (e.g. more than 99% of the entries are zero). For dense matrices it is reasonable to represent the matrices in two-dimensional arrays. However for sparse matrices, full-sized two-dimensional arrays are not worthy any more since most of the values in the matrices are zero. Hence a special matrix format CSR is used in this bachelor's thesis in order to represent matrices more sophisticatedly and to save memory space.

4.1 CSR Format

The Compressed Sparse Row (CSR) format is one efficient way to store sparse matrices. The CSR format is a row major format and is generated in left-to-right top-to-bottom order. Instead of saving matrices in single two-dimensional array, CSR format represents matrices with three one-dimensional arrays: *csr_Rows*, *csr_Cols* and *csr_Vals* (this nomination of these three arrays is consistent in this thesis for convenience, as they are commonly named differently in different articles).

The first array *csr_Rows* keeps information for how many non zeros (nnzs) are recorded in each row of the matrix. Its length is number of rows + 1. The first entry $csr\_Rows[0]$ in the array is always 0. For any row index i from 1 to number of rows, the value $csr\_Rows[i]$ represents the number of nnzs accumulated from the beginning of first row to the end of current row. Furthermore the value $csr\_Rows[i+1] - csr\_Rows[i]$ depicts the number of nnzs in the given row i. Obviously the values in *csr_Rows* array are in non-decreasing order. The values are not necessarily always increasing as there might be rows in which no nnzs are recorded so that $csr\_Rows[i+1] = csr\_Rows[i]$.

The second array *csr_Cols* stores the column index of each nnz and thus has length of number of nnzs. Similar to this, the last array *csr_Vals* stores the values of nnzs and its size is also number of nnzs. Unlike the non-decreasing order in *csr_Rows* array, there is no specific order for entries in *csr_Cols* array or *csr_Vals* array since the column

indices and values of nnzs are not necessarily in increasing or decreasing order.

As aforementioned, the three arrays in CSR format are generated in left-to-right top-to-bottom pattern. One sample of how to create the CSR format for a given matrix is shown in Figure 4.1.

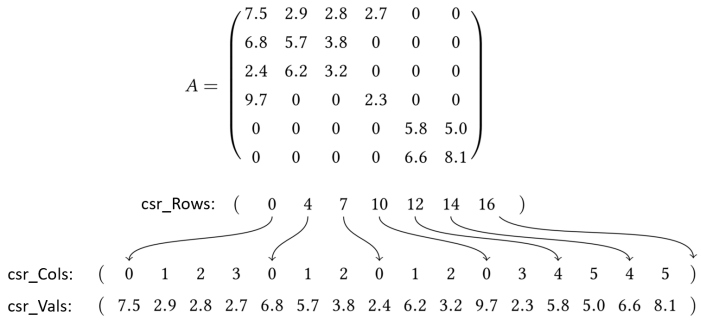


Figure 4.1: CSR Format [15]

With CSR format it is very simple to carry out matrix-vector multiplication. For any given row i to be multiplied with the vector g, the number of nnzs in this row is given by $csr\_Rows[i+1] - csr\_Rows[i]$. Using an index j in range from $csr\_Rows[i]$ to $csr\_Rows[i+1]$, the value of the j-th nnz in the i-th row is then $csr\_Vals[j]$ and its column index is $csr\_Cols[j]$. To calculate the i-th row multiplied with vector g is then equivalent to summation over j in range from $csr\_Rows[i]$ to $csr\_Rows[i+1]$ on $csr\_Vals[j] \times g[csr\_Cols[j]]$. This process is illustrated with Sample Code 4.1.

```
1 // matrix−vector multiplication using CSR format
2 for ( int i = 0; i < rows; i++ ){
3     result[i] = 0;
4     for ( int j = csr_Rows[i]; j < csr_Rows[j]; j++ ){
5         result[i] += csr_Vals[j] ∗ g[csr_Cols[j]];
6     }
7 }
```

Code 4.1: Matrix-Vector Multiplication using CSR format

The biggest advantage that can be taken of using CSR format is the save of memory space. Assuming that a square matrix of size $n \times n$ is used and its sparsity is 99%,

which means that only 1% of entries in the matrix are nnzs. Each nnz in the matrix is a 32bit float value, and number of rows/columns can be kept in 32bit integers. Storing the matrix in naive two-dimensional array will cost $4n^2$ Bytes. In contrast to this, storing the matrix in CSR format will result in three arrays: *csr_Rows* of length $n+1$ costing $4n+4$ Bytes, *csr_Cols* of length $0.01n^2$ costing $0.04n^2$ Bytes and *csr_Vals* of length $0.01n^2$ costing $0.04n^2$ Bytes. In total only $0.08n^2+4n+4$ Bytes are required. This is remarkably efficient in comparison to the memory cost of $4n^2$ in naive approach.

4.2 Matrices used

For this bachelor's thesis, three matrices of different sizes are used. Each matrix is stored in a single file and has to be loaded into CSR format. During developing phase, in order to carry out fast tests for programs, the smallest matrix is used. This matrix is actually not sparse and details about matrix is shown in Table 4.1.

Parameter	**Value**
Size	105,447 KB
Rows	65536
Columns	1024
NNZs (non zeros)	13,431,560
Density	20.015%
Max NNZ	1.0
Min NNZ	1.0
Avg NNZ	1.0
Max row NNZs	1024
Min row NNZs	1
Avg row NNZs	204

Table 4.1: Matrix 1

During testing phase, a matrix containing real values which specify characters of a MADPET-II scanner is used. This matrix is significantly larger than the matrix used in developing phase and is extremely sparse. The Table 4.2 lists details of this matrix. Additionally, a visualization of this matrix is shown in Figure 4.2, which is generated with the help of a visualization program developed by the Department of Informatrics.

Furthermore in order to test the performance of programs in extreme cases, namely using extraordinarily large matrices, a third matrix that consumes approximately 100 GB memory space is used and its details are list in Table 4.3. Figure 4.3 shows the

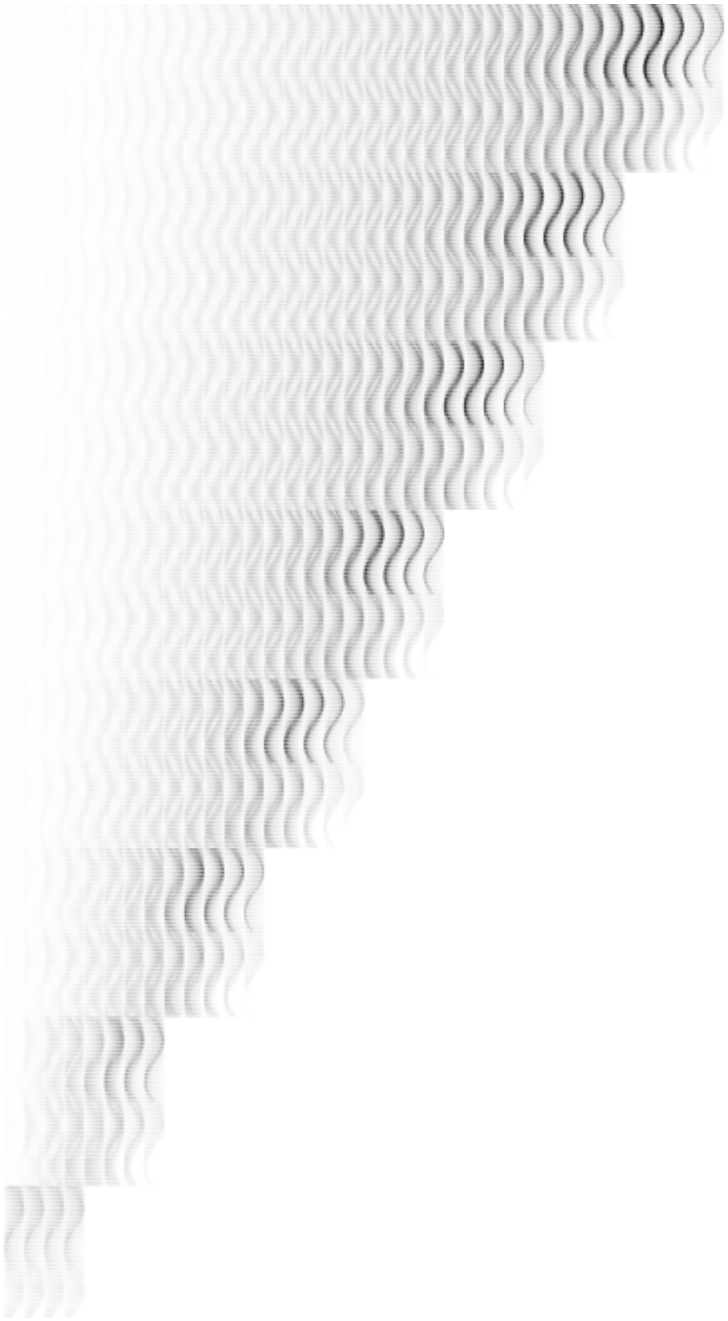


Figure 4.2: Visualization of Matrix 2

Parameter	Value
Size	12,537,704 KB
Rows	1,327,104
Columns	784,000
NNZs (non zeros)	1,603,498,863
Density	0.154%
Max NNZ	$8.90422e^{-5}$
Min NNZ	$5.50416e^{-24}$
Avg NNZ	$4.33555e^{-7}$
Max row NNZs	6537
Min row NNZs	1
Avg row NNZs	1208

Table 4.2: Matrix 2

distribution of nnzs in this matrix. Compared to the second matrix which stores real data, this large matrix is artificially generated and contains virtual data. On the platform DGX-1 V100, although the host RAM (512 GB) is large enough to load this matrix, the memory size of each NVIDIA V100 GPU (16 GB) is not as large as the host RAM. In order to run the kernel successfully, this matrix has to be partitioned uniformly into eight pieces and each GPU holds one piece of them. It is not possible to use transposed matrix here as GPUs have no enough memory left to store its transposition.

Parameter	Value
Size	about 100 GB
Rows	5,308,416
Columns	2,592,000
NNZs (non zeros)	12,312,716,683
Density	0.0895%
Max NNZ	$8.35204e^{-5}$
Min NNZ	$3.72351e^{-25}$
Avg NNZ	$8.34061e^{-8}$
Max row NNZs	14681
Min row NNZs	1
Avg row NNZs	2319

Table 4.3: Matrix 3

Figure 4.3: Visualization of Matrix 3

5 Sparse Matrix-Vector Multiplication Algorithms

As aforementioned in Chapter 1 (Introduction), the PET image reconstruction is not different from solving a linear equation system $Af = g$. To solve this equation, an iterative algorithm MLEM is used in this bachelor's thesis. As detailed in Chapter 3 (MLEM Algorithm), each iteration in MLEM can be divided into four small steps, among which two steps are equivalent to matrix-vector multiplication. Since characteristic matrices of PET scanner are normally very sparse, the whole problem traces back to Sparse Matrix-Vector (SpMV) multiplication. The key to transplant MLEM algorithm to heterogeneous systems lies in the development of kernels which process SpMV efficiently on NVIDIA GPUs using CUDA.

Traditional matrix-vector multiplication approach using CUDA is very brutal: each thread copes with one row in matrix multiplied with target vector and this process is elaborated in Sample Code 4.1. This implementation works well when the matrix is well-shaped (not extremely sparse). However, in PET image reconstruction the matrices are commonly very sparse and imbalanced. As is shown in Table 4.2, some rows/columns contain solely 1 nnz, whereas others may have over 6500 nnzs. The sparsity of matrices results in the imbalance of workload between threads. While some threads have entirely no work as the rows assigned to them contain no nnzs, some threads may have to deal with thousands of floating-point calculations. Hence it is crucial to develop some kernels that can process SpMV more efficiently on NVIDIA GPUs using CUDA.

In this bachelor's thesis, two SpMV approaches are implemented. The first one which is called merge-based SpMV [7, 8, 10] is developed by researchers from NVIDIA. This approach attaches great importance to workload balance among threads. Another approach named as csr-vector SpMV [9] is also proposed by NVIDIA researchers in 2008. Compared to the first SpMV approach, it focuses more on memory access pattern and improves the distribution of GPU resource significantly.

5.1 Merge-based SpMV

As aforementioned, the PET scanner characteristic matrices are in most cases very sparse. For example in the second testing matrix (Table 4.2), the sparsity is 99.84%. Some rows in the matrix contain solely 1 nnz, while some other rows have more than 6500 nnzs. Using traditional matrix-vector multiplication approach, where each thread deals with the multiplication between one row and target vector, will definitely lead to imbalance among threads. The imbalance in such a case can be illustrated by Figure 5.1. As is shown in the Figure, the last row contains double number of nnzs than the first and the third rows, which result in double computation time. On the contrary, there are no nnzs in the second row, thus the second thread has actually no work during function run and is always waiting for other threads completing their work.

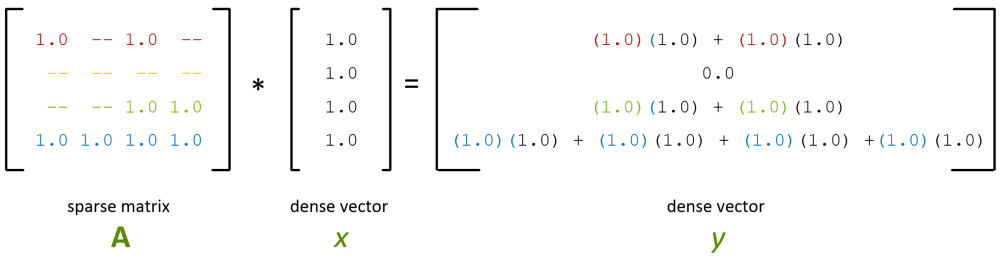


Figure 5.1: Imbalance in SpMV [10]

5.1.1 Concept

The imbalance of traditional approach results from the row-based matrix partition. For example in Figure 5.2, where the same matrix is considered as in Figure 5.1, the matrix is decomposed into four parts according to rows by default. Intuitively each thread copes with one part (one row) times with target vector. As nnzs are not equally distributed among parts, imbalance will be raised during computation.

