Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Implementation and Evaluation of MLEM-Algorithm on GPU using CUDA

Apoorva Gupta

Computational Science and Engineering (International Master's Program)

Technische Universität München

Master's Thesis

Implementation and Evaluation of MLEM-Algorithm on GPU using CUDA

Author:	Apoorva Gupta
1st examiner:	Prof. Dr. rer. nat. Martin Schulz
2nd examiner:	PD Dr. rer. nat. habil. Josef Weidendorfer
Assistant advisors:	M.Sc. Dai Yang
	Dipl.-Tech. Math. (Univ.) Tilman Küstner
Submission Date:	May 15th, 2018

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

May 15th, 2018 Apoorva Gupta

Acknowledgments

First, I would like to thank Prof. Martin Schulz to initially give me various options to choose from for my master's thesis and to later agree to supervise the chosen one. His regular guidance and inputs helped me to explore the unknown and improve the quality of work many-folds.

I could not be more thankful to Dr. Josef Weidendorfer for agreeing to also supervise the thesis.

I would like to also thank Dai Yang and Tilman Küstner without their support this thesis would not have been possible. They have never once hesitated to provide me the help that I needed while writing the thesis and have been very patient to pass on their knowledge and experience no matter how embarrassing my queries were.

I would also like to take this opportunity to thank Leibniz Supercomputing Center(LRZ) to grant me the access to the best possible resources in the form of MAC Cluster and the DGX-1 machine, with the help of which I could test my assumptions and which made the thesis even more interesting. I would not have been able to accomplish the work without the very helpful documentation of CUDA, CuBLAS, CuSparse, NVML libraries and the equally helpful CUDA forum maintained by NVIDIA.

Lastly, a big thanks goes to my Mother and Father for their unwavering belief in me and the support they have provided, especially during the moments of adversity, which has helped me to come back stronger every time.

"Redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end."

-Herb Sutter

Abstract

With the release of CUDA, a parallel computing platform and application programming interface (API) model, in 2007 by NVIDIA the learning curve to code on the GPUs have drastically reduced. Along with the performance benefits the use of GPU promises for a variety of applications, it has become very enticing for the software developers and researchers to make use of this new hardware/software stack for faster feedback to their problems.

This thesis is an attempt to solve the computationally expensive Maximum Likelihood Expectation Maximization (MLEM) algorithm with respect to the image reconstruction in Positron Emission Tomography (PET). The CuBLAS, CuSparse and NVML libraries, provided by NVIDIA, have been extensively used to run the algorithm and to harness the full power of the GPUs.

The most expensive operation in the entire process is the transpose Sparse Matrix Vector Multiplication(SPMV_T) for which the functions provided by the CuSparse libraries were used and which were later bench-marked against the custom kernels developed during the thesis. Apart from that the effect of multi GPU, Cuda Aware MPI, pinned memory and hybrid computing have also been studied with respect to the performance and accuracy of the results.

Finally, the last section has been dedicated to the discussion of the limitations of present implementation and how those limitations could be overcome by making the code resource aware. It also discusses how the performance of the code could be improved by using the merge based SPMV to rewrite the most expensive loop operation i.e. SPMV_T.

Contents

Acknowledgements vii

Abstract ix

I. Introduction and Background Theory 1

1. PET Image Reconstruction 2

2. Related Work 4

3. MLEM Algorithm and System Matrix 5

4. Sparse Matrix Format, SPMV and SPMV_T 10
- 4.1. Sparse Matrix Vector Multiplication(SPMV) 11
- 4.2. Transpose Sparse Matrix Vector Multiplication(SPMV_T) 12

5. GPU Architecture and Programming 14
- 5.1. Grids, Blocks and Threads 15
- 5.2. Register, Local, Constant, Shared, Global, and Texture memory 17
- 5.3. Host, Device and Global Kernels 19
- 5.4. Streams and Events 19
- 5.5. Asynchronous Kernels and Copy Operations 19
- 5.6. Atomic Operations 20
- 5.7. Error Checking 20
- 5.8. Coalesced Memory Access 21
- 5.9. Pinned Memory 23
- 5.10. Bank Conflicts 23

6. Other Terminologies 25
- 6.1. Heterogeneous Computing 25
- 6.2. CUDA Aware MPI 25
- 6.3. Floating Point Arithmetic 27

II. Experiments and Results **31**

7. Benchmarking MAC Cluster **32**

8. Effect of Cuda Aware MPI, Pinned Memory and Custom Kernels **34**

9. Effect of Splitting Work between CPU and GPU **39**

10. Performance vs Number of GPU **43**

11. Performance on CPU vs OpenMP Threads **47**

12. Compilation for Specific Architecture **50**

13. Profiling using NVVP **52**

III. Future Work and Conclusion **54**

14. Future Work **55**
14.1. Resource Aware Computing 55
14.2. Merge Based CsrMV 57

15. Conclusion **59**

Appendix **62**

A. Computational Resources **62**
A.1. DGX-1 62
A.2. Mac Cluster 64

B. Compiling and Using CUDA Aware OpenMPI on Mac Cluster **65**

C. Compiling NVML library on Mac Cluster **66**

D. Notes on MLEM Implementation **67**

Bibliography **69**

Part I.

Introduction and Background Theory

1. PET Image Reconstruction

The *Positron Emission Topography(PET)* is a nuclear medicine imaging technique used to measure the physiological function by looking at blood flow, metabolism, neurotransmitters and radio-labelled drugs. In this process the tracer injected into the body of a living subject typically contains a small amount of radio-nuclides, like carbon-11 and nitrogen-13, having a short half life. Nowadays these PET scanners are used in different field of medicines namely Oncology, Neuro-imaging, Infectious diseases etc.[1]

A PET scanner consists of number of fixed detectors usually arranged in a ring around the subject injected with the tracer. As the tracer undergoes positron emission decay it emits a positron. As the positron travels into the subject's body it loses its kinetic energy and is finally able to interact with a electron in the body which ultimately leads to the emission of two gamma photons of 511 keV which travel into approximately opposite directions. If these emitted gamma photons are detected by the detectors (in approximately opposite direction) in a short time window (typically of a few nanoseconds) then the event is recorded as positive else the detection is discarded. [2, 3]

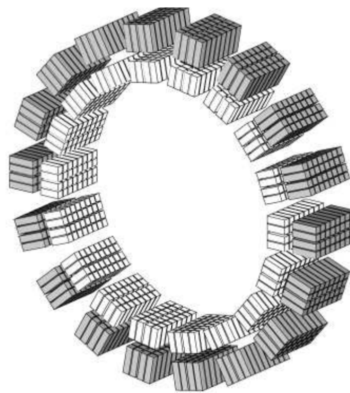


Figure 1.1.: Configuration of MADPET-II[3]

Now since the emitted gamma photons travel almost 180 degrees to each other, it is possible to localize their source along the straight line of coincidence which is typically called the *Line of Response(LOR)*. The more the number of detectors the more the number of detected events which ultimately leads to better quality of the measurements. The coverage

of the three-dimensional space of interest, i.e. *Field of View(FOV)*, by the LOR affects the achievable resolution. [1]

A possible configuration of an experimental small animal PET scanner (see Figure 1.1) developed at the *Department of Nuclear Medicine, Technische Universität München* adds a second ring of gamma photons detectors which leads to better spatial resolution. The additional ring quadruples the amount of measured data and increases the computational demand of post processing step significantly for 3D image reconstruction.

2. Related Work

There have been several algorithms made available over the years to reconstruct the original PET images. All these algorithms could be classified under 2 major categories i.e. Analytic and Iterative. A comprehensive overview along with the mathematical foundation for both analytic and iterative algorithms has been given in [4].

The most widely used analytic PET image reconstruction algorithm is *Filtered Back Projection (FBP)*. Since the analytic algorithms are linear they are very helpful in quantitative data analysis by allowing an easier control over the spatial resolution and noise correlations, they have remain important even today.

The noise introduced during the data collection in PET scanners due to the gamma photons either not travelling 180 degrees apart or because of they being registered incorrectly by the detectors leads to the FBP not performing very well with respect to the noise in the measured data and it ultimately throws up artifacts in the reconstructed images. This problem could be corrected upto a certain extent by pre-processing the data before image reconstruction.[1]

The two widely used iterative algorithms are *Maximum Likelihood Expectation Maximization (MLEM)* which was first developed by *Shepp and Vardi in 1982*[5] and *Ordered Subset Expectation Maximization (OSEM)* developed by *Hudson and Larkin in 1994* [6].

The MLEM algorithm is a standard statistical estimation method. It works by taking an initial guess of the image to be reconstructed and in each iteration maximizes the likelihood function. While the MLEM algorithm provides a consistent and predictable convergence results, it is much more computationally expensive compared to FBP.

The OSEM algorithm partitions the data into multiple subsets which are mutually exclusive and it uses only one subset of data for each update. This has the benefit that each pass over the entire data involves a greater number of updates, which leads to significant acceleration over MLEM.

The different possible splitting of the matrix for parallelization has been discussed in great detail by *Chen and Lee [7]* while the performance difference between OSEM and MLEM has been compared by *Lalush and Tsui [8]*.

3. MLEM Algorithm and System Matrix

The MLEM algorithm aims to solve the linear system represented by the equation $Af=g$. Here A is defined as a 2D system matrix of size MxN and it describes the geometrical and physical properties of the scanner. The element a_{ij} in the matrix represents the probability of a gamma photon discharge from a voxel j being recorded by a given pair of detectors i. The reason for the sparsity of the system matrix is because of the fact that the gamma photons detected by a detector pair could only originate in a relatively small number of voxels. [3]

The f in the linear equation represents the 3D image that has to be reconstructed and it is a vector of size N (number of voxels) . Finally the g in the aforementioned equation represents the measured data and it is a vector of size M(number of detector pairs).

The equation that has to be solved iteratively to get a final representation of the scanned object is represented by 3.1.

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

The f_j^{q+1} (left hand side) in the equation 3.1 represents an element in the updated image calculated while f_j^q (right hand side) represents the same element from the previous iterative step.

The iterative step to calculate the next image update can be broken down into multiple small steps. The first of them is to calculate the *norm* of the matrix, equation 3.2.

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

The *norm* is calculated per row of the matrix and is the sum of all the elements in a row of system matrix. The size of the *norm* vector is N (number of columns) and it has to be calculated just once.

The next step is to initialize the image to some value to start the iterative step. The initial value is the fraction of the summation of the g vector and the summation of the *norm* vector.

The equation 3.3 shows this step it in a mathematical form.

$$f_k = \frac{\sum_{i=1}^{m} g_i}{\sum_{j=1}^{n} norm_j} \tag{3.3}$$

The next step is called the *Forward Projection (FP)*, equation 3.4 , and it involves the multiplication of the system matrix A with the image vector f^q. This step leads to the generation of a vector *fwproj* of size M (number of rows).

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

The next step in the quest is a scaling step, equation 3.5. In this step each element in the measured data vector g is divided by the corresponding element in the vector *fwproj*. The vector generated after this step is called *correlation* and is of size M (number of rows).

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

After calculating the *correlation*, the forward projection is compared to the actual measurement and a correction factor is derived, equation 3.6. This step is called the *Backward Projection (BP)* and is equivalent to a transpose sparse matrix vector multiplication and leads to the generation of the *update* vector of size N (number of columns).

$$update_j = \sum_{l=1}^{m} a_{ij} correlation_i \tag{3.6}$$

Finally the result from backward projection is used to scale the image vector from the previous iteration to give us the newly updated image, equation 3.7.

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

The entire algorithm has been presented as a pseudo code in 1. The steps in lines 1-17 refers to the operations that have to be performed just once and it includes setting up the environment, allocation and initialization of the necessary vectors. The steps in lines 18-28 are performed based on the number of iterations required and in these steps the image is continuously updated. The Image Sum in line 24 is calculated by summing up all the values in the newly updated image vector f and it helps to check if the image is updating

properly or not. Subsequently, the lines 32-33 represents the de-allocation of the memory signalling the completion of the code.

Algorithm 1 MLEM Algorithm

```
1:  function MLEM(mpi, ranges, matrix, lmsino, image, nIterations, chkpt)
2:      function FURTHER SPLITTING()                                        ▷ timed
3:          function GET CONFIG TO USE()
4:              function GET REQUESTED CONFIG()
5:                  function SANITY CHECK OF PARAMETERS()
6:
7:      Memory Allocation and Initialization                                ▷ timed
8:      function CSR FORMAT FOR DEVICES()                                   ▷ timed
9:      Allocating and Copying Memory To Device                             ▷ timed
10:     Norm Calculation [equation 3.2]                                     ▷ timed
11:     Norm Reduction                                                      ▷ timed
12:
13:     if chkpt then
14:         Load checkpoint
15:     else
16:         Calculate Initial Value and Initialize Image [equation 3.3]    ▷ timed
17:
18:     for iter < Iterations do
19:         Calculate Forward Projection [equation 3.6]                     ▷ timed
20:         Calculating Correlation [equation 3.5]                          ▷ timed
21:         Calculate Backward Projection [equation 3.6]                    ▷ timed
22:         Reduce Backward Projection                                      ▷ timed
23:         Image Update [equation 3.7]                                     ▷ timed
24:         Image Sum                                                       ▷ timed
25:
26:         if (iter+1)/chkpt then
27:             Create Checkpoint
28:
29:     if num rows on cpu == 0 then
30:         Copy image from device to host
31:
32:     Free Memory
33:     return Time Data
```

As mentioned earlier, the system matrix A is a 2D matrix that describes the geometrical and physical properties of the scanner. The rows in the matrix represents all different pair

of detectors and the columns represent the voxels from which the gamma photons could come from. The final matrix that is generated is very sparse due to the fact that for any given voxel, the detector pairs in which the detectors are approximately 180 degrees apart, are the only viable candidates to receive non zero values.

Parameter	**Value**
Total Size(Bytes)	12,838,607,884
Rows (Pair of detectors)	1,327,104
Columns (Voxels)	784,000
Total Non Zeros(NNZ)	1,603,498,863
Matrix Density(%)	0.1541
Max Value	$8.90422e^{-05}$
Min Value	$5.50416e^{-24}$
Max NNZ in a row	6537
Min NNZ in a row	0
Avg NNZ in a row	1208
Most repeating NNZ in row	0
Occurrence of Most repeating NNZ in row	822530
2nd Most repeating NNZ in row	3
Occurrence of 2nd Most repeating NNZ in row	2034
Max NNZ in a column	6404
Min NNZ in a column	0
Avg NNZ in a column	2045
Most repeating NNZ in column	0
Occurrence of Most repeating Mode NNZ in column	215488
2nd Most repeating NNZ in column	231
Occurrence of 2nd Most repeating NNZ in column	260

Table 3.1.: MADPET-II Characteristics

In the scanner *MADPET-II*, the voxels are divided into a *140 X 140 X 40* grid and the gamma rays are detected by *1152* detectors arranged in two concentric rings[3]. The table 3.1 sheds light on some of the properties of the PET scanner *MADPET-II*. Even when the matrix is stored in the sparse format the size of the system matrix is *approximately 12.8 gigabytes* . It also shows that the configuration of the scanner leads to the creation of the matrix with *greater than 1.3 million* rows and almost *800,000* columns. The other important things to notice are the difference between maximum and minimum values in the matrix *(*$8.90422e^{-05}$ *vs* $5.50416e^{-24}$*)* which could lead to floating point errors, the difference between the maximum and average values in the rows *(6537 vs 1208)* which affects performance in *SPMV* and the difference between maximum and average values between the columns *(6404 vs 2045)* which affects performance in *SPMV_T*. Figure 3.1 shows the distribution of

the non zeros in the system matrix A.

