بسم الله الرحمن الرحيم

A Novel Deep Learning based Iterative Solver for Large Sparse Linear Equation Systems

By

Thaha Muhammed

A thesis submitted for the partial requirements of the degree of

Master of Science in Computer Science

Supervised By

Prof. Rashid Mehmood

Dr. Aiaad Al-Beshri

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY
KING ABDULAZIZ UNIVERSITY
JEDDAH – SAUDI ARABIA
Rajab 1438H – April 2017G

A Novel Deep Learning based Iterative Solver for Large Sparse Linear Equation Systems

By

Thaha Muhammad

This thesis has been approved and accepted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

EXAMINATION COMMITTEE

	Name	Rank	Field	Signature
Internal Examiner				
External Examiner				
Advisor	Dr. Aiaad Al-Beshri			
Advisor	Prof Rashid Mehmood	Professor		

KING ABDULAZIZ UNIVERSITY

Rajab 1438H – April 2017G

Dedication

My parents, my beloved wife, and my dear son Haroon.

ACKNOWLEDGEMENT

All praises due to Allah for providing me the opportunity to pursue my M.S. degree, for giving me the strength and capability to complete this thesis, and for his endless blessing that kept me reminding me that he is always the best of the planners.

I am indebted to my supervisor Prof. Rashid Mehmood for giving me directions throughout my research. I acknowledge his patient guidance, encouragement and excellent advice throughout this study. His sage advice, insightful criticism and encouragement aided me in writing this thesis in innumerable ways. Without his help, this work would not have been possible.

I would like to express my deepest sense of gratitude to my supervisor Dr. Aiaad Al-Beshri for his support and guidance throughout this research.

I admire the help of many people who offered me valuable support throughout my study. I also would like to acknowledge the support and help provided by Dr. Iyad Katib, the Vice-Dean of our faculty. I would like to express my special thanks to Dr. Kamal Jambi, the head of the Computer Science Department, for his encouragement and support.

I would like to thank my colleagues Mr. Muhammed Aqib and Mr. Yasir Arfat for their advice and encouragement throughout this work. I would also like to thank Mr. Saim Nasim and Mr. Syed Avez for their guidance and support.

I would also like to thank my family for the support they provided me through my entire life. I acknowledge my parents, wife and other family members, without whose love, encouragement, support and assistance, I would not have finished this work.

A Novel Deep Learning based Iterative Solver for Large Sparse Linear Equation Systems

Thaha Muhammed

Abstract

Sparse Matrix-Vector multiplication (SpMV) is one of the key operations in linear algebra that lies at the heart of diverse domains such as scientific computing, engineering, economic modeling, and information retrieval, to name a few. They play a significant role in solving linear system of equations using iterative methods. Sparse Kernels are computational operations on matrices whose entries are mostly zero so that computations with and storage of these zero elements may be eliminated. The emergence of parallel architectures, especially GPU, while offering higher computational performance, has led to the redesign of existing algorithms to suit the architecture.

The aim of this thesis is to design novel techniques that improve the performance of sparse Jacobi iterative linear solvers on Nvidia based GPUs. A detailed review of the relevant literature is carried out to identify the challenges and the research gaps. The review revealed that the matrix sparsity structures vary widely based on the application domains and this poses major challenges in obtaining consistent high performance from sparse iterative solvers on Nvidia Tesla K20 GPUs. These challenges include coalesced memory access to the sparse matrix and vector and load balancing among threads and warps. We have developed a deep learning tool that uses an extended set of features to dynamically address these challenges and invoke the most suitable storage format for the iterative solution of sparse linear equation systems. The iterative solver tool has been tested on matrices arising from real world problems. Compared to other leading works, our tool demonstrated 25% or higher performance on average in terms of the execution time and GFLOPS.

The contributions of this thesis include a state of the art survey on sparse storage schemes, a deep learning based novel methodology based on an extended set of sparse matrix features and a tool for the iterative solution of sparse linear equation systems.