A better solution could be partition the matrix according to nnz distribution instead of to rows. Considering the same matrix as in Figures 5.1 and 5.2. Assuming the matrix is desired to be partitioned into four parts, each part should then contain exactly two nnzs. An intuitive solution to such matrix partition is shown in Figure 5.3. Here it is significant that the last row of the matrix is divided into two parts and two threads

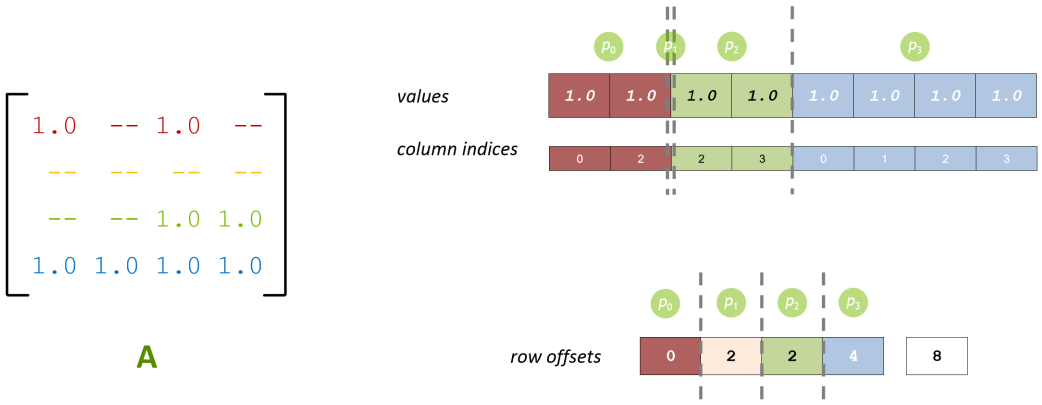


Figure 5.2: Row-based Partition [10]

should be used to work with this row.

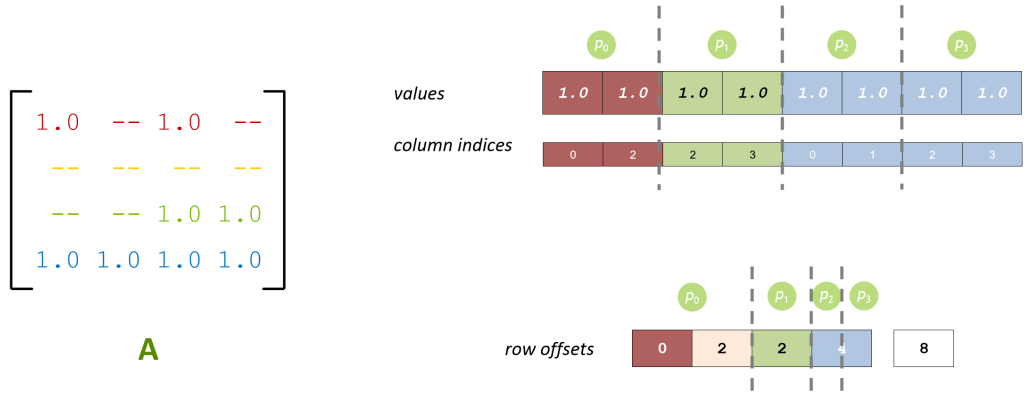


Figure 5.3: NNZ-based Partition [10]

Although the imbalance in accessing column vector and value vector is solved with help of nnz-based partition, the imbalance in accessing row vector is raised. As is shown in 5.3, the first thread has to read two elements in the row vector, whereas the third and the fourth thread has to read half element from the row vector. This imbalance will be exacerbated when the matrix is larger and sparser. What is even worse is than when more than one threads work on the same row, atomic operations (atomicAdd) have to be applied instead of simple arithmetic operations, which will influence the performance of kernel function negatively.

In order to solve this problem, the merge-based partition [7, 8] can be applied. In this approach, the *elements*, which be taken into calculation by a thread, are re-defined. Here each *element* can be not only a nnz, but also a row switch. For instance, if some thread needs to go through 4998 rows, then these 4998 row switches are considered as 4998 *elements* as well, which will be processed by the thread. A graphical illustration of this merge-based concept is shown in Figure 5.4. In this example, the matrix is partitioned into three parts. The first and the second parts both contain four *elements*, among which two are row switches and the other two are nnzs, while the last part contains solely nnzs and no row switch.

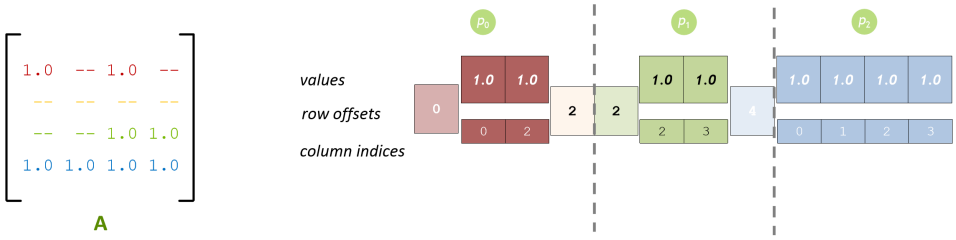


Figure 5.4: Merge-based Partition [10]

5.1.2 Implementation

The implementation of merge-based matrix partition can be divided into two steps: firstly, the total amount of *elements* (number of rows + number of nnzs) in the matrix has to be calculated. Then the user should specify the size of each partition. The number of partitions is then simply amount of elements over size of partition. Then the start point of each thread (start indices in the three arrays in CSR format) can be determined by a so-called 2D merge-path search: at beginning, a 2D grid is generated, in which each vertical line represents a row in matrix, and each horizontal line represents a nnz in matrix. The start position of the k-th thread is located along the k-th diagonal in the grid (along the k-th diagonal, all coordinates (i, j) satisfy $i + j = k$). Further the exact coordinate of start position on can be found using binary search, which is the first coordinate satisfying $csr\_Rows[i] > j - 1$ [7, 10]. The start indices of the correspond thread are then i in csr_Rows array and j in csr_Cols as well as csr_Vals. This process can be demonstrated with Figure 5.5 and Sample Code 5.1. Here the same matrix as in

Figure 5.1 is considered. The parameter *secSize* determines the size of partition.

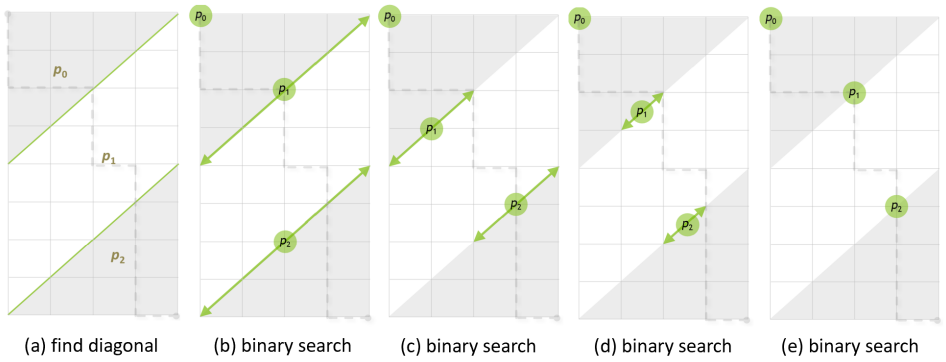


Figure 5.5: Merge-path Search [10]

```
1  __device__ void merge_based_start(int *csr_Rows, int *csr_Cols, float *csr_Vals,
       float *x, float *result, int secSize, int rows, int nnzs) {
2
3      int lefti = 0;
4      int righti = rows;
5      int nexti = righti / 2;
6      int index = blockIdx.x * blockDim.x + threadIdx.x;
7      int start = index * secSize;
8      int nextj = start - nexti;
9      int i = 0, j = start;
10
11     while (i != nexti) {
12       i = nexti;
13       j = nextj;
14
15       // find the first coordinate (i, j) that r[i + 1] > j - 1
16       if (csr_Rows[i + 1] > j - 1)
17         righti = i;
18       else
19         lefti = i + 1;
20
21       nexti = (lefti + righti) / 2;
22       nextj = start - nexti;
```

```
23      }
24
25      // next step: merge_based_work
26  }
```

Code 5.1: Merge-Path Search

Secondly, each thread needs to process floating-point calculations with the *elements* in the corresponding partitions. Every time when the thread processes a nnz, it goes downwards in the 2D grid. On the contrary, when the thread has consumed all nnzs in a row and needs to switch to next row, it goes right in the grid. The thread finishes all its work when it reaches the start position of next partition, which depicts the start indices for next thread. This step can be illustrated with Figure 5.6 and Sample Code 5.2. Here it is important to use atomic operations (*atomicAdd*) instead of simple arithmetic operations because each row may be processed by multiple threads.

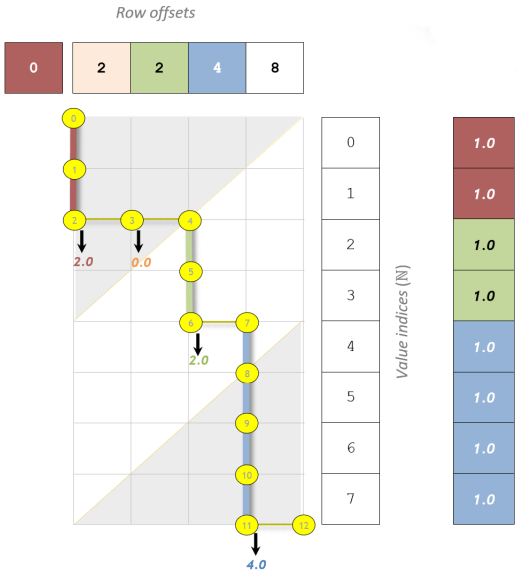


Figure 5.6: Complete Merge-Path [10]

```
1   __device__ void merge_based_work(int *csr_Rows, int *csr_Cols, float *csr_Vals,
        float *x, float *result, int secSize, int rows, int nnzs, int i, int j) {
2
3       int end = i + j + secSize;
4       if (end > nnzs + rows)
```

```
5       end = nnzs + rows;
6     float rowTimesVector = 0.0f;
7     while (i + j < end) {
8       if (csr_Rows[i + 1] > j) {
9         rowTimesVector += csr_Vals[j] * x[csr_Cols[j]];
10        j++;
11      }
12      else {
13        atomicAdd(result + i, rowTimesVector);
14        i++;
15        rowTimesVector = 0.0f;
16      }
17    }
18    if (rowTimesVector != 0.0f)
19      atomicAdd(result + i, rowTimesVector);
20  }
```

Code 5.2: merge-based SpMV

5.1.3 Advantages and Drawbacks

Merge-based SpMV is a stable solution to sparse matrix-vector multiplication in most cases and can guarantee a well-balanced workload distribution among threads and thus can improve the performance to some extent. However, merge-based SpMV does not necessarily provide higher performance in comparison to brutal matrix-vector multiplication or SpMV algorithms provided by other libraries. The comparison between merge-based SpMV and SpMV functions provided in CUSPARSE library are already carried out in [7, 10] and the result of comparison is shown in Figure 5.7. The main reason for why merge-based SpMV does not necessarily give better performance lies in the use of atomic operations (atomicAdd) since multiple threads may work on the same row simultaneously.

Another drawback of merge-based SpMV is the cross-row partition of matrix. Merge-base partition may create sections that span more than one rows, which results in a m:n relation between threads and rows: each thread may cope with multiple rows, and each row may be processed by multiplie threads. In such a case, it is almost impossible to make use of shared memory to improve performance.

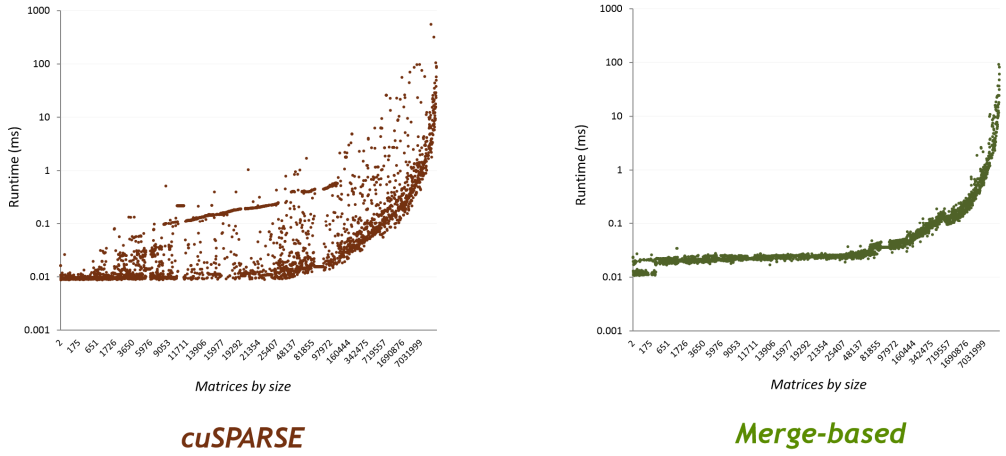


Figure 5.7: Comparing Merge-based SpMV and CUSPARSE Function [7, 10]

5.2 CSR-vector SpMV

Another SpMV approach called csr-vector was proposed by NVIDIA researchers in 2008 [9]. Compared to merge-based SpMV, csr-vector SpMV does not guarantee well-balanced workload among threads or blocks. Instead, it focuses mainly on coalesced memory access and use of shared memory.

5.2.1 Concept

For CUDA programs, a bunch of consecutive threads are managed in a warp (nowadays normally 32 threads per warp). Since resource distribution like dispatching and addressing is based on warps in CUDA, coalesced memory access can be then achieved if consecutive threads inside a warp access contiguous memory positions. The Figure 5.8 provides a graphical demonstration to this memory access pattern.