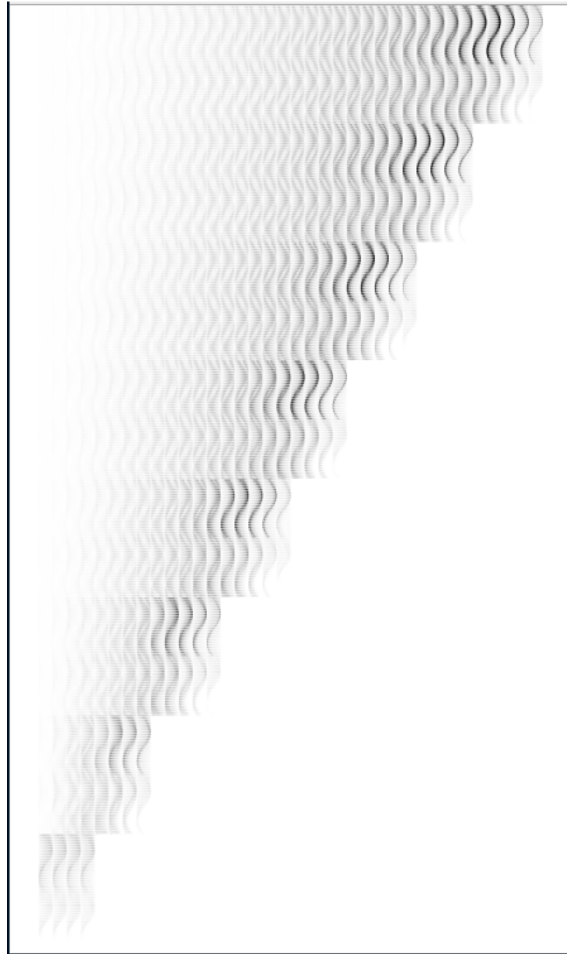


Figure 3.1.: Visualization of system matrix for MADPET-II.[3]

4. Sparse Matrix Format, SPMV and SPMV_T

The system matrix *A*, with its *1327104* rows, *784000* columns and the sparsity of *0.1541%*, if stored in a dense format the matrix would have required approximately 3800 gigabytes (with values stored as int and float) which would have been very difficult to handle. This would have put a lot of unnecessary load on the I/O not to mention the memory and computational resource wastage.

As an example consider the addition of all the values of the matrix. If stored in the dense format it would have required $\approx 1e^{12}$ operations instead of just $\approx 1.6e^{9}$ operations if only the non zeros are stored.

Due to the practical issues already mentioned, it becomes imperative that we consider "smart" formats for storing these really big matrices. So the format that has been used to store the system matrix for the PET scanner *MADPET-II* is a sparse format called *Compressed Sparse Row(CSR)*.

$$\left\{ \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 2 & 3 & 0 & 0 \\ 0 & 4 & 5 & 6 \\ 0 & 0 & 0 & 0 \end{array} \right\}$$

Table 4.1.: Full Matrix

CSR Rows (IA)	0	1	3	6	6	
CSR Columns (JA)	2	0	1	1	2	3
CSR Value (A)	1	2	3	4	5	6

Table 4.2.: Matrix in CSR format

The table 4.1 shows a typical matrix stored in a dense format while the table 4.2 shows a matrix stored in the *CSR* format.

The *CSR* format is a row major format i.e *left-to-right top-to-bottom*. In this format the matrix is stored in three different one dimensional arrays. The *CSR* Rows in the table 4.2 represents the number of non zeros in each row and it is of the size of *number of rows + 1*. [9]

The array *CSR* Columns stores the column number of each non zero element of the matrix and hence it is of the size equal to number of non zero elements in the matrix. The *CSR* Value array stores all the non zero values of the matrix and hence its size is also same as the size of *CSR* Columns i.e equal to number of non zero elements in the matrix.

To access the values and columns associated with a row, first the number of non zeros in that row *i* are calculated by subtracting *IA[i]* from *IA[i +1] i.e. (IA[i+1] - IA[i])*. Now the columns where these non zeros exist and the values can be accessed starting from the index number *IA[i]* in the *CSR* Columns and *CSR* Value arrays up-to the number of non zeros in that particular row.

The two types of operation performed with the system matrix are *Sparse Matrix Vector Multiplication(SPMV)* and *Transpose Sparse Matrix Vector Multiplication(SPMV_T)* and the other operations, like the calculation of norm 3.2 are derivative of these which will also be touched upon later.

4.1. Sparse Matrix Vector Multiplication(SPMV)

Sparse Matrix Vector Multiplication or SPMV, is the name given to a set of problems involving the multiplication of a sparse matrix, *CSR* here, with a dense vector and hence the name. The memory and computational requirements are substantially reduced by the use of SPMV instead of dense matrix vector multiplication but the use has its own performance pitfalls, especially when using multicore systems.

Source code 4.1 shows a concise way to store the *CSR* matrix.

```
struct CSR_MATRIX{
        float num_rows;
        float num_columns;
        float nnz;
        std::vector<float> csr_vals;
        std::vector<int> csr_rows;
        std::vector<int> csr_cols;
        CSR_MATRIX(int nRows, int nColumns, int NNZ){
                num_rows = nRows;
                num_columns = nColumns;
                nnz = NNZ;
                csr_vals.resize(nnz);
                csr_cols.resize(nnz);
                csr_rows.resize(nRows + 1);
        }
};
```

Source Code 4.1.: A possible way to store matrix in Compressed Sparse Row (CSR) format.

```
    CSR_MATRIX csr(nRows, nColumns, NNZ);
    std::vector vec(csr.num_columns);
    std::vector <float> spmv(csr.num_rows, 0);
    for (int i=0; i< csr.num_rows ; ++i)
{
        int nnz_this_row = csr.csr_rows[i+1] - csr.csr_rows[i];
        for (int j =csr.csr_rows[i] ;
            j< csr.csr_rows[i] + nnz_this_row; ++j)
        {
            int column_num = csr.csr_cols[j];
            spmv[i] += csr.csr_vals[j]*vec[column_num];
        }
    }
```

Source Code 4.2.: Sparse Matrix Vector Multiplication (SPMV)

The source code 4.2 shows how to perform *SPMV* with a matrix stored in CSR format as shown in 4.1. The *vec* in 4.2 is the vector of size equal to number of columns of the matrix *(csr.num_columns)* that has to be multiplied with the *CSR* matrix. For any row *i*, in line 6 we calculate the number of non zeros elements in the row and in line 7 we loop over those number of elements starting from the element number stored at the ith place in the *csr_rows* array. Finally in line 9 we find out the column number of the value pointed to by the inner loop and subsequently multiply the value from the *CSR* values array and the corresponding value from the *vec* array and add it into the output result(*spmv* array) of row *i*. The size of the generated output array is equal to the number of rows in the matrix.

4.2. Transpose Sparse Matrix Vector Multiplication(SPMV_T)

The next system matrix operation is the multiplication of the transposed sparse matrix with a vector. *SPMV_T* could also be looked as the matrix stored in column major format and then performing the *SPMV* over it. The problem with the second approach is that it is not always possible to store the matrix in row major and column major format side by side due to memory limitations. So it becomes very important that *SPMV_T* operation is also written in a way that the memory accesses are reduced to minimum.

The code 4.3 shows a possible way, for the matrix stored in *CSR* format(4.1), that is transposed and multiplied with a vector of size equal to number of rows of the matrix. The only change, apart from the changes in size of the input and output arrays in line 2-3, is that now the vector to be multiplied i.e. *vec* is accessed based on the row number and the result is saved in the output array (line 10) corresponding to the column number instead of

the row number, as was the case in *SPMV*. The output array generated is of the size equal to the number of columns of the matrix.

```
CSR_MATRIX csr(nRows, nColumns, NNZ);
std::vector vec(csr.num_rows);
std::vector <float> spmv_t(csr.num_columns, 0);
for (int i=0; i< csr.num_rows ; ++i)
{
    int nnz_this_row = csr.csr_rows[i+1] - csr.csr_rows[i];
    for (int j =csr.csr_rows[i] ;
        j< csr.csr_rows[i] + nnz_this_row; ++j)
    {
        int column_num = csr.csr_cols[j];
        spmv_t[column_num] += csr.csr_vals[j]*vec[i];
    }
}
```

Source Code 4.3.: Transpose Sparse Matrix Vector Multiplication (SPMV_T)

5. GPU Architecture and Programming

The *Graphical Processing Units(GPU)* were initially used to generate 2 or 3 dimensional images and videos that have to be outputted on the display device, be it operating system or games. The need to design a new processing unit arised because the generation of the images require a high degree of data parallel computations, where there are lot more arithmetic operations compared to memory operations, and *Central Processing Unit(CPU)* with their very few cores, especially at the start of the century, were not ready to handle the workload. This requirement finally lead to the development of devices which could handle highly parallel compute intensive workloads where most of the transistors are devoted to the data processing instead of data caching and flow control where *CPU* has the lead. [10, 11]

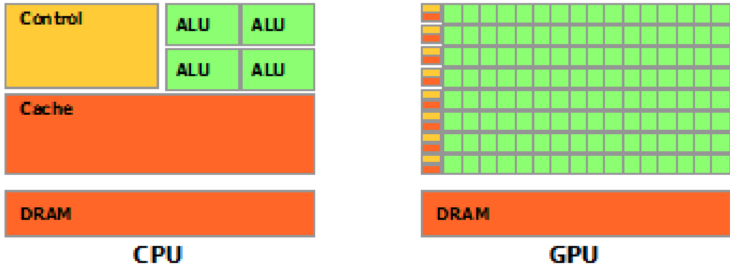


Figure 5.1.: Central Processing Unit (CPU) vs Graphical Processing Unit (GPU)[10](NOTE: Not to scale.)

The Figure 5.1 shows how the area on a *CPU* and *GPU* die is split into different components. In the *CPU* a lot of die area (percentage wise) is used for cache and control logic whereas in the case of *GPU* most of the area (percentage wise) is used for compute units. This is the reason why the *CPU* manufactures like Intel could add a lot of new instruction sets (AVX, SSE, MMX etc). The *CPU's* could also run at a much better clock frequency and are far ahead of *GPUs* in speculative execution, branch prediction, and store forwarding. [12]

Even when the *GPUs* were only used for the graphical applications and there was no easy way to communicate or program the device for the non graphics developers, it did not deter few brave souls, who understood the power of the GPUs to try to use them for their

application using the graphics APIs like *Direct3D* and *OpenGL*. Due to these limitations the learning curve was steep and the full potential of the *GPUs* could not be utilized.

This all changed with the release of *Compute Unified Device Architecture (CUDA)* for GPUs by *NVIDIA* in 2007. For the first time the non graphical developers had an API that enabled them to use *GPU* for general purpose processing. It provided them a software layer that gives access to the *GPU's* virtual instruction set and parallel compute units, for the executions of compute kernels. The CUDA was designed to work with the languages C, C++ and FORTRAN but now it could also be used from other languages like Java and Python due to the wide availability of wrappers.

The *CUDA* programming model was designed to allow developers to leverage the multiple cores in the *GPU* to develop transparently scalable software without the hassle of learning the graphical programming as was the case before. *CUDA* provided three abstractions at its core; hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions allowed the developer to partition the problem into coarse sub problems that can be solved simultaneously and independently by a group of threads in a block where the threads could solve it cooperatively by breaking the sub problem into even finer pieces. [10]

The chapter tries to give an overview to the architecture of the *CUDA* capable *NVIDIA GPUs*, the programming model along with a few optimization techniques that could accelerate the performance of the *SPMV and SPMV_T* operations.

5.1. Grids, Blocks and Threads

The threads in a *CUDA* kernel can be arranged in 1D, 2D or 3D and subsequently are called one, two or three dimensional thread block respectively. This idea is very helpful as it helps in mapping a vector, matrix or volume domain logically across the threads. But it should be kept in mind here that the number of threads in a thread block could not exceed 1024 as all the threads are supposed to reside on the same streaming multiprocessor and are expected to share the limited resources of that streaming multiprocessor. So some of the possible configurations for the thread block could be (1024,1,1) or (16,8,8).

The blocks are further arranged in 1D, 2D or 3D and subsequently are called one, two or three dimensional block grid respectively. The grid size can far exceed the number of available thread blocks that a *GPU* can process simultaneously as it is dependent on the size of data being processed.

Since the dimension of the thread block and block grid could be 2D and 3D apart from linear (which could be handled with data type int also), NVIDIA has introduced a new

data type called *dim3*. The possible example for the 1D block is (1024,1,1) for 2D is (32,32,1) and for 3D is (16,8,8). Now to calculate the global thread index of a thread, *CUDA* provides access to three variables called *blockIdx, blockDim and threadIdx*. With the help of variable blockIdx the index of a block could be calculated locally in a grid like blockIdx.x for the first dimension and so on. The variable blockDim tells the size of the grid like blockDim.x for the first dimension of the block and so on. And finally the variable threadIdx tells the user about the local thread number in a block. Using these three variables the global index of the thread could be calculated as shown in 5.1 for a 2D grid and 2D block.

```
1 int i = blockIdx.x * blockDim.x + threadIdx.x;
2 int j = blockIdx.y * blockDim.y + threadIdx.y;
```

Source Code 5.1.: Calculating local thread id in a 2D thread block in a 2D block grid.

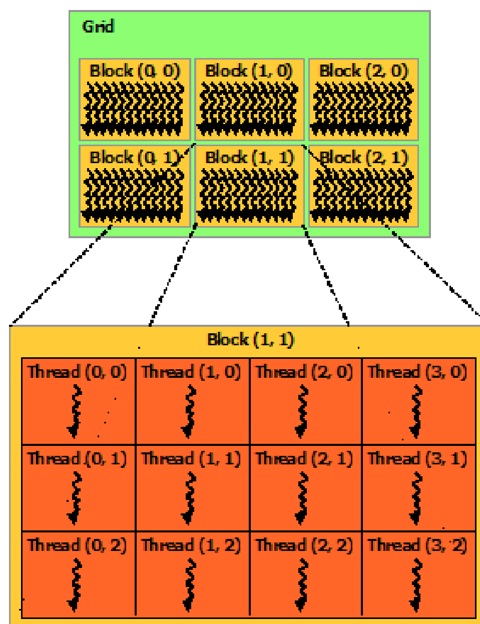


Figure 5.2.: Grid of Thread Blocks[10]

The figure 5.2 gives a visual representation of the nesting of a 2D block in a 2D grid where the size of block grid is (3,2,1) while the size of thread block is (4,3,1).

5.2. Register, Local, Constant, Shared, Global, and Texture memory

CUDA capable devices have a memory hierarchy where each type of memory has its advantages and disadvantages and selecting the correct memory to save the data could have significant effect on the performance of the kernels.

- Register: The fastest of all the memories but it is only accessible by the thread and has the lifetime of the thread.
- Local: This is not a physical type of memory but an abstraction of global memory. It has the same scope as registers. Since this memory resides off chip, the access to it is very slow and it is only used in the case when registers are out of space.
- Constant: This memory is accessible by all threads but is read only. It has the lifetime of an application.
- Shared: This memory has the lifetime of a block and let the threads in a block share data among themselves. It is also a fast memory only when there are no bank conflicts.
- Global: This memory is the largest and the slowest at the same time. It has the lifetime of an application and is accessible by all the threads in the application.
- Texture: This is another type of read only memory which is accessible by all threads in the application. This memory is really helpful in applications where there is a lot of spatial locality.

The figure 5.3 shows the different type of memory which reside on the *GPU*. Apart from that it also tells that the global, constant and texture memory are the only ones accessible from the *host*.

The table 5.1 lists the amount of different types of memory available in *NVIDIA Tesla P100* (*NVIDIA Pascal Architecture based GPU in DGX-1*).