TABLE OF CONTENTS

Examination Committee Approval ii

Dedication iii

Acknowledgement iv

Abstract v

List of Figures vii

List of Tables ix

List of Algorithms x

Chapter 1: Introduction 1

1.1 Motivation and Problem Statement 3

1.2 Research Objectives and Contributions 4

1.3 Thesis Organization 5

Chapter 2: Background 7

2.1 Numerical methods 7

2.1.1 Basic methods 8

2.1.1.1 Jacobi Methods 8

2.1.1.2 Gauss-Seidel 9

2.1.2 Krylov methods 10

2.1.3 Multigrid and Hybrid Techniques 11

2.2 CPU Sparse storage schemes 13

2.2.1 Coordinate Storage (COO) 13

2.2.2 Compressed Sparse Row (CSR) 14

2.2.3 ELLPACK (ELL) 15

2.2.4 Hybrid ELL/COO 15

2.2.5 Diagonal Storage (DIA) 16

2.3 GPU Architecture 17

2.4 Machine Learning 19

2.4.1 Linear Regression 20

2.4.2 Logistic regression 21

2.4.3 Multinomial Logistic regression (Softmax regression) 22

2.4.4 Deep Networks 23
2.4.5 Deep network architectures 24
2.4.6 Back propagation Algorithm 27
2.4.7 Activation Functions 28
2.4.7.1 Sigmoid 28
2.4.7.2 Tanh 29
2.4.7.3 ReLU 29
2.4.7.4 Leaky reLUs 31
2.4.7.5 Maxout 32
2.4.8 Loss Functions 32
2.4.9 Optimization functions 35

Chapter 3: Literature Review **36**
3.1 CPU Sparse Storage Schemes 36
3.1.1 Serial Schemes 36
3.1.2 Parallel Schemes 40
3.2 FPGA schemes 41
3.3 MIC schemes 41
3.4 GPU Sparse Storage Schemes 42
3.5 Iterative Solvers on GPUs 61
3.6 Challenges in GPU Implementation of Sparse Solvers 65
3.7 Gaps in Literature. 67

Chapter 4: Proposed Technique **71**
4.0.1 Dataset Formation 74
4.0.2 Feature Selection 75
4.0.3 Feature Distribution Analysis 76
4.0.4 SpMV Performance Analysis on feature set 79

Chapter 5: Analysis and Evaluation of proposed technique **85**
5.1 Quantitative Analysis of Deep Model 85
5.2 Comparison with state-of-art 99

Chapter 6: Conclusion **102**

List of Figures

2.1 The dense reprsentation of matrix A 14

2.2 The COO representation of matrix A 14

2.3 The CSR representation of matrix A 15

2.4 ELLPACK representation of matrix A 16

2.5 HYB representation of matrix A 16

2.6 Latency hiding in GPU and CPU [1] 17

2.7 Hardware Architecture of CUDA based GPU [1] 18

2.8 Grouping of threads in CUDA based GPU. [1] 19

2.9 Performance comparison with data size for learning algorithms [2] 24

2.10 A Simple Deep Network 25

2.11 A Simple Artificial Neural Network 25

2.12 Sigmoid function 29

2.13 Tanh function 30

2.14 ReLU function 30

2.15 Leaky ReLU function 31

3.1 ELL-R representation of matrix A 43

3.2 JAD representation of matrix A 44

4.1 Sparsity Structure of some sparse matrices 72

4.2 The correlation matrix for our selected features 76

4.3 Projection of points in various 2-D planes 79

5.1 GFLOPS and Execution Time comparison of various Sparse matrix storage schemes 88
5.2 Anomalies 90
5.3 GFLOPS for various selected application domains 91
5.4 Execution time for various selected application domains 92
5.5 Anomalies Explaination 94
5.6 Density plot of GFLOPS/s against the size of matrices 95
5.7 Total Gain and Loss incurred by our proposed Adaptive Deep Learning based stoarge technique. 96
5.8 Extrapolation of the GFLOP/s performance with size of matrices 97
5.9 Total GFLOP/s obtained from Jacobi execution 97
5.10 Extrapolating the GFLOP/s performance for jacobi as size increases . . 98
5.11 GFLOP/s obtained for Jacobi on selected matrices 99
5.12 Jacobi timings for selected matrices 99