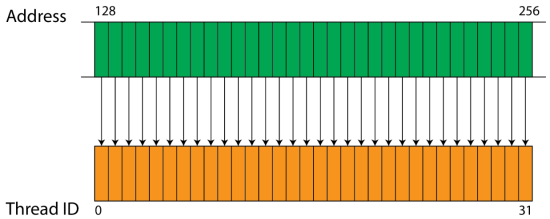


Figure 5.8: Coalesced Memory Access [16]

The basic idea of csr-vector SpMV [9] is to use one warp to deal with the multiplication of one row in matrix and target vector. Threads in the warp are indexed from 0 to 31. Thread 0 will process the multiplications of nnzs indexed with 0, 32, 64 ... and the corresponding values in target vector. Thread 1 will process nnzs 1, 33, 65 ... and all other threads follow the same pattern. Since consecutive threads always access contiguous positions in memory, performance can be considerably improved.

Another improvement of csr-vector SpMV is the use of shared memory [9]. Shared memory is a memory space allocated per block (nowadays normally 1024 threads/32 warps per full block) and access to shared memory is significantly faster than to global memory. As mentioned above, csr-vector SpMV uses one warp to deal with one row in matrix multiplied with target vector. After the computation, all temporary results are saved in the 32 threads in the warp. To get a complete result of the row-vector multiplication, the 32 results need to be added up and then saved in result vector. Simple arithmetic operations cannot be used here since multiple threads work on the same row, while atomic operations can be used but not efficient. In csr-vector SpMV, a novel approach is used to solve this problem: all threads save their temporary results in an array in shard memory, and the first thread of each warp adds up all results saved by the following 31 threads.

5.2.2 Implementation

The Sample Code 5.3 provides an implementation of csr-vector SpMV.

```
1  __device__ void mat_vec_mul_csr_vector (int *csr_Rows, int *csr_Cols, float *
       csr_Vals, float *x, float *result, int rows){
2
3      __shared__ float values[1024];
4      int WARP_SIZE = 32;
5      int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
6      int thread_lane = threadIdx.x & (WARP_SIZE-1);
7      int warp_id = thread_id / WARP_SIZE;
8      int num_warps = (blockDim.x / WARP_SIZE) * gridDim.x;
9
10     // one warp per row
11     for (int row = warp_id; row < rows ; row += num_warps){
12       int row_start = csr_Rows[row];
13       int row_end = csr_Rows[row + 1];
14
```

```
15        values[threadIdx.x] = 0.0f;
16
17        for (int jj = row_start + thread_lane ; jj < row_end ; jj += WARP_SIZE)
18          values[threadIdx.x] += csr_Vals[jj] * x[csr_Cols[jj]];
19
20        // first thread writes the result
21        if (thread_lane == 0){
22          for (int i = 1 ; i < WARP_SIZE ; i++)
23            values[threadIdx.x] += values[threadIdx.x + i];
24
25          atomicAdd(result + row, values[threadIdx.x]);
26        }
27
28        __syncthreads();
29      }
30  }
```

Code 5.3: csr-vector SpMV [9]

5.2.3 Advantages and Drawbacks

The biggest advantages of this approach is that the coalesced memory access pattern is achieved, which increases the performance tremendously. Apart from that, shared memory is also used for better performance. Furthermore, since threads within the same warp are always serviced simultaneously, there is no need for additional barriers (*__syncthreads*) inside warps.

However, similar to naive matrix-vector multiplication, the distribution of workload among warps is normally not well-balanced when using sparse matrices. Hence the time cost of csr-vector SpMV depends largely on the sparsity and the shape of matrices, which is also illustrated in [11].

6 Scan-based Sparse Matrix Transposition

The third step in MLEM algorithm, namely *Backward Projection*, is equivalent to a sparse matrix-vector multiplication, under the condition that the use of transposed matrix A^T is feasible.

A number of functions are provided in different libraries for transposition of matrices in CSR format. However, these functions normally have their own limitations. For instance, in NVIDIA CUSPARSE library there are functions like *cusparseScsr2csc* [17] to transform a matrix in CSR format into CSC (Compressed Sparse Column) format, which is not different from the transposed matrix in CSR format. These functions run on GPU and data have to be copied from host memory to GPU memory before function run. However, since GPU memory is in comparison to host memory strongly limited, it is impossible to transpose a large matrix using these functions, unless the matrix is firstly partitioned and each GPU operates only on the transposition of a part of the matrix. For example, on the testing platform with NIVIDIA Quadro P6000 (Table 11.2), whose memory size is 24 GB, it is impossible to transpose the second testing matrix (Table 4.2), even if the matrix is only about 12.8 GB. This is because during function run, additional memory space is required for local variables and temporary arrays. Apart from the NIVIDIA CUSPARSE library, other libraries such as Intel MLK (Math Kernel Library) and AMD CML (Core Math Library) also provide functions for matrix transposition, which are however dependent on platforms.

For this bachelor's thesis, a novel parallel matrix transposition algorithm called *ScanTrans* [12, 13] is used. This parallel algorithm is proposed by students in Virginia Tech, which runs on CPU and applies OpenMP. There is also another parallel matrix transposition algorithm *MergeTrans* suggested in [12, 13], which is according to the comparison of performance not as good as *ScanTrans* for large matrices.

The basic concept of *ScanTrans* is as follow [12]: firstly the matrix in CSR format is partitioned evenly among threads according to number of nnzs, then numbers of different column indices as well as the relative position of each nnz in corresponding column are counted by each thread. Finally the absolute offset of each nnz is calculated and stored in target arrays. Here for convenience the arrays of CSR format are renamed:

csr_Rows as *csrRowPtr*, *csr_Cols* as *csrColIdx*, *csr_Vals* as *csrVal*. And the arrays of the transposed matrix in CSR format are named as *cscColPtr*, *cscRowIdx* and *cscVal*.

At beginning, the *csrRowPtr* array is extended into a *csrRowIdx* array, which stores the row index of each nnz and of size *nnzs*. Then two auxiliary arrays are allocated: a two-dimensionaly array called *inter* of size $(nthreads + 1) * n$, and a one-dimensional array named *intra* of size *nnzs*. In *inter*, the *i*-th row stores the number of different column indices scanned by the thread $i - 1$. In *intra*, each element represents the relative offset of the nnz in the corresponding column in the thread. After the histogram *inter* is generated, column-wise scannings ($vertical_scan$) through the histogram are applied, and vertical accumulations as well as prefix sums are calculated. The *cscColPtr* array, which is equivalent to the *csr_Rows* array of the transposed matrix in CSR format, can be generated with the help of those prefix sums. Finally, the absolute offsets of nnzs in *cscRowIdx* and *cscVal* can be calculated as follow: (6.1), where $csrOffset$ represents the offset of the nnz in *csrColIdx* and *csrVal* [12].

$$offset = cscColPtr[colIdx] + inter[threadID][colIdx] + intra[csrOffset] \tag{6.1}$$

For instance, the matrix A in Figure 6.1 is transposed using *ScanTrans*. Here four threads are used and each thread deals with four nnzs: thread 0 works for a, b, c, d, thread 1 works for e, f, g, h and this pattern follows for threads 2 and 3. During horizontal scanning, thread 0 records the number of different column indices for nnzs: 1 for a, 3 for b, 0 for c and 1 for d. Since the column index 1 appears two times, $inter[1][1] = 2$, while $inter[1][0] = 1$ and $inter[1][3] = 1$ because indices 0 and 3 only appear 1 time. Thread 0 also records the relative position of each nnz during horizontal scanning from the point of view of thread 0 itself: as a, b, c are the first nnzs of their columns, $intra[0], intra[1], intra[2]$ are set to 0. Compared to this, d is already the second nnz in its column as a is the first nnz in the same column, $intra[3]$ is set to its relative position in this column, namely 1. A further example is for the nnz h. Thread 1 deals with nnzs e, f, g, h. The column indices are 2 for e, g and 3 for f, h, hence $inter[2][2]$ and $inter[2][3]$ are both set to 2. However, while calculating the offset for h, the values in $inter[1]$ are considered, not the values in $inter[2]$. Accumulated with values in $inter[0]$, which are all 0, the values in $inter[1]$ remain unchanged. h is the second nnz in its column as f is the first one, thus $intra[7]$ is set to 1. The offset of h in *cscRowIdx* and *cscVal* can be then caulculated as $cscColPtr[3] + inter[1][3] + intra[7] = 7 + 1 + 1 = 9$, as h is in column 3, servied by thread 1 and its offset in CSR format for *csrColIdex* is 7.

The following Pseudo Code 6.1 gives a basic sample for implementation of *ScanTrans*. This Pseudo Code can be easily customized for different data types.

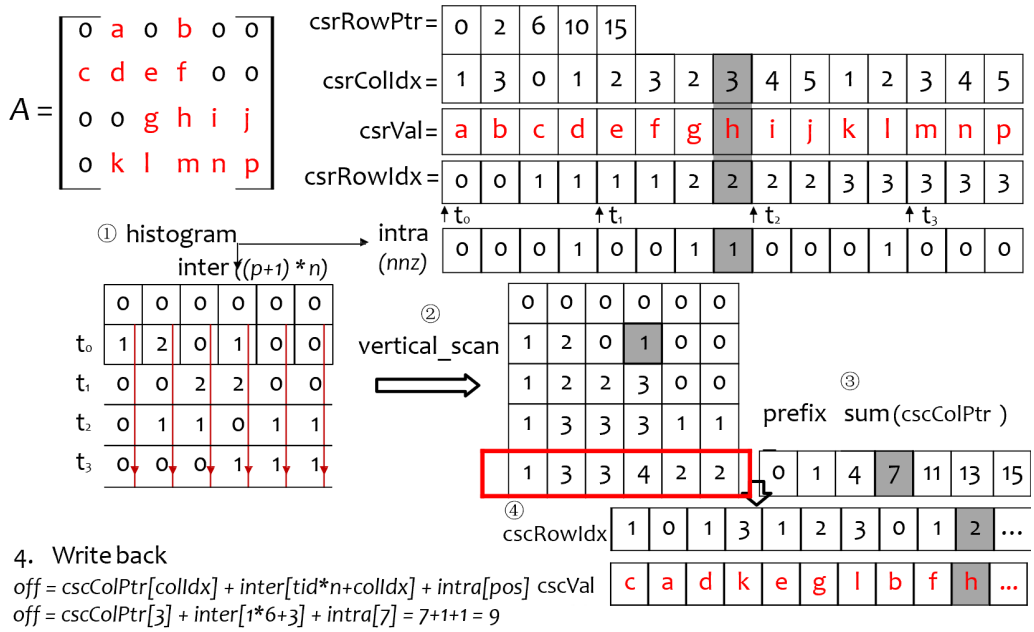


Figure 6.1: ScanTrans [12, 13]

```
1  Function ScanTrans(m, n, nnz, csrRowPtr, csrColIdx, csrVal, cscColPtr, cscRowIdx,
       cscVal)
2
3      // construct auxiliary data arrays
4      *intra = new int[nnz]();
5      *inter = new int[(nthreads + 1) * n]();
6      *csrRowIdx = new int[nnz]();
7      #pragma omp parallel for schedule(dynamic)
8      for i = 0; i < m; i++ do
9          for j = csrRowPtr[i]; j < csrRowPtr[i+1]; j++ do
10             csrRowIdx[j] = i;
11
12     #pragma omp parallel
13     // partition nnz evenly on threads, get start in csrColIdx and len for each thread
14     for i = 0; i < len; i++ do
15         intra[start + i] = inter[(tid + 1)*n + csrColIdx[start + i]]++;
16
```

```
17    // vertical scan
18    #pragma omp parallel for schedule(dynamic)
19    for i = 0; i < n; i++ do
20       for j = 1; j < nthread + 1; j++ do
21          inter[i + n∗j] += inter[i + n∗(j − 1)];
22
23    #pragma omp parallel for schedule(dynamic)
24    for i = 0; i < n; i++ do
25       cscColPtr[i + 1] = inter[n∗nthread + i];
26    prefix sum(cscColPtr,n + 1);
27
28    #pragma omp parallel
29    for i = 0; i < len; i++ do
30       loc=cscColPtr[csrColIdx[start+i]]+inter[tid][csrColIdx[start+i]]+intra[start+i];
31       cscRowIdx[loc] = csrRowIdx[start + i];
32       cscVal[loc] = csrVal[start + i];
33
34    // free intra,inter,csrRowIdx
35    return;
```

Code 6.1: ScanTrans Pseudo Code [12]

ScanTrans is an efficient algorithm as atomic operations can be avoided because two rounds of scanning are processed separately on auxiliary arrays. Apart from this, scanning operations can be implemented in SIMD (Single Instruction Multiple Data) pattern. Considering a matrix A of size $m \times n$ with number of nnzs as nnz, the complexity of *ScanTrans* is then O($m + n + nnz$). However, when writing back values to $cscRowIdx$ and $cscVal$, the memory is accessed randomly. And an amount of memory space is required on host memory as three auxiliary arrays are allocated.

7 Backward Projection using no transposed matrix

Ideally, under the condition that the use of transposed matrix is feasible, *Backward Projection* is not different from a sparse matrix-vector multiplication $A^T correl = update$, where *correl* is the result from last step *Correlation* and $update$ is the result of *Backward Projection* as well as the input for next step $Update$.

However in practice, because of the limited memory size on GPUs, it may be impossible to use transposed matrix. For instance, on the testing platform with NVIDIA Quadro P6000 (Table 11.2), of which the GPU memory size is 24 GB, it is doubtless that the transposition of matrix cannot be stored if the second testing matrix (Table 4.2) is used. The second testing matrix is about 12.8 GB, and storing the transposition of it costs approximately another 12.8 GB, which is obviously impossible on this plaform. A further example can be using the third testing matrix (Table 4.3) on the platform DGX-1 V100 (Table 11.4). Although DGX-1 has eight NVIDIA Tesla V100 with total memory size of $16 * 8 = 128$ GB, it is still impossible to store the transposition of a matrix costing 100 GB.