Memory Type	Size
Global	16 GB
Register/SM	256 KB
Register/GPU	14336 KB
Shared/SM	64 KB
Shared/GPU	3584 KB
Constant	65536 Bytes
Texture	As large as Global

Table 5.1.: Different memory size in NVIDIA Tesla P100[13]

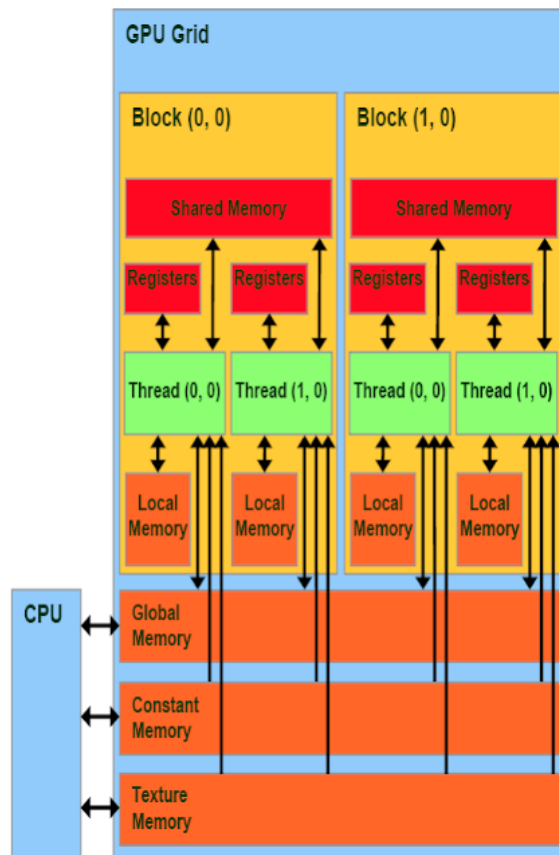


Figure 5.3.: Memory Hierarchy on GPU [14]

5.3. Host, Device and Global Kernels

These are the function space execution specifiers which tell the compiler which function has to be executed on the device and which on host apart form whether it is callable from device or from host.

- __device__: This execution space identifier declares that the function is to be executed on the device i.e. GPU and will be callable from the device only.
- __host__: This execution space identifier declares that the function is to be executed on the host and is only callable from the host. If the specifier is omitted then the compiler assumes that the function has to compiled for the host i.e. *CPU*.
- __global__: This execution space identifier declares that the function is to be executed on the device and will be callable from the host and from the device (With *CUDA* capable devices having compute capability higher than 3.2). A __global__ function must have a void return type and cannot be a member of class. Moreover the call to a __global__ function is asynchronous.

These specifiers could also be used together but all the combinations are not possible. For example the *__global__/__device__* and *__global__/__host__* specifiers could not be used together while the *__device__/__host__* can be.[10]

5.4. Streams and Events

The *CUDA* stream represent a sequence of operations which are executed in the order they are issued. The steams are the work queues to express concurrency between different tasks. The streams are particularly helpful to make use of the *GPU* fully. For example, on one stream a kernel execution could be launched while on the other stream memory transfer from host to device and from device to host could be performed(this will also depend on the available resources: two copy engines, compute resources).

CUDA events are synchronization markers that are used to synchronize the tasks from different streams, allow fine grained synchronization within a stream and finally to allow inter leave synchronization.

5.5. Asynchronous Kernels and Copy Operations

Asynchronous operations are those which does not block the process for the finishing of the operation. They do not wait for the return value and hence it is the job of the developer to make sure that the operation has completed before using the result. Now these operations, although requiring more work, are very helpful in overlapping two different kind

of work, for example the overlapping of communication and computation in the multi-threaded applications.

In the context of *CUDA* programming some of the operations are already asynchronous like the launch of *__global__* kernels by default. The control, in this case, instantly return to the host and the developer is required to explicitly add the synchronization calls (depending on the situation because the use of other synchronized operation ,like *cudamemcpy for copying the data between host and device*, will wait for all the previous operations on the device to finish before executing). In the *CUDA* kernels the asynchronized implementation of the operations like *cudamemcpy (cudamemcpyasync)* are also available and the use of them helps in hiding the latency. But the use of asynchronous operations, especially *cudamemcpyasync*, requires for a bit more extra work, apart from the explicit synchronization, and that is to allocate the memory on the host as pinned instead of pageable and also to define the stream in which the asynchronous operation has to be performed.

5.6. Atomic Operations

In the context of concurrent programming, the atomic operations are operations that happen in complete isolation without being interrupted. The operations are guaranteed to be performed without the interference from other threads.

CUDA provides a few atomic functions that work on the global and shared memory. The important thing to note here is that the use of the atomic operation only guarantees that access to the memory is serialized i.e. one thread at a time will have access to that memory but it is not guaranteed in which order the threads will have access to the memory. [10]

The atomics in the Fermi and Kepler architecture from NVIDIA had been implemented from the software side and they used to work by the lock/update/unlock sequence. But from Maxwell architecture onwards, the 32-bit atomics were implemented natively with the use of specialized hardware and the 64-bit atomics were implemented by compare and swap (CAS) approach. From Pascal architecture onwards the atomics could also be performed system wide instead of only the gpu the kernel is running on. [13]

Some examples of atomic operations available in *CUDA* are *atomicAdd*(for atomic addition) and *atomicSub*(for atomic subtraction).

5.7. Error Checking

The *CUDA API* have return functions which can be used to check errors during the execution of the functions. Source code 5.2 shows the function that has been used in the thesis to

figure out the errors in the *CUDA* kernels. The function accesses the previous *CUDA* error and prints out the information like what the error is and in which line it occurs and finally it exits the code.

```
string prev_file = "";
int prev_line = 0;
void cuda_check(string file, int line)
{
cudaError_t e = cudaGetLastError();
if (e != cudaSuccess)
{
    cout << endl << file << ", line " << line << ": "
    << cudaGetErrorString(e) << " (" << e << ")" << endl;
    if (prev_line>0) cout << "Previous CUDA call:"
    << endl << prev_file << ", line " << prev_line << endl;
    exit(1);
}
prev_file = file;
prev_line = line;
}
```

Source Code 5.2.: Error checking function used in the thesis.

Since the launch of the kernels is asynchronous they do not return any error code but that does not mean we can't check for errors. *CUDA* provides two functions called *cudaPeekAtLastError()* and *cudaGetLastError()* which could be used to get the value of error variable that *CUDA* maintains. The difference between these two functions is that the call to *cudaPeekAtLastError()* return the value of the error variable while the use of *cudaGetLastError()* get the value of that variable and sets the variable to *cudaSuccess*.

5.8. Coalesced Memory Access

To be able to completely understand the idea of coalescing we first have to understand concept of warps in *CUDA*. So every kernel that is launched on the device is executed by a *Streaming Multiprocessor(SM)*. There are many of these SM in each device (the number is 56 in Tesla P100 and each has SM has 64 *CUDA* cores).

So when a kernel is invoked, the developer have to specify the blocks and threads in each block and each of these blocks have to be mapped to a *SM* to execute. All the *SM* can share the resources available to that *SM* (like the 64 KB shared memory available to

each SM in Tesla P100). But the *SM* tries to further divide the block internally so that each thread share the same code and follow the same execution path with minimal divergence and stall at the same point in the kernel. The thread group generated because of this further splitting by the *SM* is called a warp and the current maximum size of it is 32 threads for *NVIDIA GPUs*.

There are definitely some disadvantages of this approach also especially when different threads take different execution path which leads to the under-utilization of the hardware resources.

The global memory in the device is accessed via 32,64 or 128 byte memory transactions. Now if these memory are aligned perfectly with the size of the warp and warps execute a global memory access instruction, it tries to coalesce into as less memory transactions as possible because every global memory access instruction induces a penalty on the throughput of the device. So for example suppose that each thread in a warp (32 threads) has to access a float value (4 bytes) in the global memory and if the memory access is coalesced and the memory is contiguous then this transaction will be completed in a single step but it could go as bad as 32 transactions in the worst case. [15]

```
1  __global__ void mat_vec_mul(  float* result,
2                                 const float* vec,
3                                 const float* csrVal_d,
4                                 const int* csrRowInd_d,
5                                 const int* csrColInd_d,
6                                 const uint32_t rows)
7  {
8      uint32_t i = threadIdx.x + blockIdx.x*blockDim.x;
9
10     if (i<rows){
11         uint32_t col_num;
12         for (uint32_t j=csrRowInd_d[i]; j< csrRowInd_d[i+1] ; ++j){
13             col_num = csrColInd_d[j];
14             result[i] += vec[col_num]*csrVal_d[j];
15         }
16     }
17 }
```

Source Code 5.3.: Example of a non coalesced global memory access in SPMV kernel.

The code 5.3 shows an example where the coalescing of the global memory access is not implemented in the *SPMV* kernel and the code 5.4 shows the example when it is implemented. The performance of these two kernels have been compared in the Part II of the thesis.

5.9. Pinned Memory

The memory is called pinned because after being allocated it could not be swapped out from the system memory, for example to the swap partition on the hard drive. In other words once it is given a place in the host memory then it could not be evicted. This type of allocation leads to improved *I/O efficiency* but on the other side the allocation of the pinned memory takes a bit more time compared to the pageable memory. Moreover the allocation of too much pinned memory actually degrades the performance of the code so it should be used with care. [17]

In *CUDA* the pinned memory on the host could be allocated with the help of two functions namely, *cudaHostAlloc()* and *cudaHostMalloc()*. These two functions doesn't have any difference when *cudaHostAlloc()* is used in the default mode.

The final implementation is using the pinned memory because of the inherent compulsion to use it when using asynchronous functions in the *multi GPU* case. The pinned memory is further allocated with the option *cudaHostAllocPortable* because then the memory is considered pinned for all the contexts and not just the current context.

5.10. Bank Conflicts

Shared memory is a type of on-chip memory and hence it has much lower latency and much higher bandwidth. To achieve even better bandwidth the shared memory available per block is divided into multiple equally size memory modules called banks (there are 32 banks in the modern SM which is equal to the warp size). The memory accesses by the threads to these different banks could be handled at the same time (and thus the bandwidth increases by the factor equal to number of simultaneous memory requests) but the bank conflicts occur when more than 1 thread tries to access the same memory bank. The hardware handles this by splitting the requests into as many conflict free requests as possible which is ultimately responsible for bandwidth penalty. [18, 19]

```
__global__ void mat_vec_mul_warp (  const uint32_t nnz_to_skip,
                                    const int rows ,
                                    const int* csrRowInd_d ,
                                    const int* csrColInd_d ,
                                    const float* csrVal_d ,
                                    const float* vector ,
                                    float* result)
{
    __shared__ float vals [1024];

    uint32_t thread_id = blockDim.x * blockIdx.x + threadIdx.x;

    // thread index within the warp
    uint32_t thread_lane = threadIdx.x & (WARP_SIZE-1);
    // global warp index
    uint32_t warp_id = thread_id / WARP_SIZE;
    // total number of active warps
    uint32_t num_warps = (blockDim.x / WARP_SIZE) * gridDim.x;
    // one warp per row
    for ( uint32_t row = warp_id; row < rows ; row += num_warps)
        {
            uint32_t row_start = csrRowInd_d [ row ];
            uint32_t row_end = csrRowInd_d [ row +1];
            // compute running sum per thread
            vals [ threadIdx.x ] = 0.0;
            for ( uint32_t jj = row_start + thread_lane ;
                jj < row_end ; jj += WARP_SIZE){
                vals [ threadIdx.x ] += csrVal_d[jj-nnz_to_skip]*
                    vector [ csrColInd_d [ jj -nnz_to_skip]];
            }
            // first thread writes the result
            if ( thread_lane == 0){
                for (int i =1 ; i<WARP_SIZE ; i++)
                    vals[threadIdx.x] += vals[threadIdx.x + i];
                atomicAdd(&result[row], vals[threadIdx.x]);
            }

            __syncthreads();
        }
}
```

Source Code 5.4.: Example of a coalesced global memory access in SPMV kernel.[16]

6. Other Terminologies

There are three more terminologies that have been used in the thesis and thus they warrant a brief introduction and this chapter is about those three terms.

6.1. Heterogeneous Computing

Heterogeneous Computing refers to the systems in which there are more than one kind of processors working together to solve a task and each one of them is specialized for certain kind of tasks. An example of that is the use of *GPU* along with the *CPU* to solve a problem as is the case in the present thesis. This approach can be scaled further to create a network of these heterogeneous systems where there could be one or more co-processors i.e *GPU* connected to a single host. The *MAC cluster* in *LRZ* is based on this very same idea.

In *heterogeneous computing* it is not always necessary that the host and the co-processors are all of the same type and this introduces additional difficulties in workload balancing.

The thesis is an attempt to harness the power of the host and co-processor in the best possible way to solve the *MLEM algorithm* as fast as possible especially when there is not enough memory on the GPU to store the entire matrix at once

6.2. CUDA Aware MPI

MPI stands for *Message Passing Interface* and it is a communication protocol for communicating data between distributed processes through messages. It is written with scalability for multi node systems in mind. Now the *non cuda aware MPI* implementation could be used easily to communicate data in the multi core heterogeneous systems but it involves a bit of extra work to achieve that and also leads to performance hit. Due to the above mentioned reasons it is a good idea to combine the parallel programming approaches of CUDA and MPI.

To start a multi core distributed code, *MPI* requires that the user provides the number of processes the application should scale to. Now to exchange data between the processes (where some calculation has been performed by the *GPU*), in the *non cuda aware* mode the data from the *GPU* has to be copied back for the host to send and when the host receives the data then that has to be again copied into the *GPU* explicitly. Now this approach requires

4 commands to achieve instead of just 2, if we use the *cuda aware MPI*. The example 6.1 and 6.2 shows this difference.

```
//on MPI rank 0
cudaMemcpy(h_send_buf, d_send_buf, size, cudaMemcpyDeviceToHost);
MPI_Send(h_send_buf, size, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);

//on MPI rank 1
MPI_Recv(h_recv_buf, size, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,&stat);
cudaMemcpy(d_recv_buf, h_recv_buf, size, cudaMemcpyHostToDevice );
```

Source Code 6.1.: Example of communication between processes when the MPI is not CUDA aware.

```
//on MPI rank 0
MPI_Send(d_send_buf, size, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);

//on MPI rank 1
MPI_Recv(d_recv_buf, size, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,&stat);
```

Source Code 6.2.: Example of communication between processes when the MPI is CUDA aware.

The feature works because of the *Unified Virtual Addressing(UVA)* introduces with *CUDA 4.0* (for GPU from Fermi architecture onwards). The *UVA* allows the memory of host and all the devices connected to it in a system, to be treated as a single virtual address space. In the absence of *UVA*, the *MPI* would have to be told where the memory lives i.e on host or on device, which could have been achieved automatically or with an addition argument in the *MPI* commands. The figure 6.1 shows how the *UVA* maps all the memory in a system to a single virtual address space. [20]

The use of *CUDA aware MPI* accelerates the communication by pipelining all the operation in the message transfer and also by the use of acceleration technologies like *GPUDirect* implemented incrementally by NVIDIA from Kepler architecture onwards. An example of the *GPUDirect* technology, figure 6.2, is the *GPUDirect P2P*, which is used to transfer the memory from one *GPU* to another without being staged through the host during intra node communication. This approach leads to a higher bandwidth and low latency communication between *GPUs*.

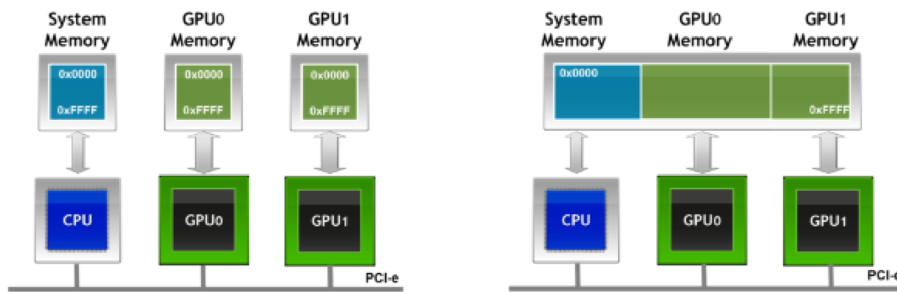


Figure 6.1.: Mapping of memory in a system with UVA(right) and without UVA(left)[20]

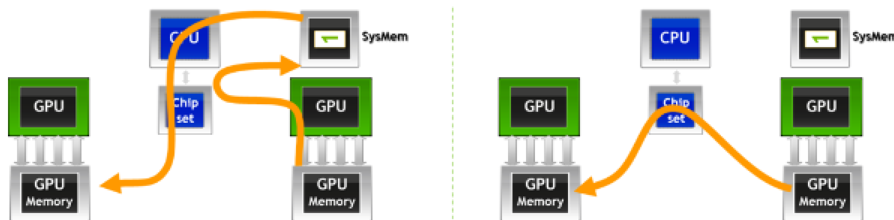


Figure 6.2.: The communication without the use of GPUDirect P2P (left) and with GPUDirect P2P (right)[20]

Even in the *GPUs* with Fermi architecture, which do not have any of the *GPUDirect* technologies implemented, the use of *CUDA Aware MPI* leads to faster communication due to the fact that is pipelines the communication and reduces some bottlenecks.