List of Tables

4.1 Distribution of class on dataset for SpMV kernel on Tesla K20c 75

4.2 Features Extracted From Sparse Matrices 77

4.3 Summary of our feature set 78

4.4 GPU Configurations 83

5.1 Evaluation of Classifier 86

5.2 Count of matrices with Loss $>$ 5x and $>$2x 87

5.3 Average slow down of sparse representations vs Deep learning model . . 87

5.4 Count of the best performing hundred largest and hundred smallest matrices in Dataset. 89

5.5 Comparison of Deep technique Accuracy with shallow techniques Accuracy 100

5.6 Comparison of Deep technique LCB with shallow techniques LCB 100

List of Algorithms

1 The Training Phase of Deep Networks 80

2 Adaptive Deep Learning Jacobi 81

3 The Deep Network predictive model 81

4 Dynamic Sparse format Jacobi Solver 82

Chapter 1

Introduction

Linear algebra is vital to many fields of science and engineering. Especially sparse linear Algebra, which has been included by the Berkeley scientists in their set of motifs (the seven dwarfs [3]). Among the sparse numerical techniques, iterative solution of linear systems can be considered as of prime importance due to its application in various important areas such as solving finite and finite differences of PDEs [4], High accuracy surface modelling [5], finding steady-state probability vector of Markov chains [6], solving the time-fractional Schrödinger equation [7], web ranking [8, 9, 10], inventory control and manufacturing systems [11], queuing systems [12, 13, 14, 15, 16, 17], fault modelling, weather forecasting, stochastic automata networks [18, 19], communication systems [20, 21], reliability analysis, wireless networks [22, 23, 24], sensor networks [25], computational biology [26], computational physics, computational chemistry and natural language modelling [27].

Sparse linear systems consist of sparse matrices with a huge number of rows and column. The fraction of non-zero elements to the total elements is quite low. Since the sparse matrices consist of only a few percentage of non-zero elements, they require specialized storage schemes and algorithms so as to efficiently store, access, and compute the sparse matrices. Sparse linear systems are generally of the form Ax=b, where A is a sparse matrix and b is a dense vector and x is the solution of the system that needs to be solved.

The system of Linear equations of the form Ax=b can be solved by direct methods or iterative methods [28, 29, 30, 31, 32]. The direct methods are based on factorizations. Gaussian elimination, LU factorization and QR [32] factorization are the important direct techniques. When the matrix A is large, the computation becomes expensive due to fill-in phenomenon, which arises due to the spawning of new entries in the matrix A, resulting from the factorization process of direct methods. Whereas iterative methods do not alter the matrix A, they only use the matrix in the context of matrix-vector multiplication (MVP) [33, 34]. Iterative methods generate approximate solutions for linear systems by successive approximations at each iteration [35, 6, 33]. They start out with an approximate solution and iteratively modify the approximation until convergence. However they do not guarantee a convergence, but they are faster than the direct methods. The iterative methods can be further classified into stationary iterative methods or pure iterative (based on matrix splittings) [28, 29, 36, 37, 32] and non-stationary iterative methods [38, 39, 32, 31]. Non-stationary iterative techniques include Krylov subspace techniques such as Conjugate Gradient and Generalized Minimal Residual (GMRES). The amount of literature on Iterative methods are huge [35, 31, 40, 32, 38]. A survey on the existing iterative solutions for linear systems can be found in [41].