Therefore, an efficient implementation of *Backward Projection* using no transposed matrix is needed in extreme cases when the matrix is considerably large. In this thesis, a novel approach developed by a master student of TUM last year is applied [11]. The basic concept of it is similar to csr-vector SpMV (5.3), namely one warp deals with one row instead of one thread deals with one row in naive SpMV. The Sample Code of this approach its is as follow (7.1):

```
1  __device__ void trans_mat_vec_mul_warp(int *csr_Rows, int *csr_Cols, float *
       csr_Vals, float *x, float *result, int rows){
2
3      int WARP_SIZE = 32;
4      int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
5      int thread_lane = threadIdx.x & (WARP_SIZE-1);
6      int warp_id = thread_id / WARP_SIZE;
```

```
7       int num_warps = (blockDim.x / WARP_SIZE) * gridDim.x;
8
9       for(int row = warp_id; row < rows ; row += num_warps){
10        int row_start = csr_Rows[row];
11        int row_end   = csr_Rows[row + 1];
12        for (int i= row_start + thread_lane; i < row_end; i += WARP_SIZE)
13          atomicAdd(&result[csr_Cols[i]], csr_Vals[i] * x[row]);
14      }
15  }
```

Code 7.1: BP using No Transposed Matrix [11]

Intuitively *Backward Projection* using no transposed matrix is inferior to the implementation using transposed matrix on performance. The comparison will be discussed later.

8 NCCL Operations

As matrix can be so large that it cannot be stored on any single GPU, the matrix is partitioned on rows evenly into parts according to nnzs and each GPU works for one part of the matrix. This is possible as matrix-vector multiplication is row-based and the multiplication of one row with target vector is independent to other rows. However, this results in that each GPU only holds part of the whole result vector, as it holds only part of the matrix, which can be explained with Figure 8.1.

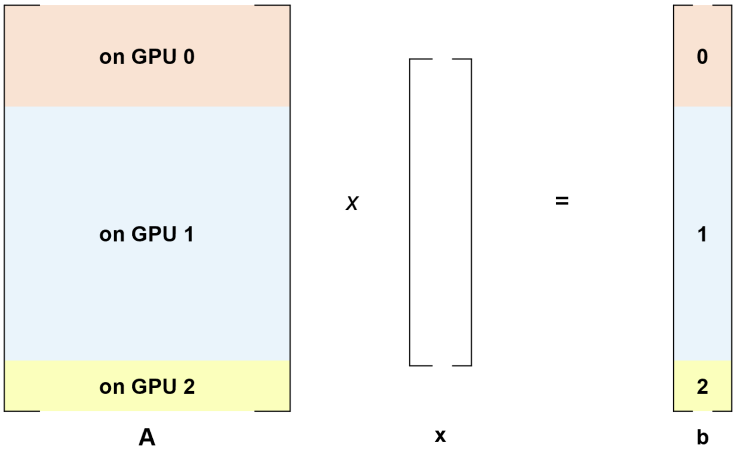


Figure 8.1: Part of Matrix, Part of Result

To get a complete result vector, all result parts in different GPUs have to be concatenated. The NVIDIA Collective Communications Library (NCCL, [18]) provides multiple functions for inter-GPU operations, which can be used to aggregate result parts. Ideally, the function *ncclAllGather* can be used to concatenate arrays stored on a series of GPUs. The Figure 8.2 illustrates this process graphically. However, *ncclAllGather* only works under the condition that the sizes of arrays on different GPUs have to be the same. As in this bachelor's thesis, in order to balance workload among GPUs, matrix is partitioned on the basis of not rows rather of nnzs. Therefore each partition may have different number of rows from others and hence the sizes of result vectors on different GPUs are unequal, which makes it impossible to apply *ncclAllGather*.

Figure 8.2: ncclAllGather [18]

As a substitute for *ncclAllGather*, another operation *ncclAllReduce* provided by NCCL is applied to aggregate shattered result vectors over GPUs. The *ncclAllReduce* operation performs reductions on data across devices and writes back the result into receive buffers. Till now there are four different types of reduction supported by *ncclAllReduce*: *ncclSum* (sum +), *ncclProd* (product *), *ncclMin* (find minimum) and *ncclMax* (find maximum). A graphic demonstration of *ncclAllReduce* is given in Figure 8.3. Compared to *ncclAllGather* which is influenced by the order of GPUs, *ncclAllReduce* is rank-agnostic and any reordering of GPUs has no impact on the outcome.

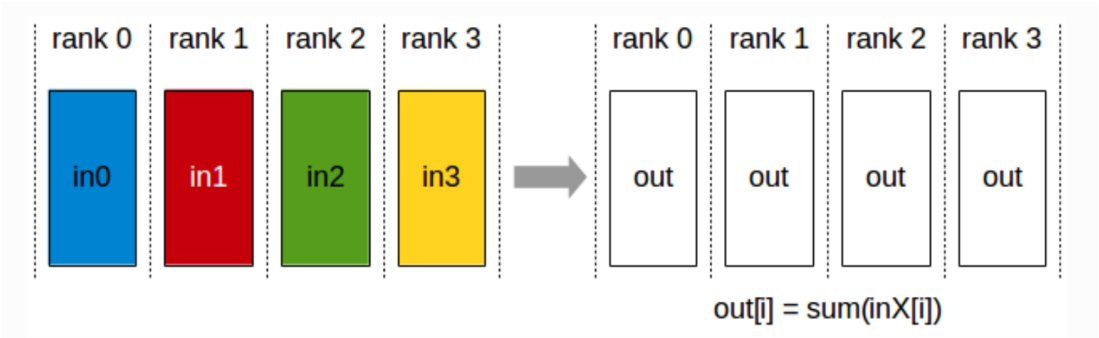


Figure 8.3: ncclAllReduce [18]

In order to apply *ncclAllReduce* for this bachelor's thesis, each GPU is designed to store its results in a full-sized result vector on itself, starting at corresponding position. For example, assuming that the full matrix has 10 rows, and is partitioned into three parts: the first GPUs gets the first 2 rows, the second gets the rows 3 to 9, and the third gets the last row. Then each GPU has a discrete result vector of size 10 onitself, and writes its results into this vector. The first GPU starts at offset 0 and stores only 2

values, the second GPU starts at offset 2 and writes 7 values, and the last GPU stores solely 1 value at position 9. All unwritten positions remain 0. After that, the three result vectors, with each GPU holding one result vector, are aggregated by applying *ncclAllReduce* using *ncclSum* operation.

As for difference in performance between *ncclAllGather* and *ncclAllReduce*, NVIDIA has provided no explanation officially. However it is reasonable to suppose that in general *ncclAllGather* surpasses *ncclAllReduce* on performance, as *ncclAllGather* is concatenation whereas *ncclAllReduce* still requires arithmetic operations. Since the comparison of these two functions is actually unimportant, it is not considered in this bachelor's thesis.

9 Experiments and Results

In this chapter, the details of experiments and their results are given. Furthermore, comparisons of algorithms are carried out and are represented with help of diagrams.

9.1 Details of Experiments

Experiments are carried out on two platforms (Chapter 11). On the platform AMD Ryzen Threadripper 2990WX + NVIDIA Quadro P6000, because only one GPU is available and its memory size is strongly limited, only the second testing matrix (Table 4.2) is used for experiments and *Backward Projection* has to be processed without transposed matrix (Chapter 7). On the platform DGX-1 V100, as there are eight GPUs available and total memory size is 128 GB, experiments on the second testing matrix can be done at high discretion. Here *ForwardProjection* using merge-based SpMV and csr-vector SpMV, as well as *Backward Projection* using merge-based SpMV, csr-vector SpMV and using no transposed matrix are tested and compared with respect to performance when the number of used GPUs changes. However, the third testing matrix (Table 4.3) is still too large for DGX-1. Hence the only experiment on this matrix is to run MLEM algorithm using all eight GPUs and the *Backward Projection* without transposed matrix. The first testing matrix (Table 4.1) is so small that it is meaningless to run tests using this matrix.

The implementation of this bachelor's thesis porting MLEM algorithm to heterogeneous systems is fairly complicated and can be divided into many steps: from loading matrix to coping data from GPUs back to host memory. Some of the steps are processed on CPUs, while the others are finished on GPUs. Generally all works except *Forward Projection*, *Correlation*, *Backward Projection Update* are done on CPUs. The following Table 9.1 lists how works are distributed between CPU and GPU in detail.

Here an important thing is that the matrices (original matrix and its transposition) may be partitioned twice depending on the SpMV algorithms chosen for both projections. In any case matrices are partitioned at least once. The aim of this partition is to divide matrices into smaller ones so that they fit GPU memory size, which is also mentioned in Chapter 8. Apart from that, if the merge-based SpMV algorithm is

Work	Worker
Read Matrix	CPU
Load Matrix to CSR Format	CPU
Calculate Norms	CPU
Read Image	CPU
Transpose Matrix	CPU
Partition Matrices	CPU
Forward Projection	GPU
Correlation	GPU
Backward	GPU
Update	GPU
Sum Up Result Vector	CPU

Table 9.1: Work Division between CPU and GPU

chosen, the smaller matrices on each GPU will be further partitioned into a number of sections and each thread deals with one section. This process is explained in Chapter 5.

Not all works are analyzed with respect to performance. Most of the pre-processing works, including reading matrix, converting matrix to CSR format and calculating norm are not considered when investigating performance. The *ScanTrans* algorithm for matrix transposition is already compared to algorithms provided in Intel MLK and NVIDIA CUSPARSE in [12]. The *Correlation* step and the *Update* step in MLEM algorithm is essentially not different from linear scaling and it is meaningless to analyze the performance of these two steps. Furthermore, the comparison of customized SpMV algorithms (merge-based and csr-vector) against algorithms in NVIDIA CUSPARSE is already carried out in [7] and [11] respectively. So in general, this bachelor's thesis only performs comparison between the two customized SpMV algorithms, with respect to *Forward Projection* and *Backward Projection* and the number of used GPUs. In addition, since the size of section for merge-based SpMV plays an important role, analyses are also carried out on the varying of section size.

In order to investigate correctness and accuracy of the implementation, every time after the result vector is obtained, a summation over the result vector is carried out. Instead of comparing all elements in result vectors, comparisons of those summations are taken. According to data that are already collected, it is reasonable to assume that all functions are implemented correctly with respect to arithmetic. The analysis of accuracy will be discussed later.

As for performance, the main focus is on the time cost. Time cost can be obtained by calling the command *nvprof*. This command outputs detailed information about kernel function after function run. It gives three different time costs: average, minimal and maximal cost per iteration. In this bachelor's thesis only the average time cost per iteration is taken into account.

By default, block size is set to 1024 (full block) and one-dimensional grid is used. Taking the sizes of matrices and GPU memory size into consideration, there is no risk that more blocks are needed than the capacity of one-dimensional grid.

9.2 Experiments on AMD Ryzen 2990WX + NVIDIA Quadro P6000

As aforementioned, because of the limited number of available GPU and memory size, it is only possible to carry out experiments using the second testing matrix (Table 4.2) and *Backward Projection* with no transposed matrix (Chapter 7). As for *Forward Projection*, it is tested using two different SpMV algorithms: merge-based and csr-vector (Chapter 5). Especially, for merge-based SpMV, the choose of section size does impact performance, which is also investigated.

9.2.1 Forward Projection using csr-vector SpMV

In order to test the performance of *Forward Projection* using csr-vector SpMV, the program is run 50 times. Each time 500 iterations are performed. The average time cost per iteration each time is recoreded and shown in Figure 9.1. The effective bandwidth in this case is 306.89 GB/s and it is 70.91% of total bandwidth of Quadro P6000.

9.2.2 Backward Projection using no transposed matrix

For *Backward Projection* using no transposition of matrix, the program is also run 50 times and each time 500 MLEM iterations are carried out. The Figure 9.2 shows the data in boxplot. Here the effective bandwidth is 136.01 GB/s (31.43% of total bandwidth).

9.2.3 Forward Projection using merge-based SpMV

When using merge-based SpMV, the section size does influence performance. Hence for this experiment the main focus is on how time cost per iteration changes on the varying of section size. The range of section size chosen is from 2 to 15 with step size 1.

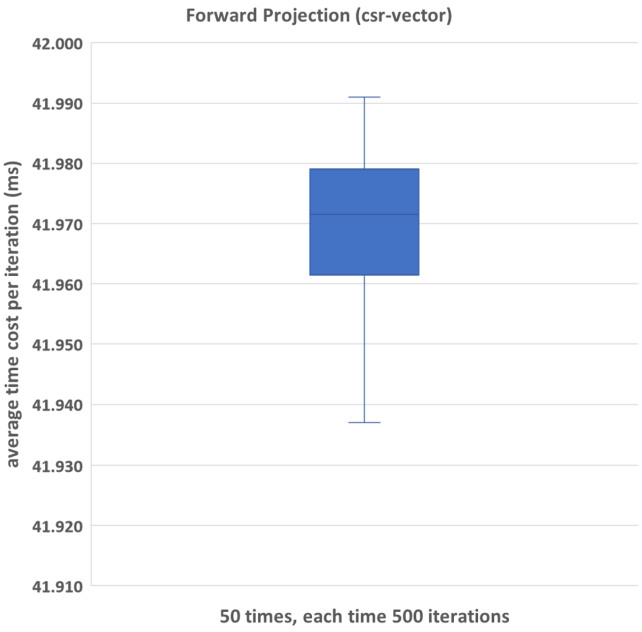


Figure 9.1: Forward Projection using csr-vector SpMV (P6000)

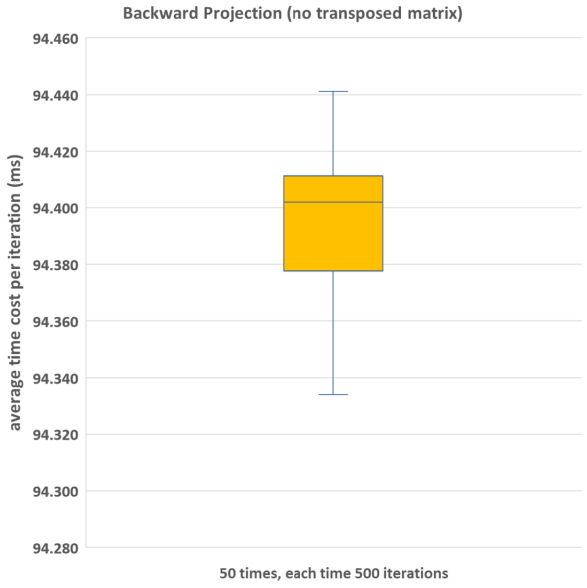


Figure 9.2: Backward Projection using no transposed matrix (P6000)

For section sizes that are larger than 15, performance is significantly decreased. For each section size, the program is run once, with 500 MLEM iterations. The results are presented in Figure 9.3. In best case, namely when section size is equal to 6, the effective bandwidth is 79.19 GB/s (only 18.3% of maximal bandwidth).