6.3. Floating Point Arithmetic

This section will try to give a very brief overview of the what floating point numbers are, how they are represented into the computers, IEEE-754 standards and finally the difference between results that are encountered during the course of the thesis and the possible reasons for those differences.

The floating point numbers are called so because in these numbers the binary/decimal point can float over the significant digits based on the exponent. For example the 1.2345 with its 5 significant digits can be represented as $1.2345x10^0$ or as $12345x10^{-4}$.

The representation $1.2345x10^0$ is also called the *normalized scientific notation*. In the *normalized scientific notation* a floating number is represented as $d_0.d_1d_2d_3...d_{n-1}\ x\ \beta^e$. In this representation $d_0.d_1d_2d_3...d_{n-1}$ are called the n significant digits, β the base and e the exponent. In this representation it is also required that the absolute value of $d_0.d_1d_2d_3...d_{n-1}$

should be greater than 1 and less than 10 (for the representation in base 10). [21]

The modern day computers are made up of billions of transistors and those transistors have only 2 states i.e 0 and 1. This is the reason why the floating point numbers in the computers are represented in base 2. Although it is possible to simulate the decimal numbers with binary circuits as well but it leads to less efficient implementation.

The need to have a floating point standard like *IEEE-754* arised because in the earlier days of computer era the vendors were implementing floating point arithmetic in their own way. For example IBM was using some guard digits to perform the floating point arithmetic exactly and then rounding of the result. This lead to the situation where programs were not giving same result on different machines which in some applications could have cascading effects. The *IEEE-754* standard released in the year 1985 was an attempt to solve this reliability and portability problem by laying down the rules for representation of floating point numbers, their interchangeability, rounding errors, exception handling and operations. [22]

The floating point number in the *IEEE-754* is represented by 32 bits in base 2. The 32 bits are divided into 3 parts. The first 23 bits are allocated for the significant digits, the next 8 are saved for the exponent and the last one for the sign of the number. With this representation the values in the range $\approx -1.4e^{-38}$ to $\approx 3.4e^{38}$ could be stored as floating point numbers. [23]

Out of the four basic arithmetic operations, addition, subtraction, multiplication and division, the use of addition and subtraction in the case where one number is very large and another one require special attention. The reason for that is that the while adding or subtraction floating point numbers, *IEEE-754* require that the exponents are matched and then the fraction part is added up. Now this operation has the unintended effect of the dropping out of the smaller of the two completely (in the worst case) because of the limited number of available bits to save the fraction part.

For example, the summation of $1.0e^{0}$ and $1.0e^{-8}$ in 32 bits would give an output of $1.0e^{0}$ which is not correct if looked from the perspective of exact arithmetic. The table 3.1 shows that the values in the system matrix are also quite far apart and hence these kind of errors could also crept there and they did. The interesting thing to note here is that even the result from the CPU and the GPU, when both use *IEEE-754* standards, were slightly different, *7108.830078 vs 7109.272461* when taking the sum of all the non zero values in the matrix.

```
1  #ifndef __STDC_IEC_559__
2  #error "Requires IEEE 754 floating point!"
3  #endif
4
5  #include <iostream>
6  #include <cblas.h>
7  #include <cublas_v2.h>
8  #include <cuda_runtime.h>
9
10 int main(int argc, char *argv[]){
11         size_t array_size = 1e8 + 1;
12         float *array = (float*) malloc(array_size*sizeof(float));
13         array[0] = 1;
14
15         float seq_sum = 0.0;
16         for (int i=0; i<array_size ; ++i) seq_sum += array[i];
17
18         float cpu_blas_sum = 0;
19         cpu_blas_sum = cblas_sasum(array_size, array, 1);
20
21         cublasHandle_t cublasHandle;
22         cublasCreate(&cublasHandle);
23         float* array_d;
24         cudaMalloc((void**)&array_d , array_size*sizeof(float));
25         cudaMemcpy(array_d , array , array_size*sizeof(float),
26                         cudaMemcpyHostToDevice);
27
28         float gpu_blas_sum = 0.0;
29         cublasSasum(cublasHandle, array_size, array_d, 1,
30                         &gpu_blas_sum);
31
32         delete [] array;
33         cublasDestroy(cublasHandle);
34         cudaFree(array_d);
35
36         std::cout << "seq_sum:" << seq_sum << std::endl;
37         std::cout << "cpu_blas_sum:" << cpu_blas_sum << std::endl;
38         std::cout << "gpu_blas_sum:" << gpu_blas_sum << std::endl;
39         return 0;}
```

Source Code 6.3.: Code to study the effect of summing up an array using different methods.

```
seq_sum: 1
cpu_blas_sum: 1.75
gpu_blas_sum: 1.99968
```

Source Code 6.4.: Result of the source code 6.3

In the code 6.3, a float array of size $1 + 10e^8$ is created and the first address is filled with value $1e^0$ and rest all of them with $1e^{-8}$ and then the array is summed up in 3 ways. In the first case the array is summed up sequentially, in the second case the *BLAS* library is used and lastly the array is summed up on the GPU using *CUBLAS library by NVIDIA* and the results are presented in 6.4. The correct result is *2* and as evident from 6.3 the GPU gives the closest result.

The possible reason for the discrepancy in the results is because when adding sequentially, the first value is big enough that all the other values becomes insignificant during addition due to the use of *floats*. The reason for better result when using the *BLAS* library on the CPU and *CUBLAS* on the GPU is that the summation now happens with *reduction* and hence some of the smaller values are added up and they become large enough to become significant with respect to the larger value before being added into it. But still the GPU result is more accurate possibly because of the fact that the array is divided into even smaller pieces for reduction to make better use of large number of cuda cores.

Part II.

Experiments and Results

7. Benchmarking MAC Cluster

The purpose of doing this test was two-fold. One, to gauge the performance of *Fermi Architecture* based GPUs (M2090) in Mac cluster vs *Pascal Architecture* based GPUs available in DGX-1. Secondly, the default MPI implementation available in Mac Cluster is from *Intel* and so it was also interesting to see how other implementations perform on it. So, the other MPI implementation used is *OpenMPI*.

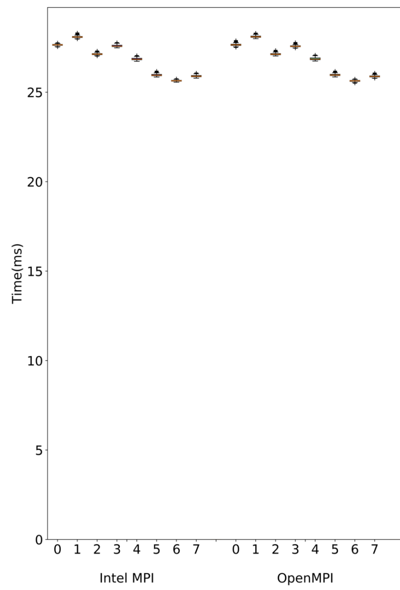
For this test all the 4 nodes (2 sockets per node) available in the cluster were used. Only 1 task per socket was launched and the processes were bound to the core. For each MPI implementation the application was launched 50 times and in each run 10 iterations were performed.

The figure 7.1a and 7.1b shows that the GPUs are solving the SPMV and SPMV_T operation at almost the same speed across processes. The slight variation in performance could be explained by the fact that the matrix is divided between the processes equally with respect to non zero values. Now in the figure 3.1 it could be seen that the non zeros are more in initial rows and they subsequently decrease in later rows. So this leads to the case where the initial ranks have more non zeros per row and later ranks have more rows for same number of non zeros.

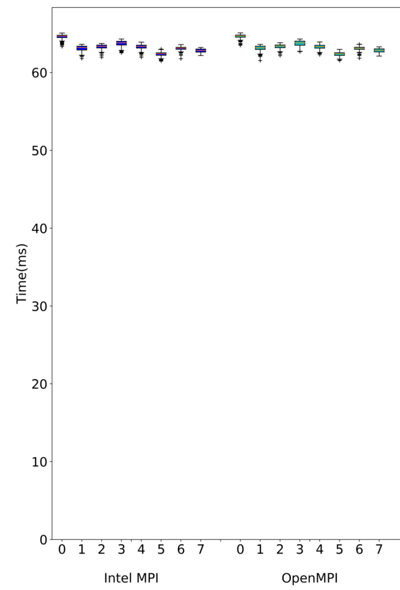
The "U" shape created by the later ranks in figure 7.1a could be probably because when the non zeros per row falls, it leads to less computation per thread and the device has to launch more thread blocks which brings with it some overhead.

The $\approx$ *28ms* it takes to perform the Forward Projection on the Mac Cluster is *7 times* slower than the time it takes to perform the same operation on DGX-1 (Chapter 10). The things gets even more interesting, *64.6 ms vs 7.3 ms*, in the case of Backward Projection. Here the speedup is almost *8.9 times* and this effect could be explained by the presence of *hardware support for atomics* in the *Pascal Architecture* based GPUs which was absent in Fermi based GPUs (Backward projection uses lot more atomics compared to Forward Projection).

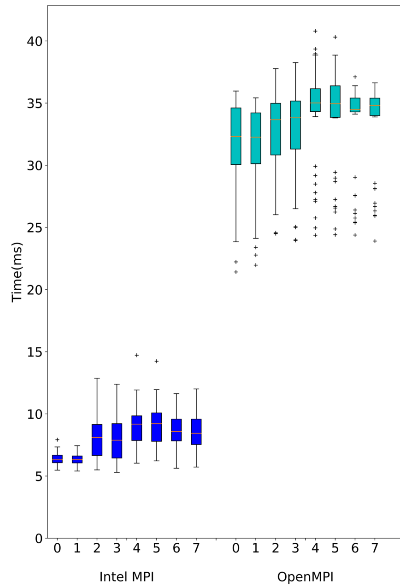
The figures 7.1c and 7.1d presents the time it takes to perform the reduction operation at different stages in the application. After the norm calculation, the *MPI_Allreduce* operation is performed for the first time and here the OpenMPI takes a lot more time compared to the Intel MPI but once that is over and when again the reduction operations are performed during the iterations, the OpenMPI edges slightly ahead of the Intel MPI.



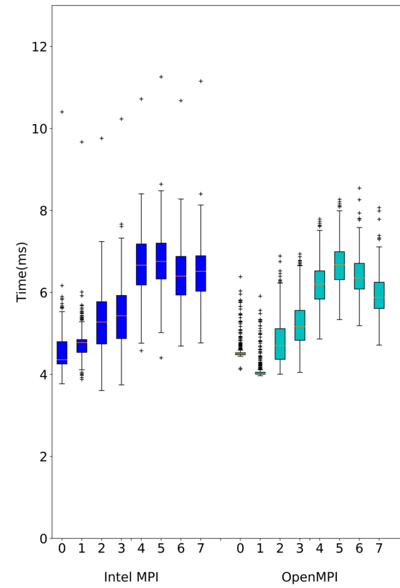
(a) Time to perform Forward Projection (SPMV).



(b) Time to perform Backward Projection (SPMV_T).



(c) Time taken to reduce the norm calculated.



(d) Time taken to perform reduction in the loop.

Figure 7.1.: Mac cluster benchmarking results.

8. Effect of Cuda Aware MPI, Pinned Memory and Custom Kernels

The final MLEM implementation was not achieved in a single attempt. There were few iterations where different path to achieve a objective was implemented, studied and finally the best possible was chosen. This chapter show the effect of some of those implementations on the performance on different kernels.

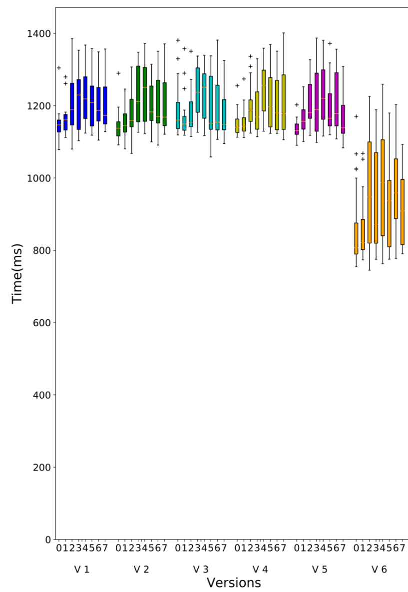
The table 8.1 shows the difference between different implementations. For example in the version 1,2 and 3 the only difference is whether the reduction is implemented with or without *CUDA AWARE MPI* and if not then whether the memory for it was pinned or not. The differences between version 5 and 6 are that the version 6 is *not CUDA AWARE* and it uses pinned memory for the reduction, it is also the only one which allocated *pinned memory* for the *CSR arrays* and finally the last difference is between the block size used for the SPMV and SPMV_T kernels. The block size in the case of the version 5 is *1024* and in the case of version 6 it is *64*.

Change\Versions	V 1	V 2	V 3	V 4	V 5 (1024)	V 6 (64)
Cuda Aware	NO	NO	YES	YES	YES	NO
Pinned Mem for Reduction	NO	YES	N/A	N/A	N/A	YES
Pinned Mem for CSR	NO	NO	NO	NO	NO	YES
CuSparse SPMV	YES	YES	YES	NO	NO	NO
CuSparse SPMV_T	YES	YES	YES	NO	NO	NO
Coalesced Mem Access SPMV_T	NO	NO	NO	NO	YES	YES
Non Coalesced Mem Access SPMV_T	NO	NO	NO	YES	NO	NO
Coalesced Mem Access SPMV	NO	NO	NO	NO	YES	YES
Non Coalesced Mem Access SPMV	NO	NO	NO	YES	NO	NO

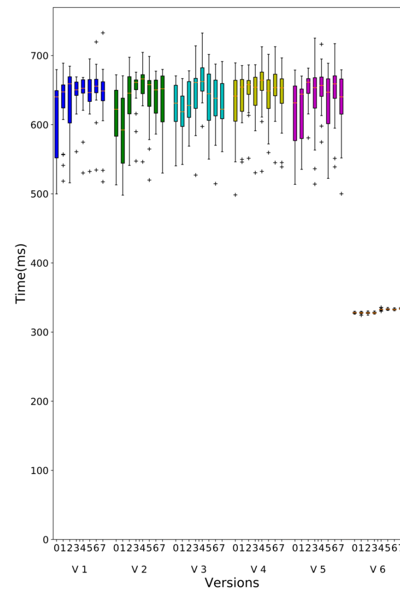
Table 8.1.: Differences between implementations.

The test was performed on the *DGX-1* with 8 MPI processes. Each MPI processes was allocated a GPU and the MPI processes were *bound to core*. The data for each version was collected by running it 20 times and each time 10 iterations were performed.

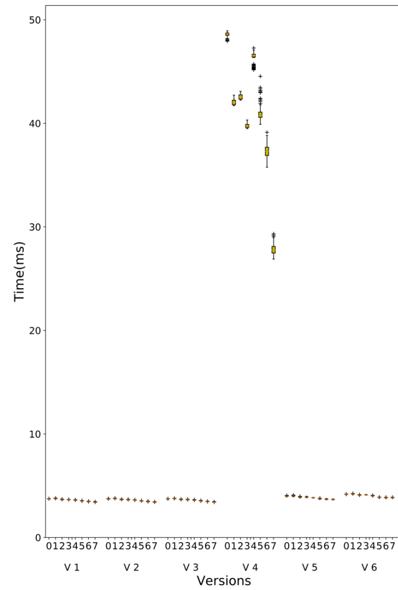
The figure 8.1a tells that it takes a considerably less time to load the data and convert it



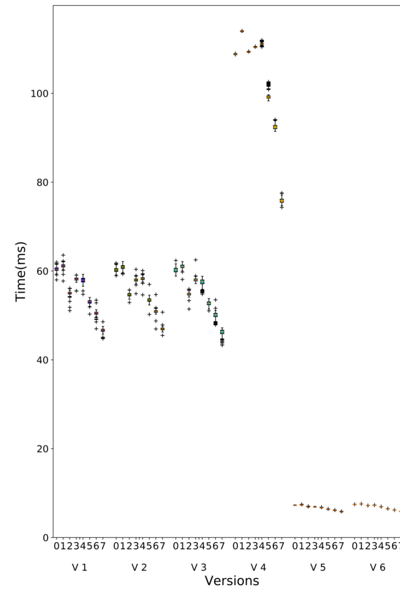
(a) Loading and converting the data into CSR format.



(b) Allocation of memory on the device and copying the data into it.



(c) Performance of SPMV kernel in different implementations.



(d) Performance of SPMV_T kernel in different implementations.

Figure 8.1.: Performance difference between implementations.

into the CSR format, if pinned memory is used for it. The possible reasons for the wide variations between the timing for the same version are the I/O limitations because all the processes will try to read the data at the same time and so some will read it faster than the others. The other reason for the variation could be from how the experiment was performed. For performing the experiment the MPI processes were bound to the core but they were free to be allocated on any core of the dual socket DGX-1.

The figure 8.1b shows that the time to send the data to the GPU is much faster if the pinned memory is used and the variation between the timing is also very less in that case.

The forward projection time in different implementations, figure 8.1c, shows that the custom SPMV implementation using coalesced memory access and the Cusparse implementation provided by NVIDIA are very close in terms of performance but the same can't be said about the non coalesced version implemented in the case of version 4. Moreover, there is not much variation between timing in the implementations, apart from non coalesced memory access version, and the reason for that is due to the non zeros in the rows that are allocated to each MPI process.

The backward projection time for different implementations is shown in figure 8.1d. It shows the custom kernel written using coalesced memory access is faster than the SPMV_T implementation provided by NVIDIA. The possible reason for this behaviour could be that the kernels provided by NVIDIA are guaranteed to provide bit wise same result and there is no efficient way to achieve that yet in SPMV_T. Here again the performance hit because of not using coalesced memory could be seen in version 4.

The figure 8.2 shows the time it takes to perform reduction operation reduces when pinned memory is used but it increases when the *CUDA AWARE MPI* is used. The possible reason could be because of the fact that the MPI was not built considering the underlying communication network which might degrade the performance. But still it warrants further scrutiny. The reason for the variation between processes stems from the SPMV_T calculation difference from earlier in the iteration loop, figure 8.1d, and hence they should be viewed together. The process which finishes the Backward Propagation faster has to wait longer at the reduction step.

The figures 8.3a and 8.3b provides a closer look at the performance difference when the block size for the kernels is changed from (1024,1,1) in implementation 5 to (64,1,1) in implementation 6. It could be seen that unexpectedly there is a performance hit going from version 5 to version 6. It was expected that the performance will improve with this change as the kernel allocates a warp to a row and so when the row is big it stalls the entire block and going for a smaller block was supposed to alleviate this and improve performance.

All the performance tests in later chapters have been done with block size of (1024,1,1).

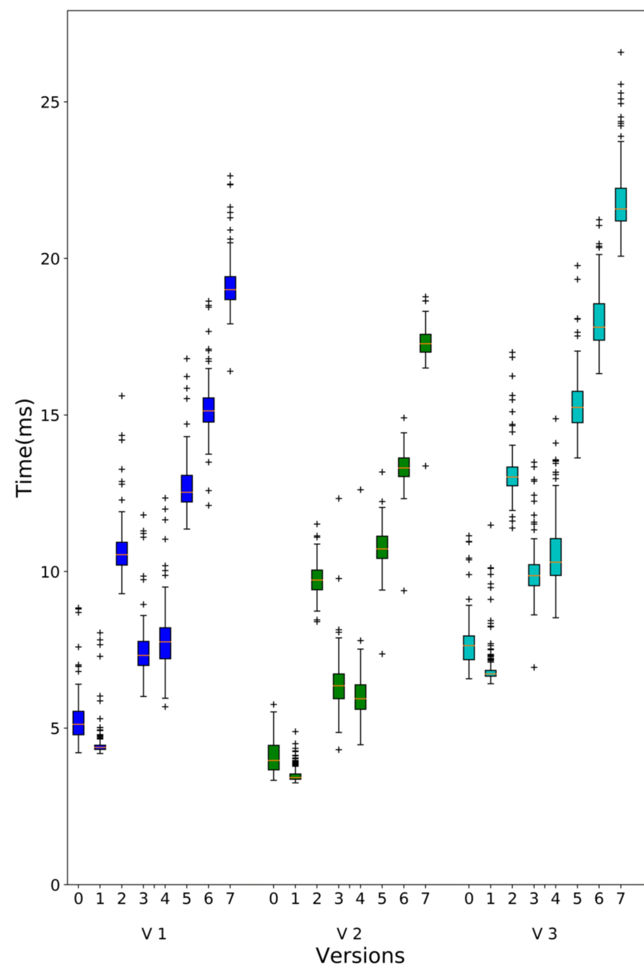
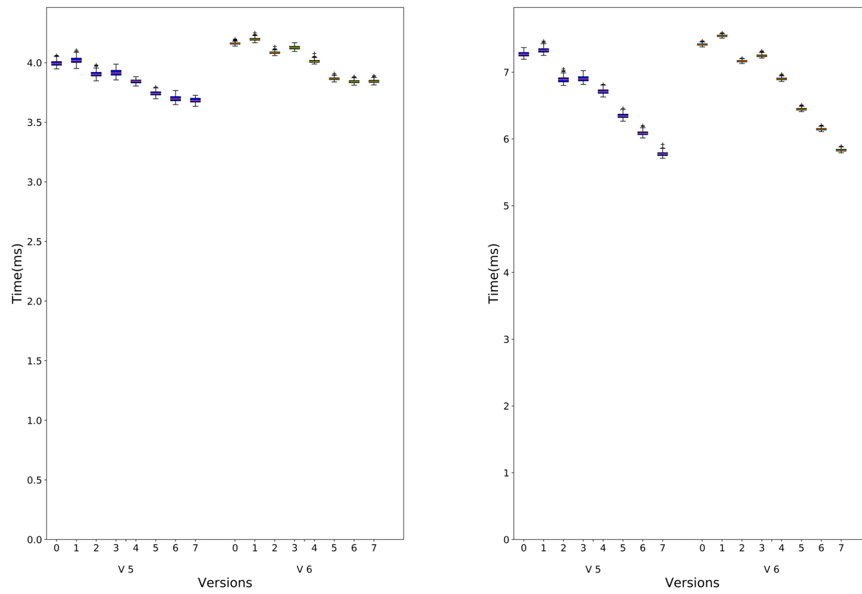


Figure 8.2.: Reduction operation performance in different implementations.



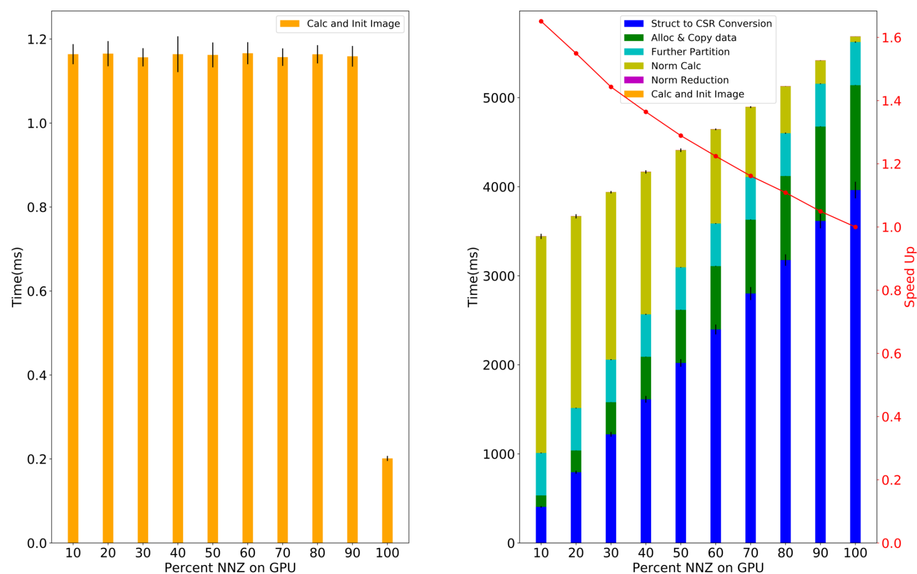
(a) Closer look at the Forward Projection.

(b) Closer look at the Backward Projection.

Figure 8.3.: Performance difference when the block size for the kernels is changed.

9. Effect of Splitting Work between CPU and GPU

This test was performed to see the effect of how the problem scales when more and more part is solved on the GPU. For this test the DGX-1 machine was used and the application was launched with only 1 MPI process. The application was bound to the *core* and the mapping was done per *core*. For solving the part on the CPU this application uses *2 Openmp* threads, which are on the same core. The tests were performed 50 times with 10 iterations for each distribution of work between host and device.



(a) Time to calculate initial value and then initializing image vector.

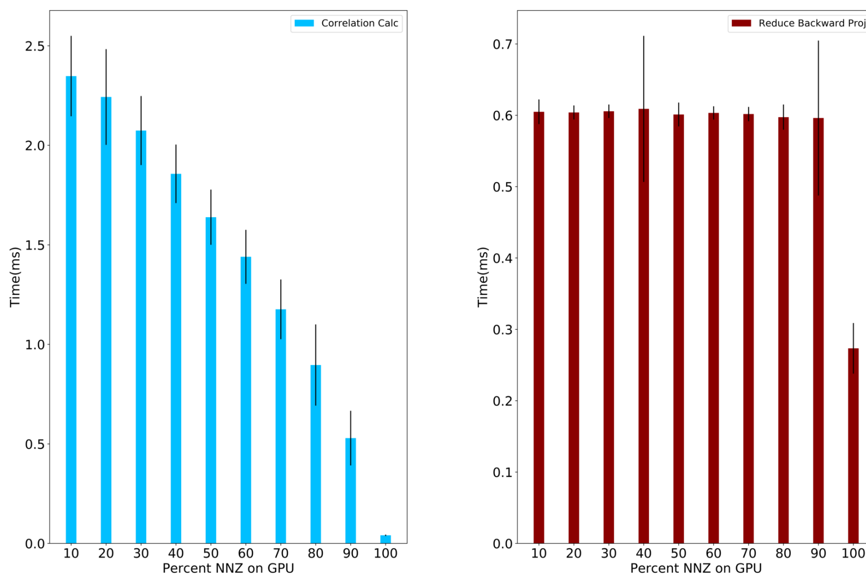
(b) Total time for Allocation and Initialization Operations.

Figure 9.1.: Time to perform Allocation and Initialization Operations.

The figure 9.1a shows the difference between the time it takes for initializing the *image*

vector of size *784,000* floats on the CPU and on the GPU. Before initialization of the image, the *initial* value is calculated by performing 2 *BLAS* operation on the GPU. The drastic drop when only using GPU is because now there is no need for a *Image* vector on the host. It is also important to note here the image initialization on the host is also accelerated by *OpenMP*.

In the figure 9.1b the effect of the increasing workload allocation to the device with respect to the total allocating and initializing vectors time could be seen. The *Norm Calculation*, SPMV_T operation where the vector is a unit vector, time reduces as we increase the percentage allocation on the device which is exactly what is expected.



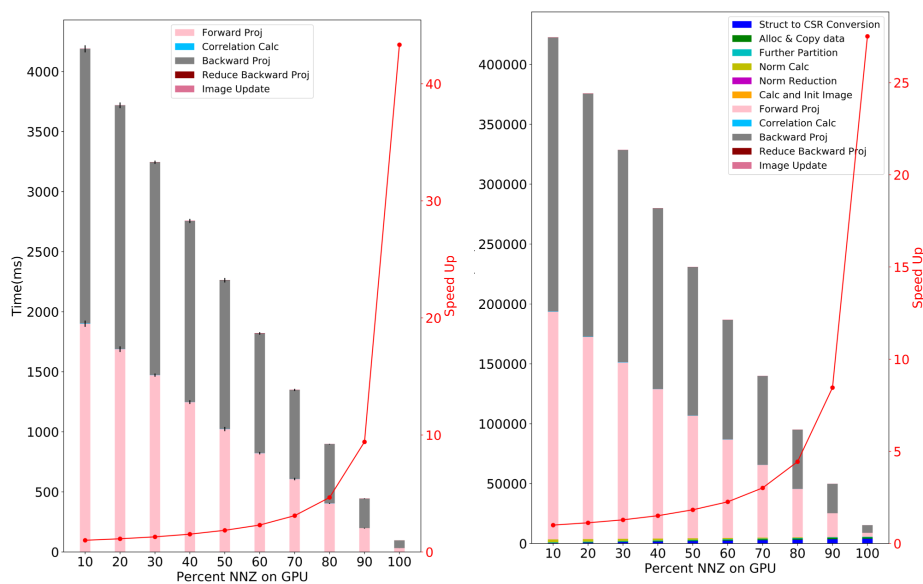
(a) Time to calculate correlation. (b) Time to reduce Backward Proj.

Figure 9.2.: Effect of increasing device workload on Correlation calculation and MPI_Allreduce time.

After partitioning the system matrix, the application has to determine which part has to be solved on the host and which on the device and this splitting is performed in *Further Partition*. Again the constant partition time is what is expected from the application.

As more and more part is allocated to the device, more of the initial data has to be converted into the CSR vectors, more data has to be copied on the GPU and hence the

s



(a) Time to perform a iteration.

(b) Total application runtime.

Figure 9.3.: Total time to complete a iteration and total runtime of application with 100 iterations.

overall time for allocation and initialization should increase. The same effect could also be seen in the figure 9.1b.

The correlation operation is a scaling step, equation 3.5, and from the figure 9.2a it could be deduced that the GPU is much more capable in accomplishing it compared to a CPU with 2 OpenMP threads.

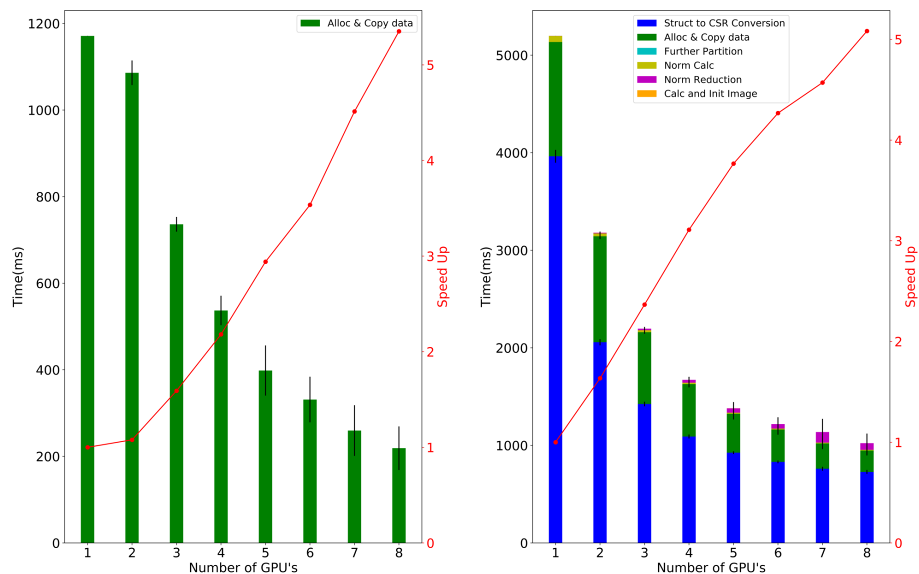
In the reduction step, the data is copied from *device to host*, the result from the host is added, then the *MPI_Allreduce* operation is performed and finally the result is copied back into the device. Now when all the workload is on the device, the addition of result from the host need not to be performed and hence it should make this operation faster. The exact same effect could be seen in the figure 9.2b.