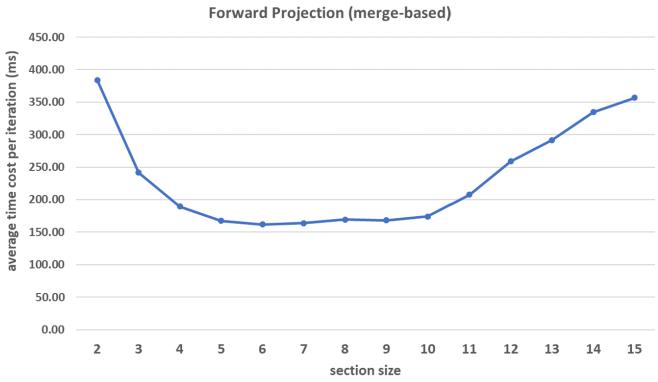


Figure 9.3: Forward Projection using merge-based SpMV (P6000)

9.2.4 Accuracy

Apart from time cost, each time a summation over result vector is performed and used for analyzing accuracy. For this aim, the implementation where *Forward Projection* uses csr-vector SpMV and *Backward Projection* uses no transposition of matrix is considered. The program is run 50 times with 500 MLEM iterations per time. The distribution of summations over result vector are given in Figure 9.4.

9.3 Experiments on DGX-1 V100

As on DGX-1 up to eight GPUs are available and their total memory size is 128 GB, various experiments can be executed. For the second testing matrix which is about 12.8 GB, *Forward Projection* using csr-vector SpMV and merge-based SpMV, as well as *Backward Projection* using csr-vector and merge-based SpMV and using no transposition of matrix are all tested with respect to varying of number of applied GPUs. Moreover the trend of time cost when merge-based SpMV is used and section size changes is also recorded. For accuracy, the distribution of all summations over result vector are put together graphically.

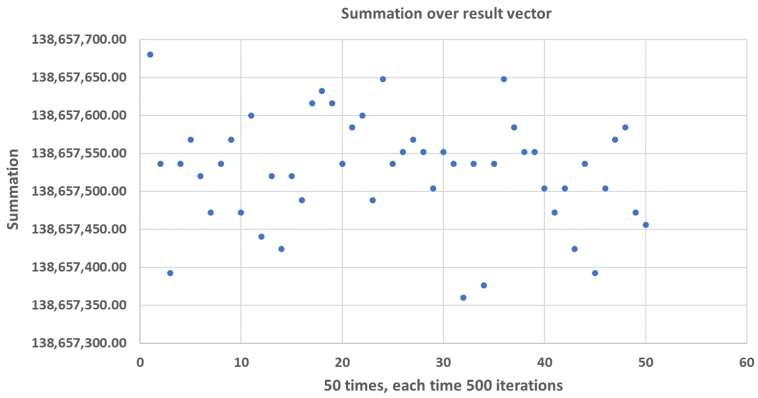


Figure 9.4: Distribution of Time Cost (P6000)

9.3.1 Forward Projection using csr-vector SpMV

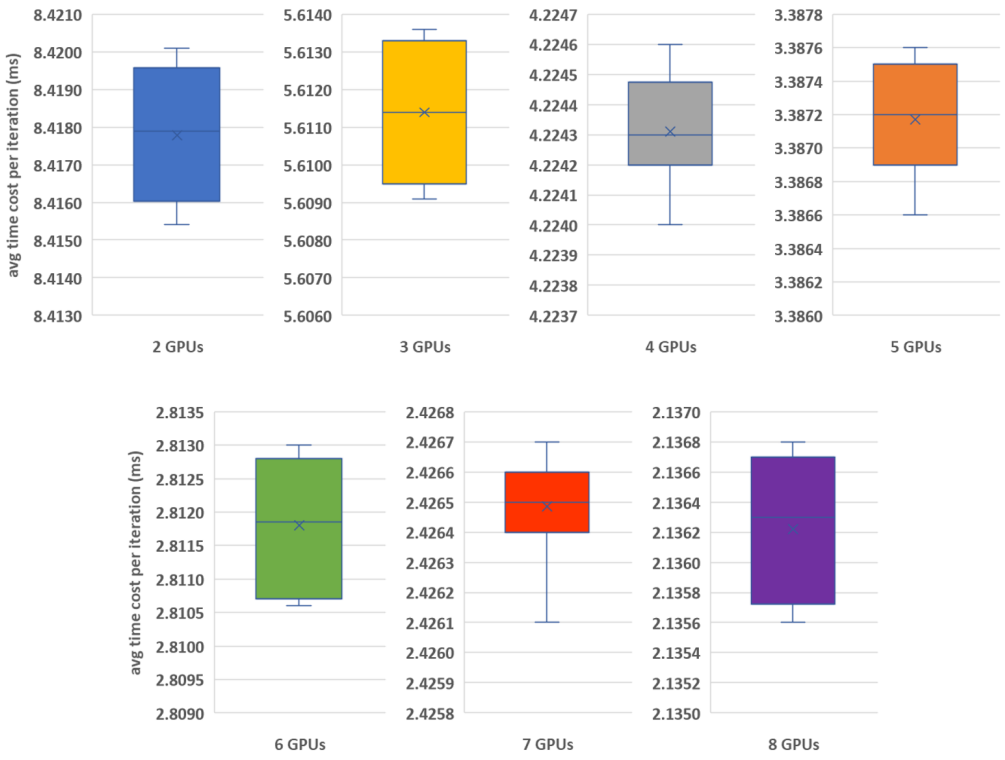
For *Forward Projection* using csr-vector SpMV, the program is run 20 times for every number of GPUs used from 2 to 8. Each time 500 MLEM iterations are performed. The data are represented respectively in Figure 9.5a as box-plots and in Figure 9.5b as line chart. The effective bandwidth is respectively 1525.3 GB/s (85.02% of maximal bandwidth) when 2 GPUs used and 6011.24 GB/s (83.77% of maximal bandwidth) when 8 GPUs used.

9.3.2 Forward Projection using merge-based SpMV

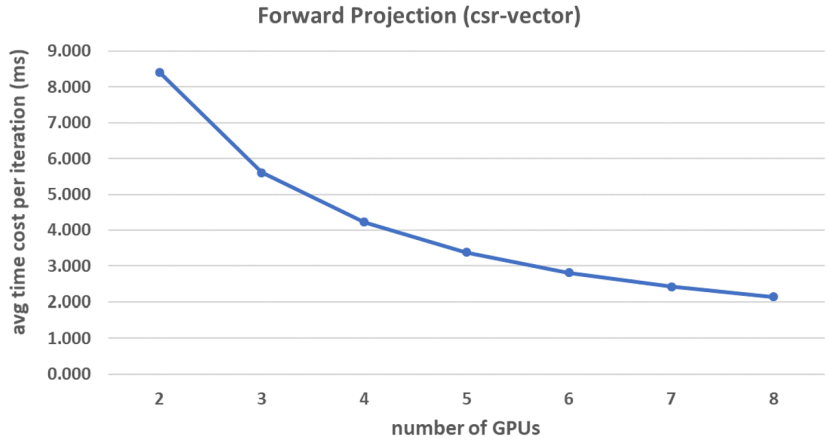
To test *Forward Projection* using merge-based SpMV, for number of GPUs ranged from 2 to 8, the programs is run 20 times with 500 MLEM iterations per time. The section size of merge-based SpMV is set to 5. The data are shown respectively in Figures 9.6a and 9.6b. The effective bandwidth is 434.08 GB/s (24.2% of max bandwidth) when 2 GPUs applied and 1757.46 GB/s (24.49% of max bandwidth) when 8 GPUs applied.

9.3.3 Backward Projection using csr-vector SpMV

For *Backward Projection* using csr-vector SpMV, the program is run 10 times per number of GPUs ranged from 2 to 8. Each time 500 MLEM iterations are performed. The Figures 9.7a and 9.7b provide closer look at the data. The effective bandwidth is respectively 1617.33 GB/s (90.15% of total bandwidth) and 6381.71 GB/s (88.93% of total bandwidth), when 2 GPUs and 8 GPUs are applied.

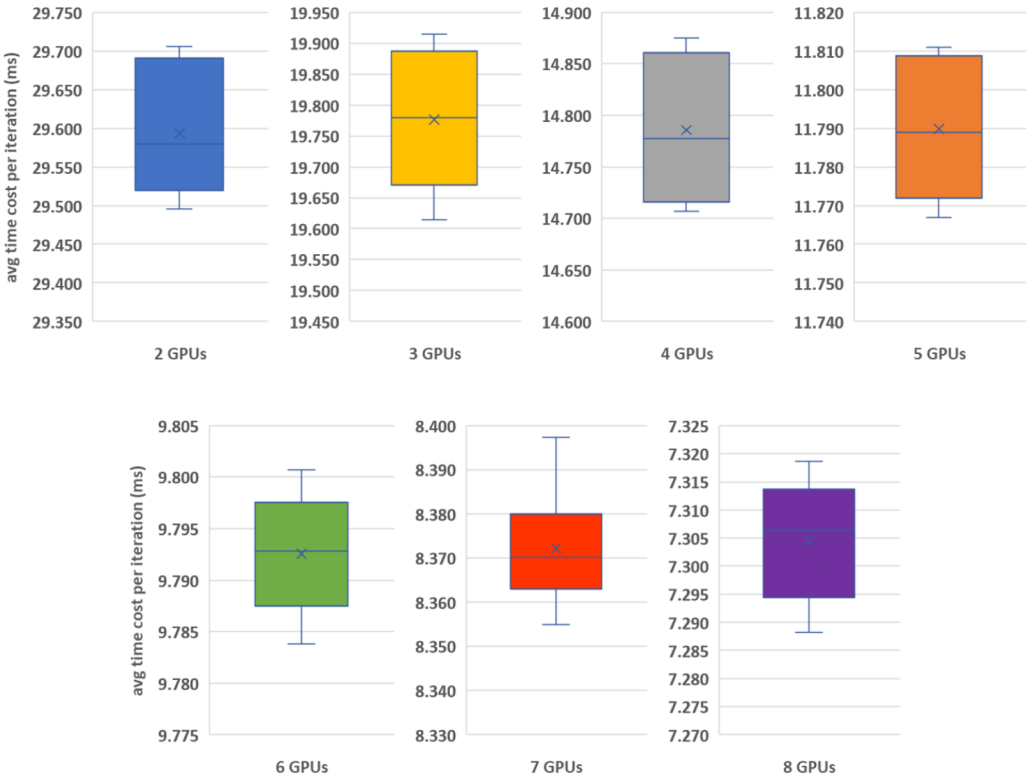


(a)

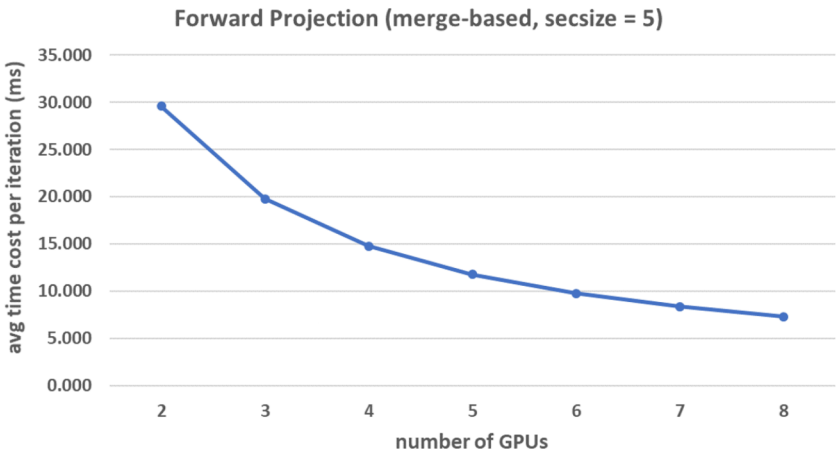


(b)

Figure 9.5: Forward Projection using csr-vector SpMV (DGX-1)

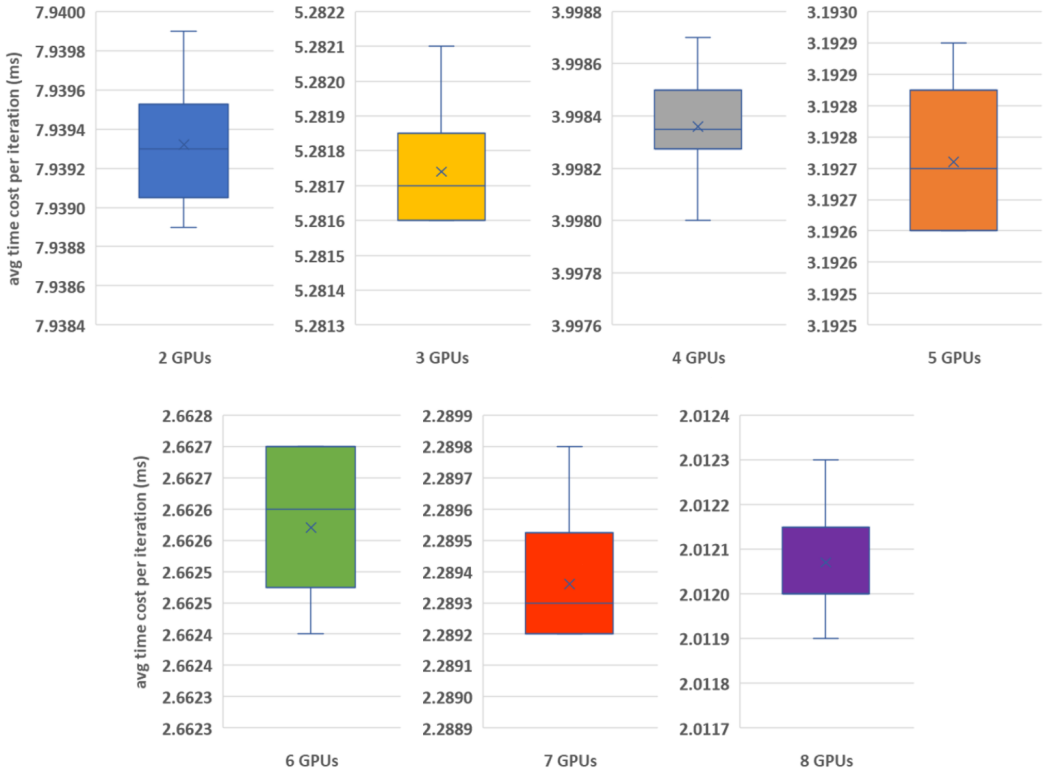


(a)