In the figure 9.3a, it could be seen that the total iteration time reduces proportionately from *10 to 90 percent* but when only using device it drops few folds. This shows the edge the GPU hold over the CPU in this computation.

The figure 9.3b sheds light on how the use of even 1 GPU could reduce the computation time by many folds. It is also interesting to note that when all the computation is happening on the GPU, the allocation and initialization time is comparable to the time it takes to perform 100 iterations. This effect could also be seen in the profiling results shown in chapter 13.

10. Performance vs Number of GPU

The purpose of this test is to determine the scaling of the implementation when the number of MPI processes are increased from *1-8* with each connected to a *GPU*. This test was performed on *DGX-1*. The processes were bound to *core* and they were mapped by *L3cache*. The tests were performed *50 times* for each increase in MPI processes with 10 iterations. So the results presented in figure 10.1 are calculated over 50 data points and the results presented in 10.2 are calculated over 500 data points.



(a) Time to Allocate and Initialize CSR and measured data

(b) Total time for Allocation and Initialization Operations.

Figure 10.1.: Time to perform Allocation and Initialization Operations.

The possible reason why the perfect scaling is not achieved in the allocation and copying the data on the devices,figure 10.1a, is because of the way the test was performed. Since we mapped the processes to the *L3cache* which means that we only use one of the CPU

and since each CPU has only *two PCIe-3 x16 slots*, which means that the total double data transfer rate is 64 Gigabytes per second (GBps). Moreover, each *PCIe-3* slot is directly connected to 2 GPUs and so each CPU has direct connection to only 4 GPUs and for the rest of the GPUs the data transfer takes over *NVlink*. But we see the speedup as we increase the number of MPI processes because not each MPI process asks the *PCIe-3* slot to transfer the data at the same time and due to this staggering the data is loaded faster.

The figure 10.1b shows how the total time reduces as the number of MPI processes (with a GPU each) are increased from 1-8. The limitation on *I/O, i.e RAM and storage*, could be argued as the reason why the reading and conversion to CSR vectors of the system matrix (marked as blue) does not scale perfectly with the increase in number of MPI processes.

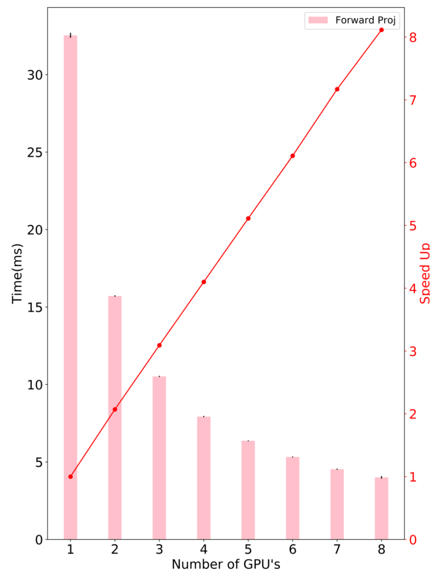
The figure 10.2a and figure 10.2b shows the effect on *Forward Projection(SPMV)* and *Backward Projection (SPMV_T)* calculation time with increasing number of GPUs. These two operation scales more than perfectly with the increasing number of GPUs and the reason is the increase in total available *atomic functional units*. It could also be argued that this increase might be coming from the increasing total memory bandwidth but that is unlikely. It is so because the total amount of data that has to be read and written in each kernel is *12838.6 + 5.3 + 3 = 12846.9 MegaBytes* and the forward and backward projection kernels for 1 GPU finishes in approximately *32.3 ms and 63.1 ms* and in *4.0 ms and 7.3 ms* for 8 GPUs respectively. This gives a effective bandwidth of *397.7 GBps* and *203.6 GBps* in case of 1 GPU which increases to *401.5 GBps* and *220.0 GBps* for 8 GPUs for forward and backward projection respectively while the theoretical maximum is *732 GBps*.

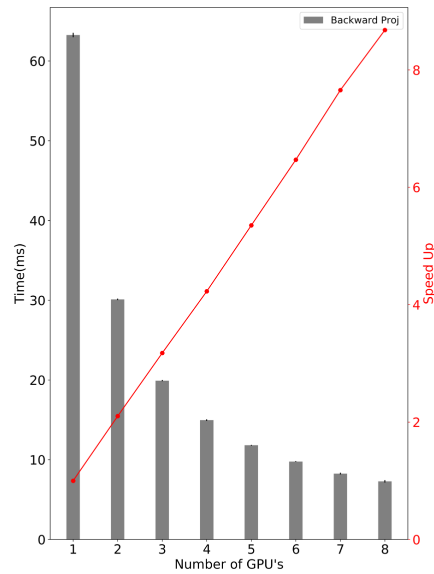
The effect of increasing number of *atomic functional units* is more pronounced in the case of backward projection because there are lot more total atomic operation in the backward projection, *1,327,104 vs 1,603,498,863* to be precise.

The figure 10.2c presents the typical behaviour of *MPI_Allreduce* operation with the increasing number of MPI processes, i.e. increasing operation time with increasing MPI processes.

Finally, the figure 10.2d shows the decrease in total runtime for a iteration with increasing GPUs. Even though the forward and backward projection scaled more than perfectly with increasing GPU, that effect is adversely affected by the *MPI_Allreduce* time in the overall iteration time.

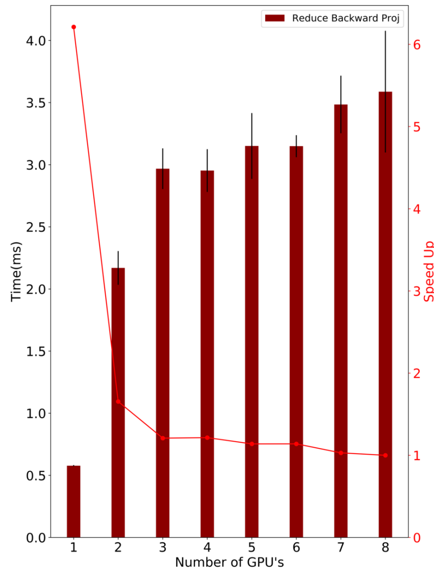
The figure 10.3 shows the total runtime of the implementation with 100 iterations and overall it scales very well but not perfectly with increasing GPUs. This figure also tells that the GPUs could be used to reduce the problem runtime proportionately but for that to happen, there should be lot more computation.

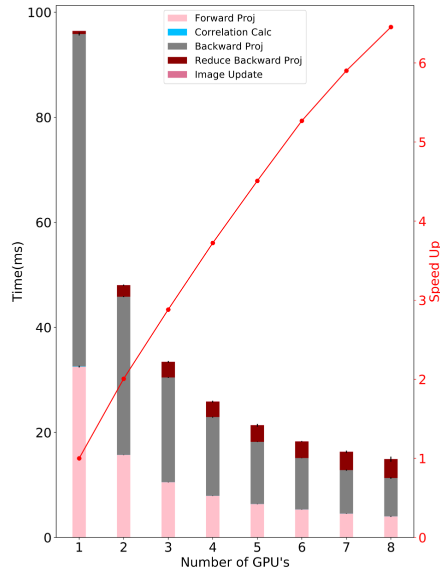




(a) Time to perform Forward Projection (SPMV)

(b) Time to perform Backward Projection (SPMV_T)





(c) Time to perform MPI_Allreduce after Backward Projection

(d) Total time taken to complete a single iteration.

Figure 10.2.: Time to perform operations in the iterations.

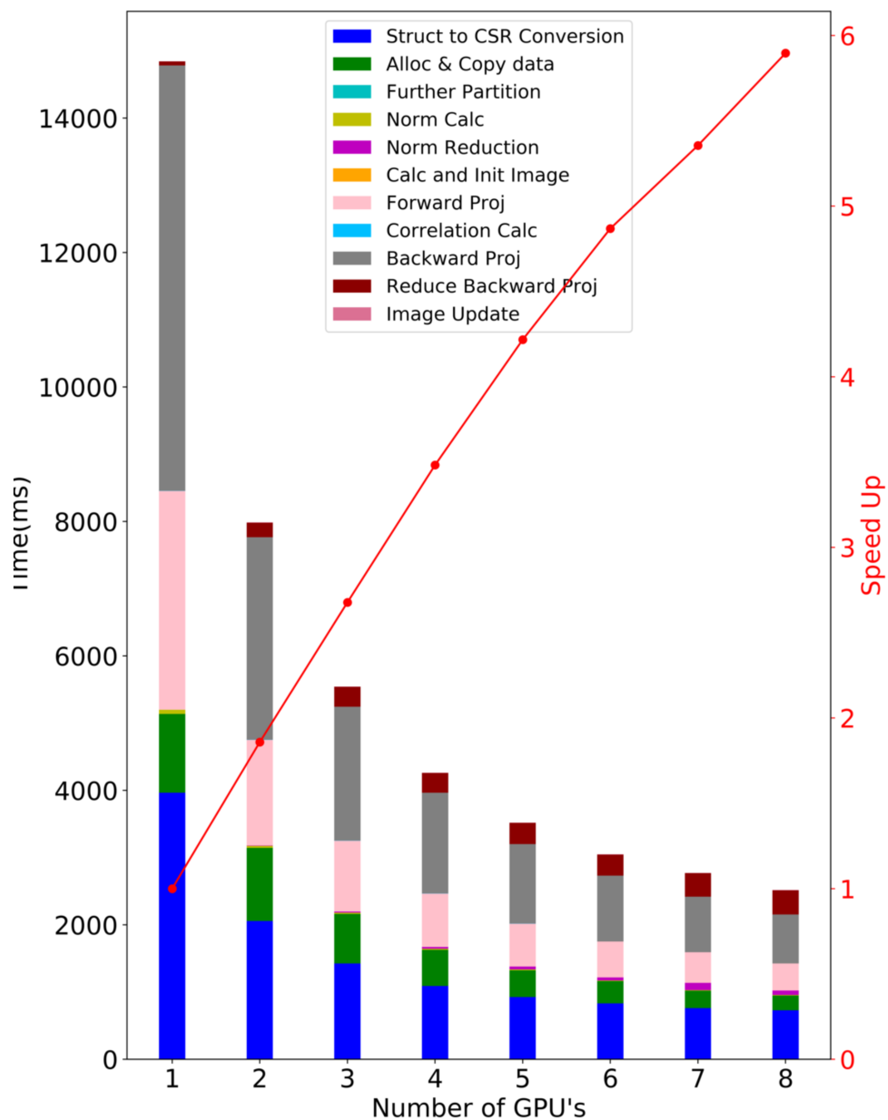


Figure 10.3.: Total runtime from start to end for 100 iterations.

11. Performance on CPU vs OpenMP Threads

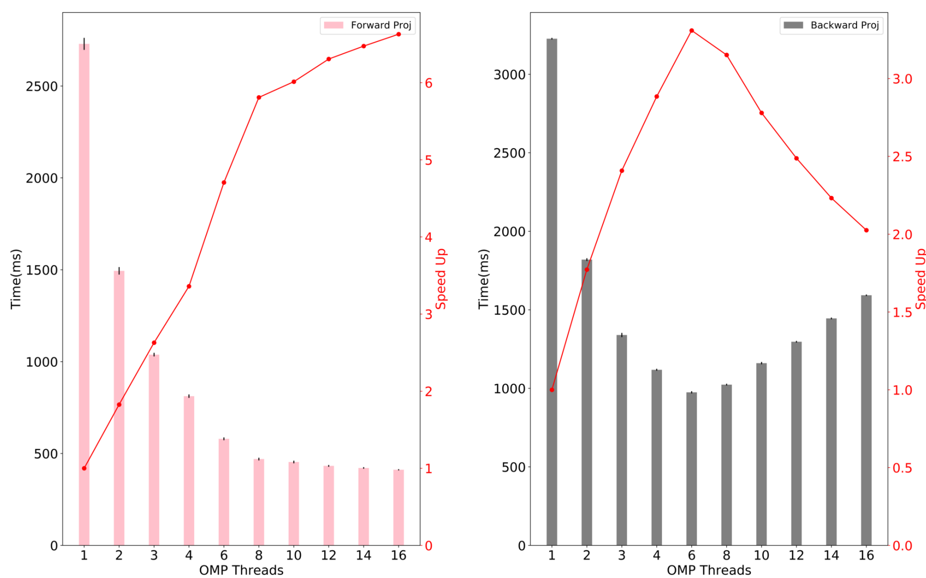
This test was performed to see the effect of how the implementation scales on the host when more and more OpenMP threads are provided. For this test, a node was allocated on the *Mac Cluster* and the application was launched with only 1 MPI process. The application was bound to the *hwthread* and the mapping was done per *L3cache*. The tests were performed 50 times with 10 iterations for each subsequent increase in OpenMP threads available to the host.

The figure 11.1a and 11.1b show the effect of increasing openmp threads for the computation of *Forward Projection (SPMV)* and *Backward Projection (SPMV_T)* respectively. In both the figures it could be seen that the increasing openmp threads have clear positive effect on the time it takes to complete the computation upto 8 threads, although not proportionally. The reason for not seeing the proportional increase is because all the threads are mapped by the *L3cache* and hence are using the same host resources like *Last Level Cache (LLC)*.

Now the reason for the very slow decrease in time taken to solve the *Forward Projection* and even increase in time taken to solve the *Backward Projection* after *8 Openmp* threads is due to the fact that now the *hyper-threading* of the host kicks in. After 8 Openmp threads, the subsequent threads are allocated on the same core which has already one thread running on it and due to this a lot of context switching takes place during the execution which ultimately degrades the performance. The effect is more pronounced in the case of *Backward Projection* because it uses a private array per thread to compute the *SPMV_T* compared to *none* for computing *SPMV* in *Forward Projection*, which degrades the cache performance. The codes 11.1 and 11.2 are the host implementation of *Forward* and *Backward* projection respectively to show the use of private array per thread.

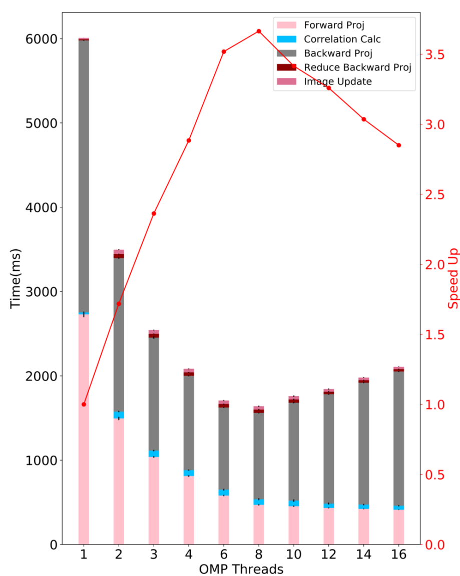
The figure 11.1c shows how the increase of openmp thread affects the performance of a iteration in total. In the figure it could be seen that the performance increase only till 8 openmp threads because after that the degradation in backward projection overwhelm the slight increase in performance for forward projection.

The table 11.1 shows the bandwidth achieved, per process and per thread, as the number of OMP threads are increased from *1 to 16* when launching the application with 1 MPI process only. It shows that in the case of Forward projection the bandwidth increases as we increase the OMP threads but falls with respect to per thread. In the case of Backward projection the bandwidth rises when the threads are increased from 1 top 8 but falls



(a) Time to perform Forward Projection (SPMV).

(b) Time to perform Backward Projection (SPMV_T).



(c) Total time taken to complete a single iteration.

Figure 11.1.: Time to perform operations in a iteration.