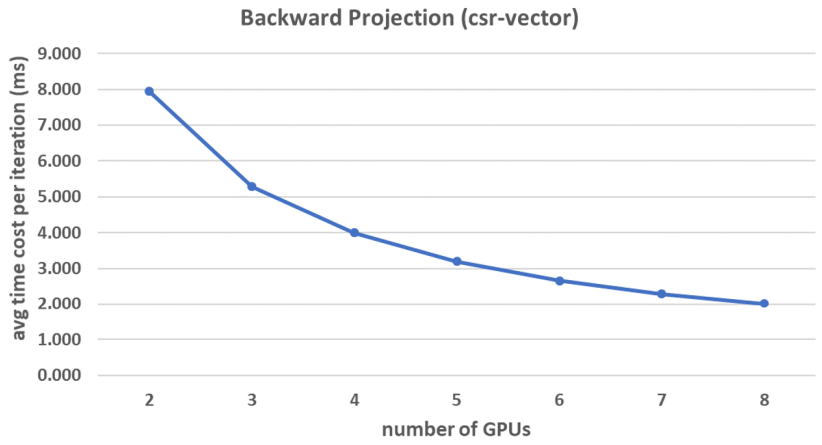


(b)

Figure 9.6: Forward Projection using merge-based SpMV (DGX-1, secsize = 5)



(a)



(b)

Figure 9.7: Backward Projection using csr-vector SpMV (DGX-1)

9.3.4 Backward Projection using merge-based SpMV

To test *Backward Projection* using merge-based SpMV, the program is run 10 times for each number of used GPUs, from 2 to 8. Each program run performs 500 MLEM iterations. The section size of merge-based SpMV is set to 5. The data are presented in Figures 9.8a and 9.8b. The effective bandwidth is 456.94 GB/s (25.47% of maximal bandwidth) when 2 GPUs used and 1861.41 GB/s (25.94% of maximal bandwidth) when 8 GPUs used.

9.3.5 Backward Projection using no transposed matrix

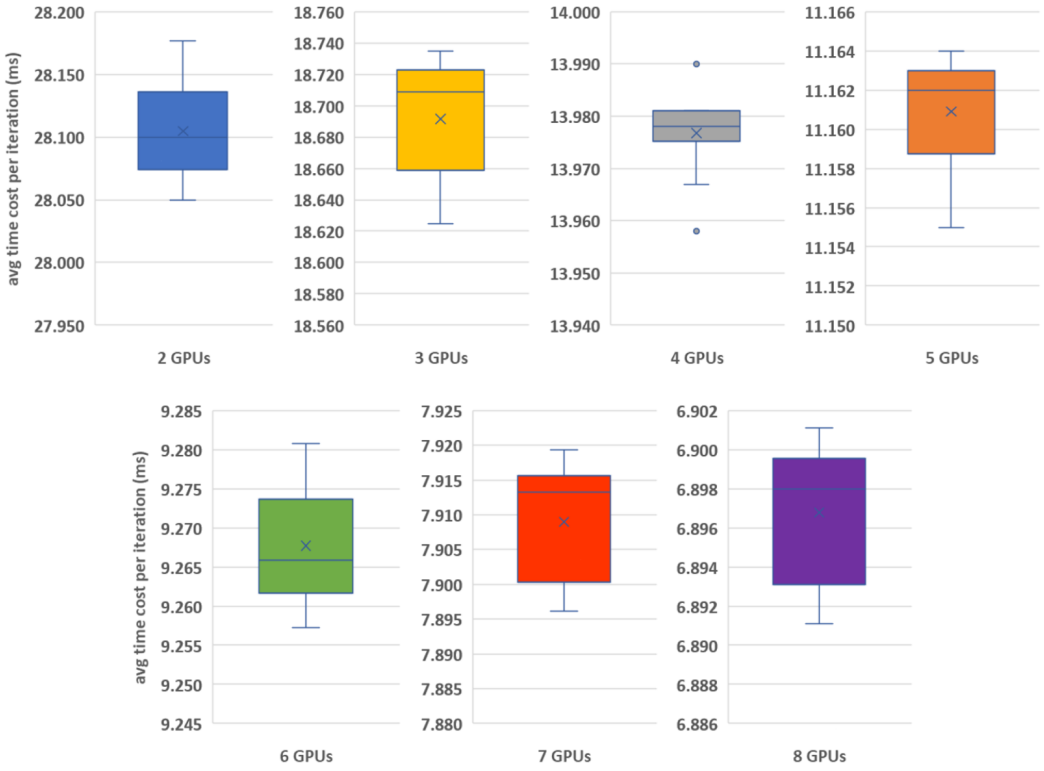
Backward Projection using no transposition of matrix is also tested on performance. For number of GPUs ranged from 2 to 8, the programs is run 20 times with 500 MLEM iterations per time. The records of time cost are shown respectively in Figures 9.9a and 9.9b. The effective bandwidth in this experiment is 982.21 GB/s (54.86% of max bandwidth) when 2 GPUs applied and 3900.36 GB/s (54.35% of max bandwidth) when 8 GPUs applied.

9.3.6 Comparison of FP and BP

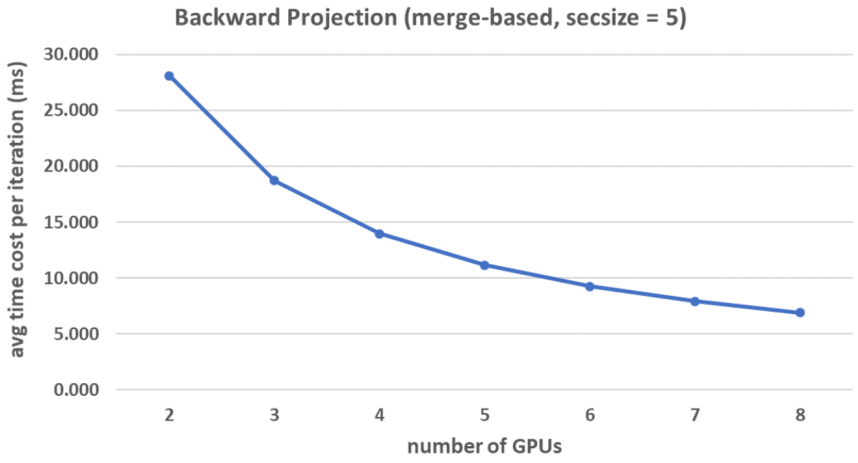
The records of average time cost, respectively for *Forward Projection* using csr-vector SpMV and merge-based SpMV (section size = 5), as well as for *Backward Projection* using csr-vector SpMV and merge-based SpMV (section size = 5) and using no transposed matrix, are also put together for further comparison. The Figure 9.10 provides a closer look at the results.

9.3.7 Influence of Section Size on merge-based SpMV

As is mentioned in Chapter 5, for merge-based SpMV, the section size does play a role when it comes to performance. Section size is the number of *elements* that are processed by a thread, where an *element* is either a nnz or a row switch. In order to investigate the influence of section size on performance of merge-based SpMV, the program is run once with 500 iterations for each section size ranged from 2 to 10, respectively in *Forward Projection* and in *Backward Projection*. For any larger section size, time cost is increased remarkably. In order to exclude the influence of number of applied GPUs, always 2 GPUs are used. The results are shown in Figure 9.11. In best cases, namely section size is 5 for both *Forward Projection* and *Backward Projection*, the effective bandwidth is respectively 434.06 GB/s (24.2% of total bandwidth) and 457.36 GB/s (25.49% of total bandwidth) according to calculation.

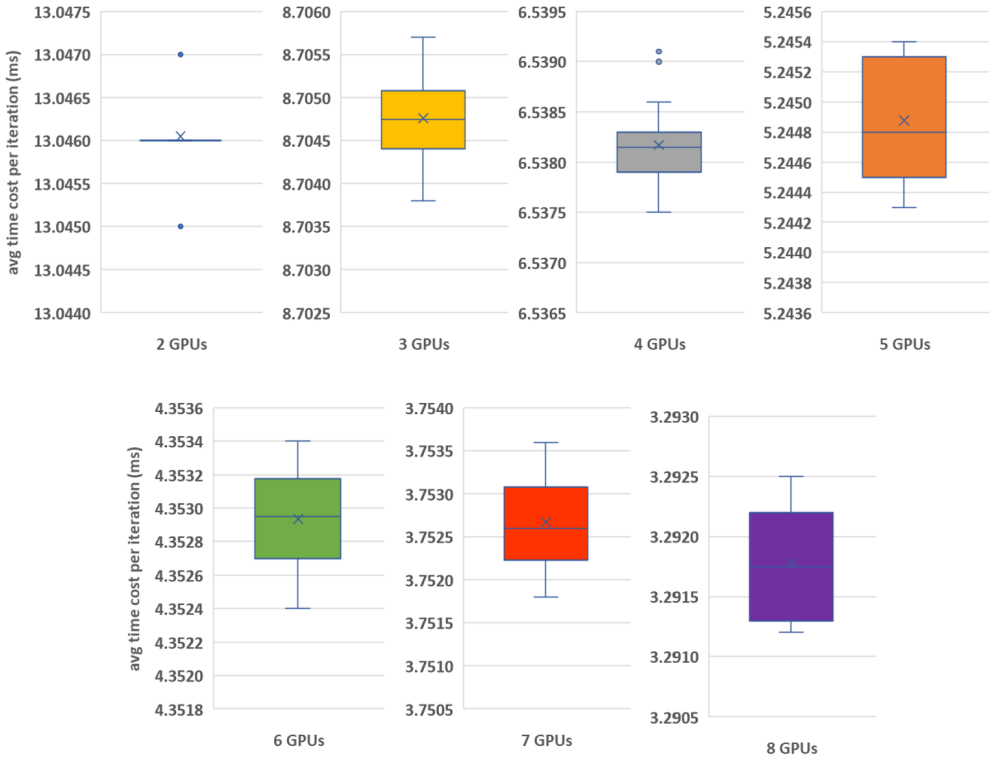


(a)

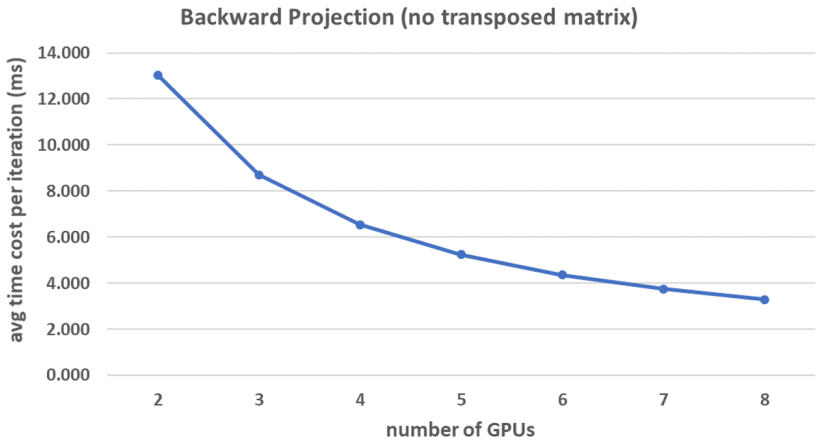


(b)

Figure 9.8: Backward Projection using merge-based SpMV (DGX-1, secsize = 5)



(a)



(b)

Figure 9.9: Backward Projection using no transposed matrix (DGX-1)

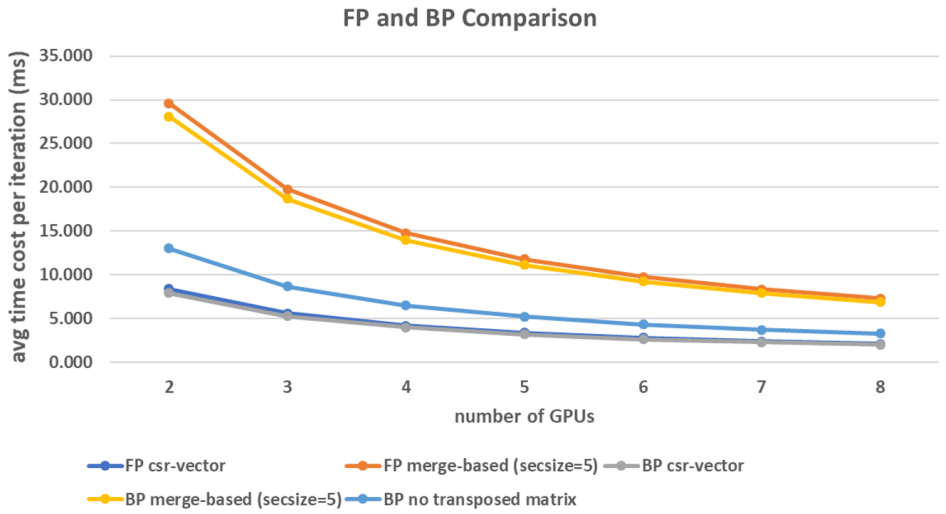


Figure 9.10: Comparison of FP and BP on Performance

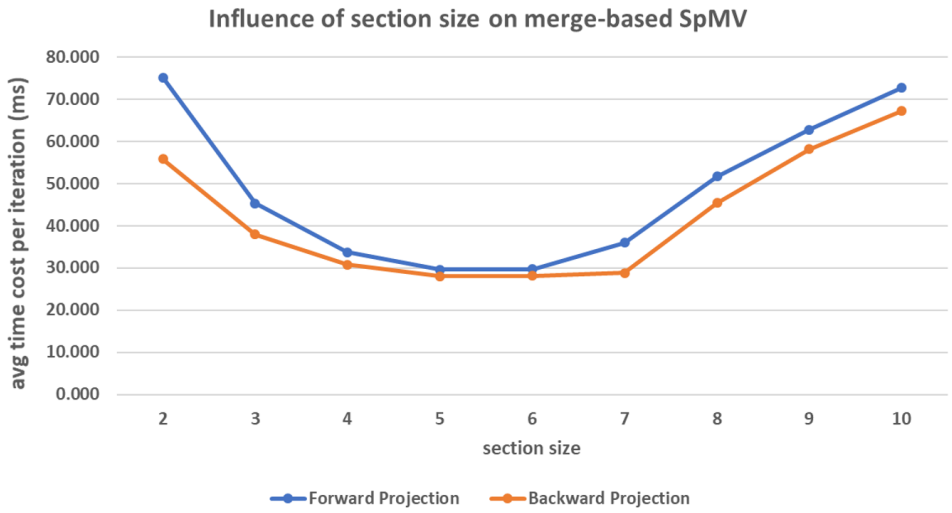


Figure 9.11: Influence of Section Size on merge-based SpMV

9.3.8 Accuracy

A summation over the result vector is performed each time after function run in order to analyze the accuracy of different implementations. Totally, 298 data points of summation are recorded. The Table 9.2 provides information about sources of theses summations and Figure 9.12 depicts the distribution of summations. When testing the influence of number of applied GPUs, the program is run 10 times for all combinations of *Forward Projection* and *Backward Projection* for every number of used GPUs, ranged from 2 to 8. The case when solely 1 GPU is used is neglected, as single NVIDIA Tesla V100 has only 16 GB memory space and is not sufficient to store the transposed matrix. Similarly, when testing the influence of section size on merge-based SpMV, the program is run exactly once per section size ranged from 2 to 10. The case section size is 1 is ignored as it is meaningless that each threads deals with only one nnz. The cases when section size is larger than 10 are also neglected as in these cases the performance is significantly decreased.

In all cases, the program is run with 500 MLEM iterations, using the second testing matrix (Table 4.2). Time cost is acquired through *nvprof* and summation over result vector is calculated on CPU and output on prompt.

Forward Projection	Backward Projection	Times
csr-vector	csr-vector	70 (10 times per number of GPUs)
merge-based (secsize=5)	merge-based(secsize=5)	70 (10 times per number of GPUs)
csr-vector	no transposed matrix	70 (10 times per number of GPUs)
merge-based (secsize=5)	no transposed matrix	70 (10 times per number of GPUs)
merge-based	merge-based (secsize=5)	9 (1 time per secsize FP)
merge-based (secsize=5)	merge-based	9 (1 time per secsize BP)

Table 9.2: Combination of FP and BP

9.3.9 Performance when Using Large Matrix

In the end, the extreme case when an exceedingly large matrix is used in the MLEM algorithm is considered. For this experiment, the third testing matrix (Table 4.3) is used, of which the size is about 100 GB. As the matrix is remarkably large, it is impossible to store a transposition on DGX-1 V100, since the total memory size of eight GPUs on this platform is solely 128 GB. Therefore *Backward Projection* using no transposed matrix (Chapter 7) is applied. And because of the limited access to DGX-1, the program is

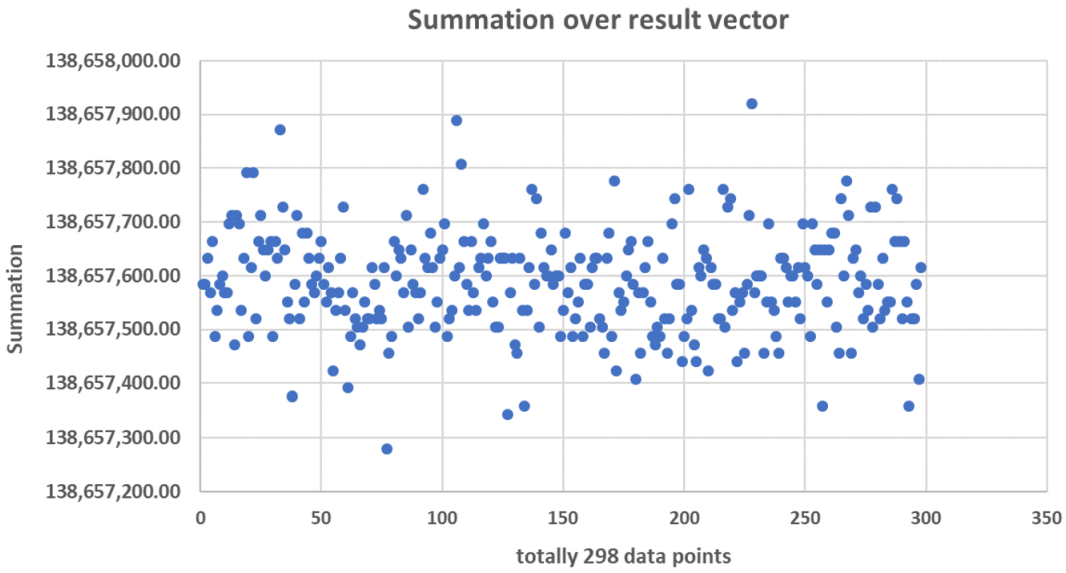


Figure 9.12: Distribution of Time Cost (DGX-1)

run solely once, with 10 MLEM iterations and all eight GPUs used. The algorithm chosen for *Forward Projection* is csr-vector SpMV, as it overwhelms merge-based SpMV considerably on performance. The detaisl of time cost are listed in Table 9.3. In this extreme case, the effective bandwidth remains high for *forward projection* (5972.85 GB/s, 83.23% of theoretical bandwidth), but is rather low for *backward projection* (47.1 GB/s, solely 6.56% of theoretical bandwidth).

Name	Avg Time Cost per Iteration
BP (no transposed matrix)	2.09212s
ncclAllReduce	1.10013s
FP (csr-vector)	16.498ms
Update	49.424us
Correlation	3.2000us

Table 9.3: Performance when Using Large Matrix

9.4 Discussion

Based on the results of experiments shown above, some details of performance and accuracy will be discussed in the following.

9.4.1 Performance of Different SpMV Algorithms

According to Figures 9.1, 9.2, 9.3 as well as 9.10, it is obvious that compared to merge-based SpMV, csr-vector SpMV is much more efficient, no matter for *ForwardProjection* or for *Backward Projection*, and no matter how many GPUs are used. This is in contradict to the conclusion in [7, 10], as it claims that although merge-based SpMV does not provide optimal performance, its time cost is to some extend close to the time cost of SpMV algorithm provided in NVIDIA CUSPARSE library.

When it comes to *Backward Projection*, it is expected that the implementation using transposed matrix should be more efficient than the implementation using no matrix transposition. The comparison above has already proven this prognosis. What is unexpected is that *Backward Projection* using merge-based SpMV is even slower than the version without transposed matrix, although the former does use transposed matrix. Another remarkable discovery is that when both using csr-vector SpMV, *Backward Projection* is a little faster than *Forward Projection*, which is shown Figure 9.10. This is interesting as the transposed matrix of the second testing matrix has a similar shape (Figure 4.2) to its original. Another discovery is that the time cost of *Backward Projection* using no transposition of matrix is largely dependent on the matrix size. As is shown in Table 9.3, the time cost for *Backward Projection* is about 2.092s per iteration, while *Forward Projection* costs only 16.498ms per iteration.

As for the influence of section size in merge-based SpMV, the Figures 9.3 and 9.11 provide a closer look at the trends. In both diagrams the second testing matrix (Table 4.2) is considered. On the platform NVIDIA Quadro P6000, the section size for optimal performance of *Forward Projection* is 6. And on the platform DGX-1 V100, the section sizes for optimal performance are 5 for both *Forward Projection* and *Backward Projection* when two GPUs are applied. However, even if the optimal section size is used, merge-based SpMV is still considerably slower than csr-vector SpMV.

Another discovery is that when using extremely large matrix, the time cost of *ncclAllReduce* is no more ignorable. As is shown in Table 9.3, the time cost for *ncclAllReduce* using *ncclSum* operation is about 1.1s per iteration, which is significantly longer than the time cost for *Forward Projection*, *Update* and *Correlation*. It is

reasonable to assume that if instead of *ncclAllReduce* the *ncclAllGather* function is applied, the overall performance can be somehow improved, which is however impossible according to how the matrix is partitioned (Chapter 8).

9.4.2 Unstable Accuracy

As is shown in Figures 9.4 and 9.12, there is a range for how summations over result vector are distributed, which means that implementations of MLEM algorithm in this bachelor's thesis do not always provide an accurate solution. It is reasonable to assume that the inaccuracy can be attributed to the following two reasons (three reasons when merge-based SpMV is used for *Forward Projection / Backward Projection*).

The first reason is that the large difference in values in the matrix is so large that floating-point errors can be raised when performing arithmetic operations. As is shown in Table 4.2, the smallest and the biggest values in this matrix are respectively $5.50416e^{-24}$ and $8.90422e^{-5}$. The difference between these two values is sufficient to raise floating-point errors.

The second reason, which is theoretically the fundamental reason, lies in the calculation of norms. Norm is simply the summation over a column in matrix. The current function for calculating norms, which comes from previous work of Chair of Computer Architecture and Parallel Systems, applies OpenMP and binary reduction. For such a reduction, there is no fix order for threads summing up values in the column. Since for floating-point operations the order does have impact on results, this will lead to different norm values each time the program is run.

Additionally, if the SpMV algorithm chosen is merge-based, the section size does have an effect on accuracy. Section size describes the number of *elements* that each thread should deal with. An *element* is either a nnz or a row switch. When section size changes, nnzs may be operated in different orders, which may result in different final values each time.

10 Future Work

10.1 Matrix Format

Since the characteristic matrices in PET reconstruction are normally very sparse, it is fairly expensive to store the matrices with simple two-dimensional arrays, as most of the elements in the arrays are 0. It is of importance use some special formats to store the matrices sophisticatedly. In this bachelor's thesis, the CSR format (Compresses Sparse Row, introduced in Chapter 4) is represent matrices. CSR format is widely used as it is relatively easy to carry out matrix-vector and matrix-matrix multiplication on this format. However, CSR format is not specialized and optimized for the MLEM algorithm, as for example *Backward Projection* requires transposed matrix, which is not included in CSR format.

Apart from the CSR format and its transposition *CSC* format (Compressed Sparse Column, which is equivalent to the CSR format of transposed matrix), there are many other existing formats for representation of sparse matrices. For example, there is a modified version of CSR format introduced in [19], which reduces the size of *csr_Vals* array. [20] lists a number of existing formats in detail. Moreover, since the matrices characterizing PET scanner are normally left-upper triangle-shaped (Figures 4.2 and 4.3), it can be possible to develop some formats customized for such matrices.

10.2 Matrix Transposition

Ideally, the *Backward Projection* in MLEM algorithm is not different from a matrix-vector multiplication, if the transposition of matrix is provided or can be pre-processed and the memory size is large enough. As the CSR format is used for storing matrix in this bachelor's thesis, an efficient approach of transpoing matrix in CSR format is required.

Current algorithms for CSR format matrix transposition provided in Intel MLK, AMD CML and NVIDIA CUSPARSE all have their special limitations. For example, the *cusparseScsr2csc* function provided in CUSPARSE library is not suitable for large

matrices. As is explained in Chapter 6, it is impossible to transpose a matrix of 12.8 GB on the platform NVIDIA Quadro P6000 using *cusparseScsr2csc*, even if the memory size of P6000 (24 GB) is almost double of matrix size. This is because that additional to the original matrix, an amount of memory space is required during function run to store temporary variables and arrays. A possible way to solve this problem is to firstly partition the matrix. Then on any heterogeneous systems which have more than one GPUs, each GPU processes only part of the original matrix. After all GPUs finish their work, the transposed parts are aggregated together into a complete transposition of original matrix. However, the aggregation of vectors across GPUs and the communication among GPUs may require additional time cost.

In this bachelor's thesis, a novel parallel matrix transposition algorithm called *ScanTrans* from [12, 13] is used, which runs on CPU and applies openMP. In [12, 13] there is also another approach called *MergeTrans* proposed. However, for MLEM algorithm that are specialized for heterogeneous systems using matrices shaped like in Figures 4.2 and 4.3, there may be other matrix transposition approaches that are more efficient.

Another solution to the problem can be a novel algorithm for transposed SpMV. The approach used in this bachelor's thesis for *backward projection* without transposed matrix is efficient when the matrix is relative small, but rather inefficient in extreme cases where the matrix is large. As is shown in Table 9.3, the effective bandwidth in this case is solely 47.1 GB/s (only 6.56% of total bandwidth of eight NVIDIA Tesla V100). Improvements are in urgent need.

10.3 Use of Pinned Memory and Shared Memory

Pinned memory is a special type of memory in contradict to pageable memory for CUDA programs. Pinned memory is on host memory and is allocated through *cudaHostAlloc* or *cudaHostMalloc* instead of simple *malloc* or *calloc*. Compared to pagable memory, pinned memory will not be swapped out from system memory after allocation. The data transfer between GPU and pinned memory is more efficient than the transfer with pageable memory. However, the time cost for allocation and deacllocation of pinned memory is higher than for pageable memory. Since the main focus is on the performance of MLEM algorithm itself, pinned memory is not applied in this bachelor's thesis, which could be improved in future.