Operation(BW) \OMP Threads	1		8		16	
	Process	Thread	Process	Thread	Process	Thread
Forward Projection (GBps)	4.59	4.59	26.7	3.34	30.47	1.90
Backward Projection (GBps)	3.88	3.88	12.25	1.53	7.87	0.49

Table 11.1.: CPU bandwidth per MPI process and per thread as a factor of increasing OMP thread when launching application with 1 MPI process.

when increase from 8 to 16. In this case also the bandwidth per thread keeps falling as we increase the OMP threads.

```
#pragma omp parallel for schedule(dynamic)
for (size_t row=further_split_for_cpu.start ;
    row< further_split_for_cpu.end ; ++row){
        float res = 0.0;

        std::for_each(matrix.beginRow2(row), matrix.endRow2(row),
                        [&](const RowElement<float>& e){
                        res += e.value()*image_mask[e.column()];});
        correlation[row] = res;
    }
```

Source Code 11.1.: Forward projection on host.

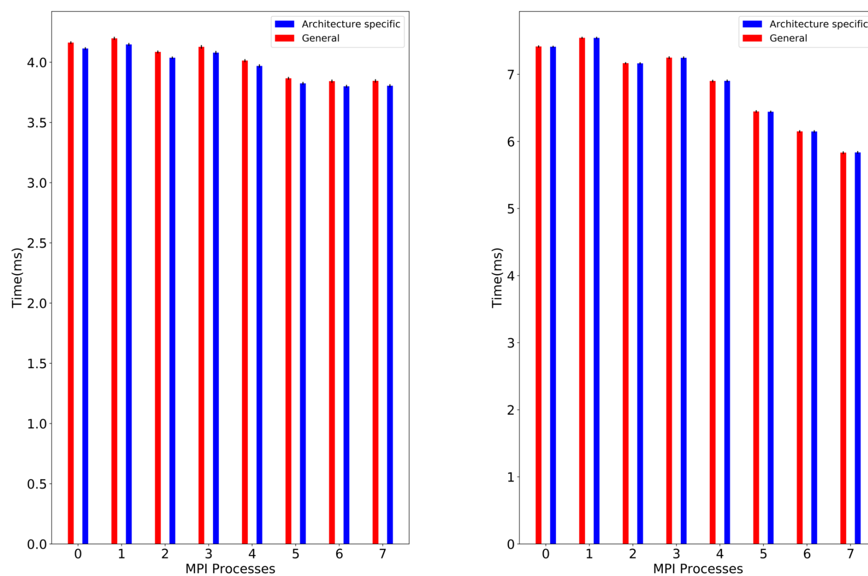
```
#pragma omp parallel{
    Vector<float> pri_update(update.size(), 0.0);
    #pragma omp for schedule(dynamic)
    for (uint32_t row=further_split_for_cpu.start;
        row<further_split_for_cpu.end; ++row){
        std::for_each(matrix.beginRow2(row), matrix.endRow2(row),
            [&](const RowElement<float>& e){
            pri_update[e.column()]+=e.value()*correlation[row];});
        }
    #pragma omp critical
    for (size_t i = 0; i < update.size(); ++i)
    update[i] += pri_update[i];
}
```

Source Code 11.2.: Backward projection on host.

12. Compilation for Specific Architecture

When compiling the project, the compiler *nvcc* could also be asked to compile the project for a specific architecture. This enable architecture specific optimization.

Since this test was performed on the *DGX-1* and the *Pascal* GPUs have the compute capability of *6.0*, the architecture specific optimization was enabled using the option *–gpu-architecture=sm_60*. The test was performed 20 times each with and without compiling the implementation for specific architecture. Each time the implementation was asked to perform 10 iterations.



(a) Forward Projection time.

(b) Backward Projection time.

Figure 12.1.: Performance difference with and without architecture specific compilation.

The figures 12.1a and 12.1b shows the effect of compiling the code with and without the architecture specific optimization for SPMV and SPMV_T respectively. In the forward pro-

jection the performance improves slightly when the code is compiled for the architecture but in the case of backward projection the difference is almost non existent.

13. Profiling using NVVP

This test was performed to gauge the performance of different kernels using the *NVIDIA Visual Profiler (NVVP)*. The test was performed with *1 MPI process*. The test was performed on a virtual server with *16 virtual core* and *1 Tesla P100 GPU*. For profiling, the application was run for 100 iterations.

The figure 13.1 shows the timeline generated for the implementation for 100 iterations. The interesting thing to note here that it takes only approximately *31.5 seconds* from start to end and out of that almost *16 seconds* are taken up by initialization steps. It can also be seen that since the memory was pinned, it takes around 1 second to free it.

The table 13.1 shows the occupancy for the kernels and the data transfer rate for the memory movement operations between host and device.

Kernel	*Performance*
CudaMemCpy (HtoD) Transfer Rate	11.3 - 11.6 GBps
CudaMemCpy (DtoH) Transfer Rate	10.4 - 10.7 GBps
SPMV_T kernel time	64.5 ms
SPMV_T kernel Occupancy	79.1 %
SPMV kernel time	31.6 ms
SPMV kernel Occupancy	98.9 %
Correlation Calculation kernel time	33 μs
Correlation Calculation kernel Occupancy	67.1%
Image update Kernel time	30 μs
Image update Kernel Occupancy	82.3 %
Cublas Sum time	11.5 μs
Cublas Sum Occupancy	82.3 %
Cuda MemSet time	864 ns
Cuda MemSet Throughput	6144 GBps

Table 13.1.: Performance of kernels during the iterations.

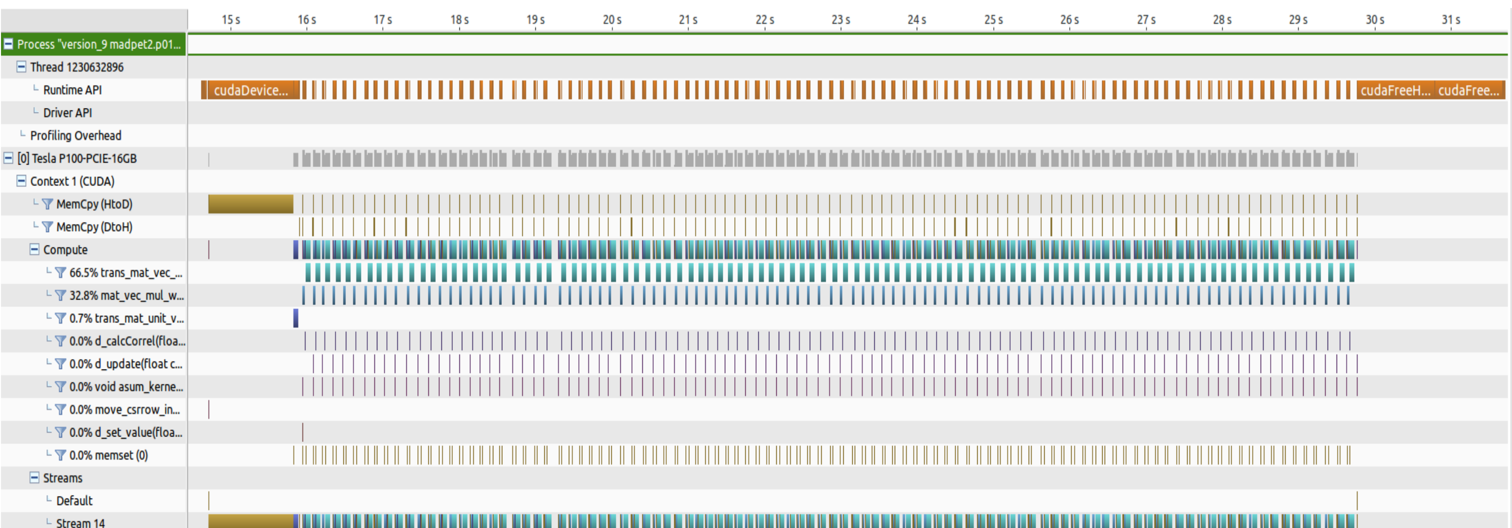


Figure 13.1.: Application profiling using NVVP for 100 iterations.

Part III.

Future Work and Conclusion

14. Future Work

14.1. Resource Aware Computing

The Figure 14.1 gives a visualization of present implementation of the MLEM algorithm done in the thesis. The implementation is able to run in parallel across multiple nodes with the use of MPI library. Each MPI process could also make use of the *none or many* non overlapping CUDA capable GPUs (the GPUs accessible by one MPI process should not be accessible by any other). Also when the entire work allocated to a MPI process could not be offloaded to the GPUs then the remaining work is solved on the host by making use of *OpenMP* to accelerate it.

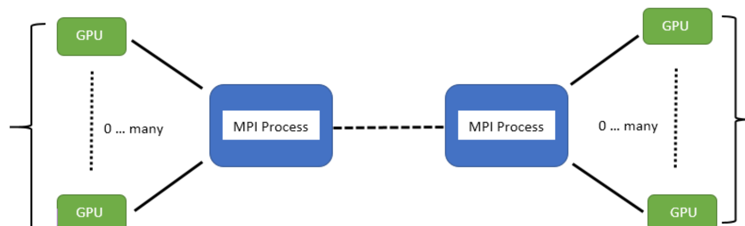
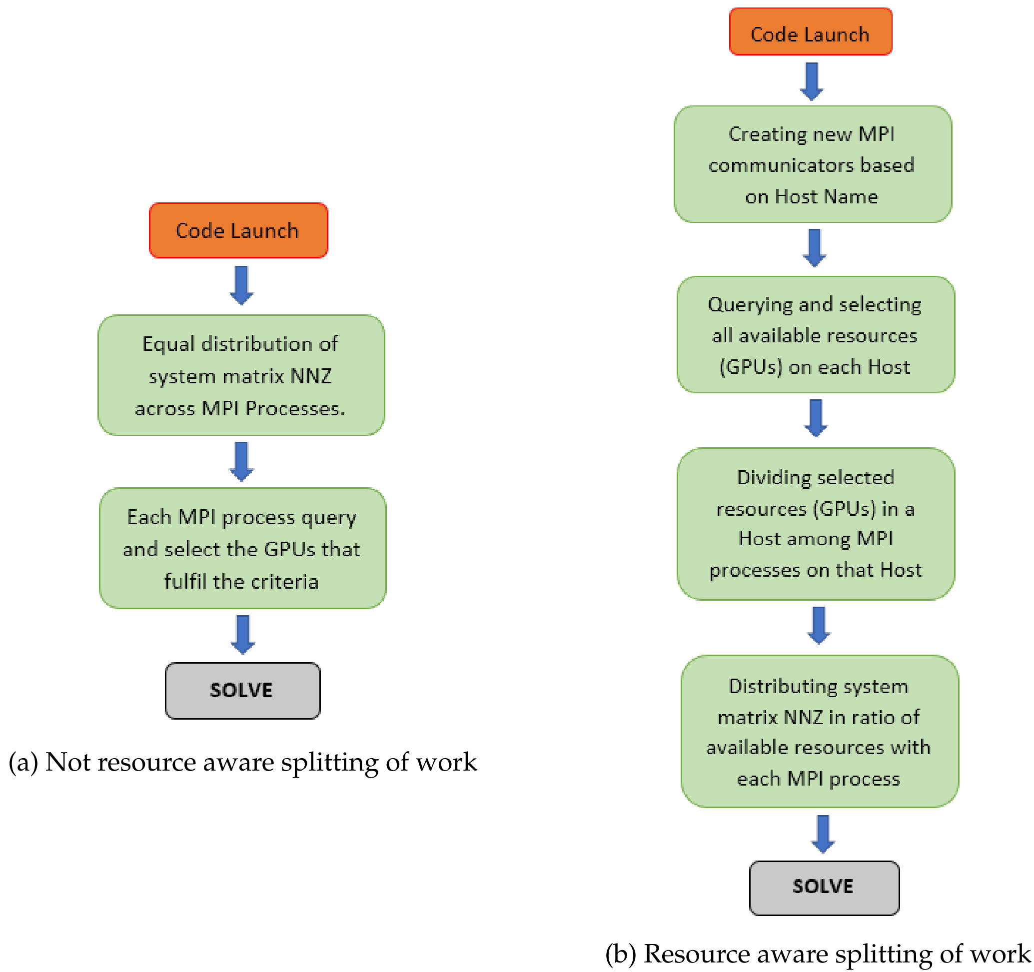


Figure 14.1.: Capability of the present implementation.

Figure 14.2a shows the present implementation of the splitting of work between the MPI processes. In the present case the system matrix is divided equally amongst each MPI process which has a huge drawback when even one of the MPI process does not have enough resources to solve it fast enough so that other processes does not have to wait. This implementation also could not make full use of the resources available in machines like DGX-1 because each MPI process could only handle upto 3 GPUs and launching more than 1 process will lead to overlapping of the GPUs amongst the MPI processes which in the worst case would lead to the implementation crashing.



(a) Not resource aware splitting of work

(b) Resource aware splitting of work

Figure 14.2.: Splitting of system matrix NNZs when the implementation is resource aware and when it is not.

All the above mentioned drawbacks could be ameliorated by using flowchart for splitting the work shown in figure 14.2b. In the resource aware case, the MPI processes make sub communicators based on their host-names and the MPI processes in each sub communicator splits the available resources amongst themselves. Now after deciding on the resources, each MPI process takes the chunk of the the system matrix that it can solve with the results being ready at almost the same time as all other processes. This implementation leads to much better load balancing, faster execution and optimal resource utilization.

For example, when the resource aware implementation is used on the completely idle DGX-1 machine, it would split the system matrix in the ratio of *3/8 : 3/8 : 2/8* or *2/8 : 2/8 :*

2/8 : 1/8 : 1/8 depending whether the implementation is launched with 3 MPI processes or 5 MPI processes.

14.2. Merge Based CsrMV

The merge based CsrMV has been developed by *Merrill and Garland* in 2016 at NVIDIA[24]. This SPMV does not require any type of preprocessing or formatting while trying to solve the issue of unequal workload in the case of multi threaded systems. The other techniques that could also solve the unequal workload problem usually entails either some preprocessing time or they need some extra storage which might not be always available.

The CsrMV implementation used in the thesis does the multiplication on a per row basis and this has the disadvantage that the solve time is determined by the largest number of non zero elements in a row. For example, in the system matrix for *MADPET-II*, the maximum number of non zero elements in a row are almost 5 times the number of average non zero elements in a row which leads to 5 times more operations for that row versus the average.

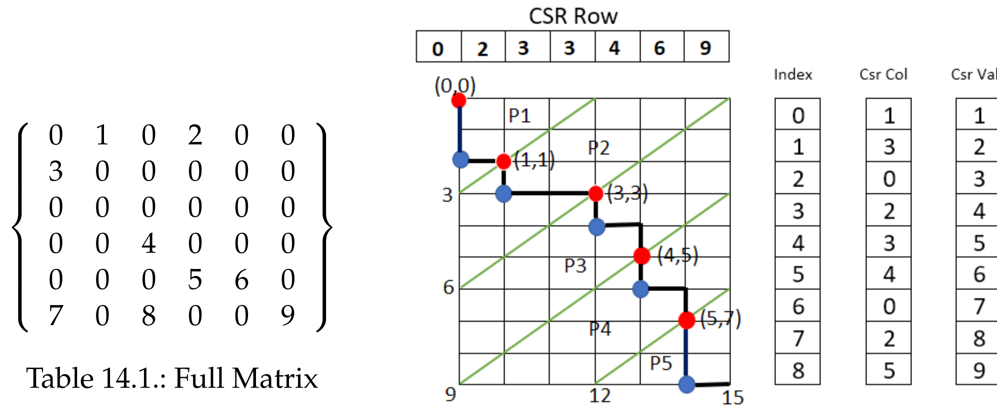


Table 14.1.: Full Matrix

Table 14.2.: Example showing visualization of merge based CsrMV.

Table 14.1 shows a dense matrix and the table 14.2 shows the arrangement of the *CsrRow* on the x-axis and the index of the arrays *CsrCol* and *CsrVal* on the y-axis which leads to the generation of a grid which a merged path computation will transverse from top-left to bottom-right in $|CsrRow| + |CsrCol|$ steps. While moving along the path marked by the *dark black line* in 14.2, when the merge path moves vertically it consumes the elements from *CsrRow* and when moving horizontally it consumes the elements from *CsrCol*/*CsrVal*.