Another possible improvement is the use of shared memory. Shared memory is a

special memory space on device (GPU), which is allocated per block and shared by all threads in the block. The access to shared memory is faster than to global memory, but slower than to local memory. Its size is normally strongly limited, especially the gird size is large (number of blocks too high). In the Sample Code for csr-vector SpMV 5.3, an array of size 1024 is allocated per block. In the implementation for merge-based SpMV 5.2, it is impossible to apply shared memory as the relation between rows in matrix and thread is of type $m : n$ (Each thread may cope with multiple rows, and each row may be processed by multiple threads). The implementations for *Correlation* and *Update* also do not use shared memory as these two steps are fairly simple in comparison to other steps, which can however be improved in future. Other possibilities of applying shared memory may also exist with respect to the current program and to any future novel implementations.

10.4 Work Division between CPU and GPU

In this bachelor's thesis, all pre-processing works like loading matrix, calculating norms and transposing matrix are done on CPU. Other works that are directly related to MLEM algorithm are done on GPUs. In case that the total GPU memory is not large enough to store transposed matrix, *Backward Projection* will be carried out without transposition of matrix.

It is possible to redistribute the work between CPU and GPUs in such case to improve the overall performance. Although it may be impossible to store the whole transposed matrix on GPUs, part of the transposed matrix can still be used. For example, on NVIDIA P6000 with 24 GB memory size, if the second testing matrix is used (Table 4.2), the memory space left is not sufficient to store its transposition. However, a large part of the transposed matrix can still be kept on the GPU, with the left part kept on host memory. Then the csr-vector SpMV algorithm can be used for *Backward Projection* on GPU. Since in this case GPU holds only part of matrix, it holds also only part of result vector (similar to 8.1). The other part of result vector has to be calculated by carrying out *Backward Projection* on CPU. To get a complete result vector, the result vectors on GPUs and CPU have to be concatenated.

11 Platforms

For this bachelor's thesis, two different platforms are used. During developing phase and testing phase, a platform combining AMD Ryzen Threadripper 2990WX 32-Core and NVIDIA Quadro P6000 is used. For tests using large matrices and multiple GPUs, the platform NVIDIA DGX-1 V100 provided by Leibniz Supercomputing Center (LRZ) is used, which is equipped with two Intel Xeon E5-2698 v 20-Core and eight NVIDIA Tesla V100.

11.1 AMD Ryzen Threadripper 2990WX + NVIDIA Quadro P6000

This platform is mainly used during developing phase and for tests using small matrices. It is provided by the Chair of Computer Architecture and Parallel Systems and it comprises additionally a second GPU NVIDIA Tesla K20c, which is not used in this bachelor's thesis because of its low performance and strongly limited memory size. The Tables 11.1 and 11.2 list the specifications of CPU and GPU on this platform in more detail.

Specification	Value
Model	AMD Ryzen Threadripper 2990WX
OS	Ubuntu 18.04.2 LTS
Host RAM	64 GB
Core	32
Thread	64
Base Frequency	3.0 GHz
Turbo Frequency	4.2 GHz
L1-Cache	3 MB
L2-Cache	16 MB
L3-Cache	64 MB

Table 11.1: AMD Ryzen Threadripper 2990WX [21]

Specification	Value
Model	NVIDIA Quadro P6000
Architecture	Pascal
Compute Capability	6.1
Memory Size	24 GB
Memory Type	GDDR5X
Memory Bus Width	384 bit
Bandwidth	432.8 GB/s
CUDA Cores	3840
Streaming Multiprocessors	30
FP32 Performance	12.63 TFLOPS

Table 11.2: NVIDIA Quadro P6000 [22]

11.2 NVIDIA DGX-1 V100

The NVIDIA DGX family are integrated systems which are specialized for deep learning. For this bachelor's thesis, a DGX-1 V100 provided by LRZ is used. Compared to the last generation DGX-1 P100, DGX-1 V100 achieves dramatically higher throughput with up to 3.1× faster deep learning training for convolutional neural networks. It has two 20-core Intel Xeon E5-2698 v4 CPUs and eight NVIDIA Tesla V100 GPUs combined through high-performance NVLink. Further specifications of the CPUs and GPUs are listed in the Tables 11.3 and 11.4.

Specification	Value
Model	Intel Xeon E5-2698 v4
OS	Red Hat Enterprise Ubuntu
Host RAM	512 GB
Core	20
Thread	40
Base Frequency	2.20 GHz
Turbo Frequency	3.60 GHz
L1-Cache	640 KB
L2-Cache	5 MB
L3-Cache	50 MB

Table 11.3: Intel Xeon E5-2698 v4 [23, 24]

Efficiency of GPU-accross application can be constrained by the performance of PCIe

Specification	Value
Model	NVIDIA Tesla V100
Architecture	Volta
Compute Capability	7.0
Memory Size	16 GB
Memory Type	HBM2
Memory Bus Width	4096 bit
Bandwidth	897.0 GB/s
CUDA Cores	5120
Streaming Multiprocessors	80
FP32 Performance	15.67 TFLOPS

Table 11.4: NVIDIA Tesla V100 [23, 24]

bus connections between GPUs. To address this problem, the eight NVIDIA Tesla V100 GPUs on DGX-1 are connected through high-performance NVLink. NVLink uses new High-Speed Signaling interconnect (NVHS) from NVIDIA, which provides a peak bandwidth of 25 GB/s. Each Tesla V100 GPU on DGX-1 has six NVLink connection points and each point address a point-to-point connection to another GPU (Figure 11.1). These interconnects shape a hybrid cube-mesh network and the topology of this network can be visualized by the Figure 11.2.

Thanks to the high-speed interconnects between GPUs on DGX-1, it is possible to take advantage of functions provided in NVIDIA Collective Communications Library (NCCL). NCCL functions have to be used in bachelor's thesis since large matrices cannot be kept on single GPU and it is desired to investigate performance when different numbers of GPUs are applied, which will be discussed later.

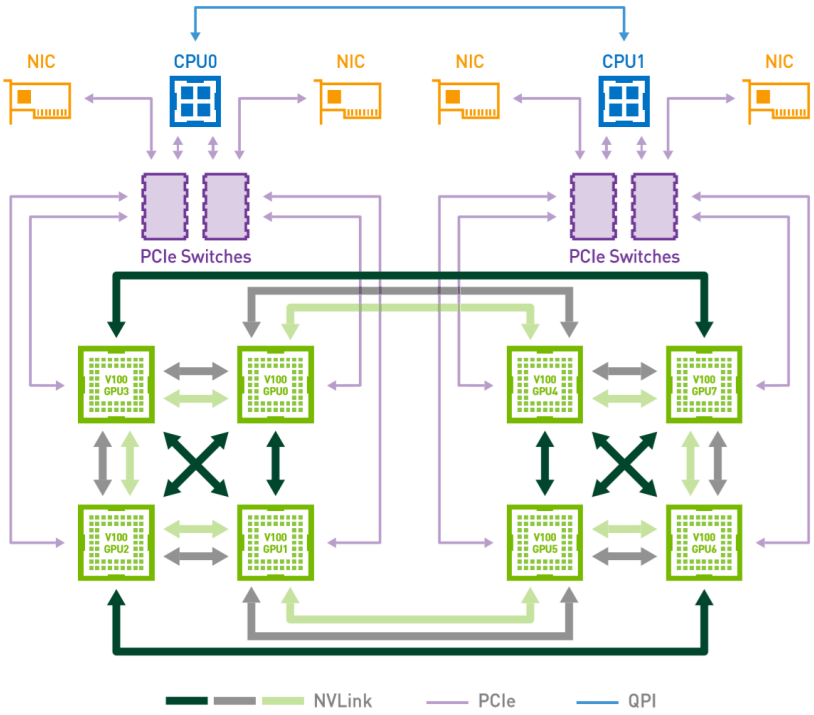


Figure 11.1: DGX-1 Connections [23]

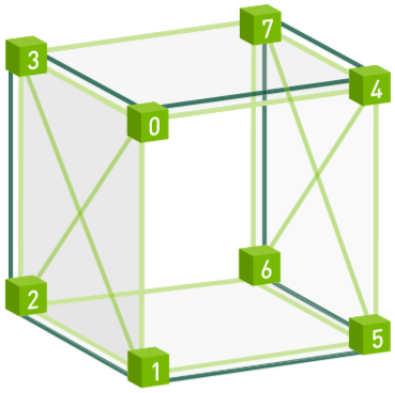


Figure 11.2: DGX-1 Topology [23]

List of Figures

List of Tables

List of Codes

Bibliography

[1] D. L. Bailey, D. W. Townsend, P. E. Valk, and M. N. Maisey. *Positron Emission Tomography*. Springer, 2005. DOI: 10.1007/b136169.

[2] D. P. McElroy, M. Hoose, W. Pimpl, V. Spanoudaki, T. Schüler, and S. I. Ziegler. "A true singles list-mode data acquisition system for a small animal PET scanner with independent crystal readout." In: *Physics in Medicine and Biology* 50.14 (July 2005), pp. 3323–3335. DOI: 10.1088/0031-9155/50/14/009.

[3] T. Küstner, J. Weidendorfer, J. Schirmer, T. Klug, C. Trinitis, and S. Ziegler. "Parallel MLEM on Multicore Architectures." In: *Computational Science – ICCS 2009*. Ed. by G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. A. Sloot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 491–500. ISBN: 978-3-642-01970-8.

[4] H. M. Hudson and R. S. Larkin. "Accelerated image reconstruction using ordered subsets of projection data." In: *IEEE Transactions on Medical Imaging* 13.4 (Dec. 1994), pp. 601–609. ISSN: 0278-0062. DOI: 10.1109/42.363108.

[5] R. M. Lewitt and S. Matej. "Overview of methods for image reconstruction from projections in emission computed tomography." In: *Proceedings of the IEEE* 91.10 (Oct. 2003), pp. 1588–1611. ISSN: 0018-9219. DOI: 10.1109/JPROC.2003.817882.

[6] J. Cui, G. Pratx, B. Meng, and C. S. Levin. "Distributed MLEM: An Iterative Tomographic Image Reconstruction Algorithm for Distributed Memory Architectures." In: *IEEE Transactions on Medical Imaging* 32.5 (May 2013), pp. 957–967. ISSN: 0278-0062. DOI: 10.1109/TMI.2013.2252913.

[7] D. Merrill and M. Garland. "Merge-based Sparse Matrix-vector Multiplication (SpMV) Using the CSR Storage Format." In: *SIGPLAN Not.* 51.8 (Feb. 2016), 43:1–43:2. ISSN: 0362-1340. DOI: 10.1145/3016078.2851190.

[8] D. Merrill and M. Garland. "Merge-Based Parallel Sparse Matrix-Vector Multiplication." In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2016, pp. 678–689. DOI: 10.1109/SC.2016.57.

[9] N. Bell and M. Garland. "Efficient Sparse Matrix-Vector Multiplication on CUDA." In: *NVIDIA Technical Report* (Dec. 2008). Available at `https://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf`.

[10] D. Merrill. *MERGE-BASED SpMV*. Available at `https://images.nvidia.com/events/sc15/pdfs/sc15-Merge-Based-Parallel-Sparse-Matrix-Vector-Multiplication-merrill.pdf`. Last Access: Sep. 2nd. 2019.

[11] A. Gupta. "Implementation and Evaluation of MLEM-Algorithm on GPU using CUDA." Masterarbeit. Munchen, 80805: Technische Universität München, 2018.

[12] H. Wang, W. Liu, K. Hou, and W.-c. Feng. "Parallel Transposition of Sparse Data Structures." In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. Istanbul, Turkey: ACM, 2016, 33:1–33:13. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926291.

[13] H. Wang, W. Liu, K. Hou, and W.-c. Feng. *Parallel Transposition of Sparse Data Structures*. Available at `http://people.cs.vt.edu/~kaixihou/slides/ICS'16_full_v2.pdf`. Last Access: Sep. 2nd. 2019.

[14] L. A. Shepp and Y. Vardi. "Maximum Likelihood Reconstruction for Emission Tomography." In: *IEEE Transactions on Medical Imaging* 1.2 (Oct. 1982), pp. 113–122. ISSN: 0278-0062. DOI: 10.1109/TMI.1982.4307558.

[15] A. Elafrou, G. I. Goumas, and N. Koziris. "A lightweight optimization selection method for Sparse Matrix-Vector Multiplication." In: *ArXiv* abs/1511.02494 (2015).

[16] P. Nee. *Introduction to GPGPU and CUDA Programming: Memory Coalescing*. Available at `https://cvw.cac.cornell.edu/GPU/coalesced`. Last Access: Sep. 2nd. 2019.

[17] *CUDA Toolkit Documentation*. Available at `https://docs.nvidia.com/cuda/cusparse/index.html`. Last Access: Sep. 2nd. 2019.

[18] *NVIDIA Collective Communication Library (NCCL) Documentation*. Available at `https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/index.html`. Last Access: Sep. 2nd. 2019.

[19] T. Küstner, J. Weidendorfer, J. Schirmer, T. Klug, C. Trinitis, and S. Ziegler. "Parallel MLEM on Multicore Architectures." In: *Computational Science – ICCS 2009*. Ed. by G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. A. Sloot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 491–500. ISBN: 978-3-642-01970-8.

[20] M. Grossman. *101 Ways to Store a Sparse Matrix*. Available at `https://medium.com/@jmaxg3/101-ways-to-store-a-sparse-matrix-c7f2bf15a229`. Last Access: Sep. 2nd. 2019.

[21] *AMD Ryzen™ Threadripper™ 2990WX Prozessor*. Available at `https://www.amd.com/de/products/cpu/amd-ryzen-threadripper-2990wx`. Last Access: Sep. 2nd. 2019.

[22] *NVIDIA Quadro P6000*. Available at `https://www.techpowerup.com/gpu-specs/quadro-p6000.c2865`. Last Access: Sep. 2nd. 2019.

[23] *White Paper: NVIDIA DGX-1 With Tesla V100 System Architecture*. Available at `https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf`. Last Access: Sep. 2nd. 2019.

[24] *NVIDIA TESLA V100 GPU ARCHITECTURE*. Available at `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. Last Access: Sep. 2nd. 2019.