For launching this algorithm on *p* threads, first the grid is divided into *p* parts diagonally of equal width and then each thread determines it own path in the part allocated to it. The starting vertex for each thread (*marked by red dots*) could be determined by the constrained binary search along the first diagonal in the thread's swath. More specifically, for any diagonal *k*: the vertex *(i,j)* fulfills the condition that $CsrRow_i >$ *all items before* $CsrCol_j$ and $i + j = k$. By using this approach the total work to be performed is of the order of *O(|CsrRow| + |CsrCol|)*.

15. Conclusion

In this work, the *Maximum Likelihood Estimation Maximization (MLEM)* algorithm for the *Positron Emission Topography (PET)* has been implemented making use of the GPUs which has been built upon the previous implementation of MLEM for multicore architectures. The implementation tries to offload the maximum amount of computation work on the GPUs. The remaining work is solved by the host to minimize the data transfer between the host and device which ultimately leads to better performance. The work on the host is further accelerated by the use of multiple threads using OpenMP. Moreover, the implementation also gives user the option to specify some of the parameters for selecting the GPUs through the configuration file.

The first part of the thesis gives a short introduction to the PET scanners, their configuration and why they are used. It also gives a mathematical foundation to the equations solved in the MLEM algorithm, all of which has also been presented as a pseudo code. After that the *Compressed Sparse Row (CSR)* format to store the matrix has been explained along with the sample codes to show how the Sparse Matrix Vector Multiplication (SPMV) and Transpose Sparse Matrix Vector Multiplication (SPMV_T) computations are performed using this format. The architecture of the GPUs has been discussed briefly along with CUDA as a programming language, which has been used to write the kernels for NVIDIA GPUs in the thesis. Most of the discussion about CUDA has centered around the technologies necessary to produce this work. It also discusses the optimization strategies from the host as well as device side to gain extra performance.

The next part has focused mainly on the performance and scaling of the implementation under different scenarios namely with respect to increasing number of OpenMP threads on the host, increasing number of GPUs and in the case when GPU memory alone is not enough. Apart from that, some of the tests also focuses on the performance improvements with the use of Pinned Memory, CUDA Aware MPI, compilation of code for specific GPU architecture and using custom kernels. Many conclusions can be drawn from the these tests, which can be summarized as follows:

- Although different MPI implementations are built upon the same standards, there are still some noticeable performance differences between them.
- The pinned memory could be used to speed up all sort of data transfer i.e. between host and device, between different MPI processes apart from I/O operations. But the

pinned memory should be used carefully as it takes more time to be allocated and deallocated and could even degrade the performance in low memory systems.

- The implementation of the library functions should be understood before using them, especially CuSparse from NVIDIA, since they are not always the best as has been the case with the SPMV_T using CSR format shown in the thesis.
- For accelerating the performance of the host code, the OpenMP threads should not be increased blindly as they not always lead to better performance.
- Compiling the code for a specific architecture not always leads to better performance in every kernels but since it does not degrades the performance, the codes should be compiled architecture specific whenever possible.

During the course of working on the implementation and later during the tests, there have been times when unexpected behaviour has been observed. An educated guess could be made to explain some of these behaviours but not for all of them. Further tests are needed to explain these behaviours. They are as follows:

- On the K20 machine in the LRR chair, if the code is launched with only *./implementation* it does not detect the GPU which is not the case when launched like *mpirun -n 1 ./implementation*.
- The reason for OpenMPI being slower in the first allreduce operation in the implementation compared to Intel MPI while being faster after that on the Mac Cluster is still not well understood.
- The reason for the slight difference in results between CPU and GPU is probably due to different floating point arithmetic implementations but it still require further tests to be sure.
- The unexpected performance hit when using Cuda Aware MPI warrants further scrutiny.
- A single MPI process could only handle upto 3 GPUs which is possibly due to I/O limitations but more research is need to be completely sure.

In hindsight, it could be said that the implementation is performing as intended under different scenarios. The implementation is able to adapt to the hardware it is being run on and perform satisfactorily. The two major performance bottlenecks could be solved by changing the splitting of system matrix from equally among MPI processes now to be more resource aware and by changing the implementation of SPMV_T to merge based SPMV_T which promises better performance and higher GPU occupancy.

Appendix

A. Computational Resources

A.1. DGX-1

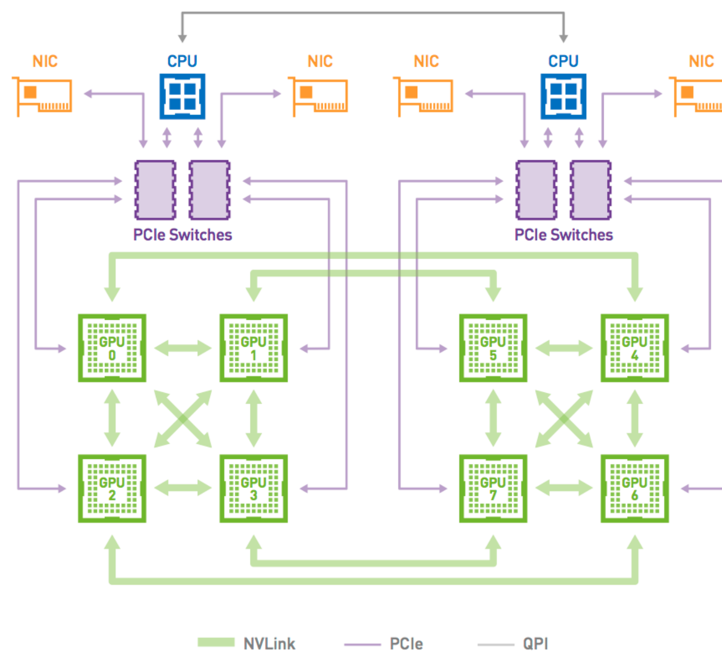


Figure A.1.: Schematic of DGX-1 showing different components [25]

The DGX-1 is a integrated system specifically designed for deep learning. It has 2 Intel 20 core Xeon processors connected with each other through *QuickPath Interconnect*, 8 Tesla P100 GPUs (Pascal Architecture) connected with each other through NVLink (NVIDIA's high speed GPU interconnect) and four 100Gb Infiniband network interface cards (NIC). The system provides 170TFlops of GPU performance on half precision floating point in addition to 3 TFlops of CPU performance on floating point. The figure A.1 gives a visual representation of all the components in the machine and how they are connected.

Specification	Value
OS	Ubuntu 16.04LTS
Host RAM	512 GB 2133 MHz DDR4
Model	Intel Xeon Broadwell E5-2698V4
Core	20
Threads	40
Base Frequency	2.20GHz
Turbo Frequency	3.60 GHz
Cache	50 MB
Bus Speed	9.6 GT/s QPI

Table A.1.: Each CPU Specifications

Specification	Value
Model	Tesla P100
Architecture	Pascal
Compute Capability	6.0
Cuda Cores	3584
Streaming Multiprocessors	56
Cores/SM	64
Single Precision Perf.	9300 GFlops
Base Frequency	1189 MHz
Turbo Frequency	1328 GHz
Memory Size	16 GB HBM2
Max Memory Clock	715 MHz
Memory Bus Width	4096 bit
Atomics	Hardware and CAS

Table A.2.: Each GPU Specifications

A.2. Mac Cluster

Mac cluster is made up of 4 nodes where each node is dual socket *Intel SandyBridge-EP Xeon E5-2670* with 2 *NVIDIA M2090 GPUs* and FDR Infiniband. The table A.3 and A.4 list the specification of CPU and GPU in more detail.

Specification	**Value**
Host RAM/node	128 GB
Model	Intel SandyBridge-EP Xeon E5-2670
Core	8
Threads	16
Base Frequency	2.60GHz
Turbo Frequency	3.30 GHz
Cache	20 MB
Bus Speed	8.0 GT/s QPI

Table A.3.: Each CPU Specifications

Specification	**Value**
Model	Tesla M2090
Architecture	Fermi
Compute Capability	2.0
Cuda Cores	512
Streaming Multiprocessors	16
Cores/SM	32
Single Precision Perf.	1331 GFlops
Base Frequency	1300 MHz
Memory Size	6 GB GDDR5
Max Memory Clock	1850 MHz
Memory Bus Width	384 bit
Atomics	lock/update/unlock

Table A.4.: Each GPU Specifications

B. Compiling and Using CUDA Aware OpenMPI on Mac Cluster

To compile the CUDA Aware OpenMPI on Mac Cluster follow these steps.

- First of all allocate one of the node from *nvd* partition and then ssh into it. The reason for this is that the files required for *NUMA* support are available on the compute nodes and not on the login node. Also don't forget to unload any other MPI implementation.
- Download the OpenMPI 1.10.7 by executing *wget http://www.open-mpi.de/software/ompi/v1.10/downloads/openmpi-1.10.7.tar.gz*
- Extract the file by *tar -xvzf openmpi-1.10.7.tar.gz*
- Go into the folder by *cd openmpi-1.10.7*
- Configure the OpenMPI by executing *./configure –prefix=$HOME/.openmpi –with-cuda =/lrz/sys/parallel/cuda/7.5/cuda/include CFLAGS=-D__LP64__ –with-wrapper-cflags="-D__LP64__ -ta:tesla"*

  The compiler flags *CFLAGS=-D__LP64__ –with-wrapper-cflags="-D__ LP64__ -ta:tesla"* are required with NVIDIA driver version 7.5 for things to work correctly.
- Next execute *make*
- Next execute *make install*
- Next close the ssh connection and go bask to the login node
- Put these 2 commands in bashrc

  export PATH=$PATH:$HOME/.openmpi/bin

  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/.openmpi/lib

To compile the code with the newly installed OpenMPI, unload any other MPI implementation and add

```
LFlags += -L$(HOME)/.openmpi/lib
IFlags += -I$(HOME)/.openmpi/include
```

Now launch the code normally.

C. Compiling NVML library on Mac Cluster

The NVML library has been used to check the available and total memory of the GPUs apart from checking the device temperature, core utilization and memory utilization. This library is installed with the CUDA Toolkit by default from version 8 onwards but that is not the case with the toolkit available in the Mac Cluster (Version 7.5). So to install it manually perform the following operations

- wget http://developer.download.nvidia.com/compute/cuda/7.5/Prod/gdk/gdk_linux_amd64_352_79_release.run
- chmod +x gdk_linux_amd64_*_release.run
- ./gdk_linux_amd64_*_release.run
- Link the library with the project in the Makefile.

D. Notes on MLEM Implementation

- Programming Language: C++
- Compilers
  - mpicxx : For compiling *.cpp* and *.c* files. *Intel MPI library version 5.1.3 for Mac Cluster* and *OpenMPI version 1.10.7 for DGX-1*.
  - nvcc : For compiling *.cu* files. CUDA version *7.5.17 on Mac Cluster* and CUDA Version *9.1.85 on DGX-1*.
- Compilation Flags
  - -O3 : For level 3 optimization
  - -std=c++11 : For using C++ 11 standard.
- Flags used with nvcc
  - -lcublas : For performing blas operations on NVIDIA GPUs.
  - -lcusparse : For performing sparse operations on NVIDIA GPUs.
  - -lmpi : To be able to compile code with MPI.
  - -lnvidia-ml : To be able to use NVML library.
  - -lboost_system -lboost_filesystem : To be able to use components of the BOOST library
  - -Xcompiler -fopenmp : To compile files using OpenMP.
  - -lgomp : To be able to link the files using OpenMP.
  - –gpu-architecture=*Value* : To compile the code for specific architecture. The *Value* is *sm_20* for Mac Cluster and sm_60 for DGX-1.
- Configuration File : The parameters in the configuration files are as follows
  - max_devices_to_use : Allows the user to set the maximum number of GPUs each process could use. Allowed range is [0,3]. Default value is 3.
  - max_nnz_per_device : Allows the user to define how many non zeros be offloaded on each GPU. Allowed range is [0 , non zeros per rank]. Default value is non zeros per rank.
  - max_temp_allowed : Maximum GPU temperature to be selected. Allowed range is [0,100]. Default value is 100.

- min_free_mem_req : Minimum memory percentage of GPU to be free for its selection. Allowed range is [0,100]. Default value is 100.
- max_mem_uti_allowed : Maximum memory percentage utilization of GPU for its selection. Allowed range is [0,100]. Default value is 100.
- max_gpu_uti_allowed : Max compute power percentage utilization of GPU for its selection. Allowed range is [0,100]. Default value is 100.

NOTE: Any of the parameter in the configuration file could be set to default by providing values out of range (negative included). If implementation does not find any configuration file it will use the *Default* values.

Bibliography

[1] D. L. Bailey, D. W. Townsend, P. E. Valk, and M. N. Maisey, *Positron emission tomography*. Springer, 2005.

[2] A. Berger, "How does it work?: Positron emission tomography," *BMJ: British Medical Journal*, vol. 326, no. 7404, p. 1449, 2003.

[3] T. Küstner, J. Weidendorfer, J. Schirmer, T. Klug, C. Trinitis, and S. Ziegler, "Parallel mlem on multicore architectures," in *International Conference on Computational Science*. Springer, 2009, pp. 491–500.

[4] M. Defrise, P. E. Kinahan, and C. J. Michel, "Image reconstruction algorithms in pet," in *Positron Emission Tomography*. Springer, 2005, pp. 63–91.

[5] L. A. Shepp and Y. Vardi, "Maximum likelihood reconstruction for emission tomography," *IEEE transactions on medical imaging*, vol. 1, no. 2, pp. 113–122, 1982.

[6] H. M. Hudson and R. S. Larkin, "Accelerated image reconstruction using ordered subsets of projection data," *IEEE transactions on medical imaging*, vol. 13, no. 4, pp. 601–609, 1994.

[7] C.-M. Chen and S.-Y. Lee, "On parallelizing the em algorithm for pet image reconstruction," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 860–873, 1994.

[8] D. S. Lalush and B. M. Tsui, "Performance of ordered-subset reconstruction algorithms under conditions of extreme attenuation and truncation in myocardial spect," *The Journal of Nuclear Medicine*, vol. 41, no. 4, p. 737, 2000.

[9] J. Dongarra, "Compressed row storage (crs)," 1995. [Online]. Available: http://netlib.org/linalg/html_templates/node91.html

[10] C. Nvidia, "Cuda c programming guide, version 9.1," *NVIDIA Corp*, 2018.

[11] J. Nickolls and D. Kirk, "Graphics and computing gpus," *Computer Organization and Design: The Hardware/Software Interface, DA Patterson and JL Hennessy, 4th ed., Morgan Kaufmann*, pp. A2–A77, 2009.

[12] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[13] N. T. Nvidia, "P100 gpu," *Pascal Architecture White Paper*, 2016.

[14] N. Gupta, "Texture memory in cuda," http://cuda-programming.blogspot.de/2013/02/texture-memory-in-cuda-what-is-texture.html.

[15] M. Harris, "How to access global memory efficiently in cuda c/c++ kernels," 2013. [Online]. Available: https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels

[16] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[17] M. Boyer, "Pinned vs. non-pinned memory." [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html

[18] N. Gupta, "Bank conflicts in shared memory in cuda," http://cuda-programming.blogspot.de/2013/02/bank-conflicts-in-shared-memory-in-cuda.html.

[19] C. Nvidia, "Pascal tuning guide, version 9.1," *NVIDIA Corp*, 2018.

[20] J. Kraus. (2013) An introduction to cuda-aware mpi. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[21] D. Fleisch and J. Kregenow, *A Student's Guide to the Mathematics of Astronomy*. Cambridge University Press, 2013, chapter 1.

[22] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[23] W. Kahan, "Ieee standard 754 for binary floating-point arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.

[24] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 58.

[25] M. Harris, "Nvidia dgx-1: The fastest deep learning system," 2017. [Online]. Available: https://devblogs.nvidia.com/dgx-1-fastest-deep-learning-system/