Sparse Matrix-Vector multiplication (SpMV) is one of the key operations that play a significant role in solving iterative linear systems. Iterative solvers such as Jacobi and Gauss-Seidel mainly consist of multiple iterations of Sparse Matrix-Vector computations whereas a single iteration in Krylov solvers include several SpMV multiplications. Sparse Matrix-Vector Multiplication (SpMV) is a Level-2 BLAS operation between a sparse matrix and a dense vector and can be represented mathematically as $y = A \times x + b$.

The performance improvement of single threaded applications has given way to parallel computing to attain a higher throughput. The vanguard among the current massively parallel chip-based architectures are the GPUs. The advent of GPUs as a revolutionary technology providing massive parallelism and higher throughput has resulted in the accelerated performance of scientific applications. A large number of scientific applications has been parallelized to run on GPU for improved performance. Moreover, the ubiquity of the technology enables researchers easy access to GPUs. The key elements for running serial algorithms on GPU are the design, analysis, and implementation of parallel algorithms that scale to hundreds of coupled cores. The difficulty is in adapting the serial code to parallel architecture.

Nvidia introduced a parallel computing architecture called CUDA (Compute Unified Device Architecture). It provides a parallel computing platform and API for developing and running applications on the GPU. This approach is also known as GPGPU (General

Purpose GPU) approach. In this method, the compute intensive parts of the application are offloaded to the GPU whereas the sequential code is run on the GPU. The GPU acts as a coprocessor to the CPU (host). An extended C/C++/Fortran is used to develop applications in CUDA. The parallel compute intensive parts that will be run on the GPU is written in special functions called kernels. A kernel is identified by providing unique qualifiers, __global__ and __device before the method names. The kernels to be executed are called from the CPU. From the hardware perspective, the central processing unit in the GPU is called streaming multi-processor units (SMs). At a given time an SM unit executes a warp of threads. A warp consists of a group of 32 parallel threads. Each SM runs the threads using SIMT (Single Instruction Multiple threads). A single instruction is carried out by all the threads in a single warp at the same time. The threads are logically grouped into blocks that can be 1, 2, or, 3-dimensional thread arrays. The blocks are further grouped into grids that can be 1, 2, or 3-dimensional grids of blocks.

Sparse matrix-vector multiplication is heavily memory bound. Hence, we need to focus on efficient and compact data structures and improving the bandwidth efficiency. GPU is a throughput device. Utilizing the full potential of GPU requires exposing a significant amount of fine-grained parallelism, Computations should be structured such that there is sufficient regularity in the execution path and the memory access. Dense matrices are limited due to the floating point throughput, whereas sparse matrices are restricted primarily by the bandwidth due to irregular memory access.

1.1 Motivation and Problem Statement

Memory bound algorithms such as Sparse Iterative Solvers exhibit superior performance on GPUs due to the high bandwidth memory hierarchy in cache based GPUs. PageRank algorithm, HITS, and Random walk with a restart on GPUs have reported a performance improvement of 18 to 32 times. [43] show that the GPU-accelerated LOBPCG based on SpMM kernel is 3 to 5 times faster than multicore CPUs with the same power draw, which further indicates that the energy efficiency of executing the SpMV kernel on GPU is more than that of CPU [42, 43].

Implementation of parallel iterative solvers on GPU has numerous issues. Specialized storage structures are used to improve the performance of sparse SpMV. These structures have design issues for translating it to GPUs. The problems faced in implementing parallel sparse iterative solvers and converting sparse structures to GPU is as listed below:

1. Coalesced memory access to both the sparse matrix and the vector.
2. Load balance among threads and warps
3. Thread divergence among the threads.
4. Performance variance based on the structure of the sparse matrices
5. The amount of memory access required for computations.

Depending upon the sparsity of the structure the performance of the matrices vary and issues such as load balancing among threads, and thread divergence are dependent upon the sparsity structure of the matrices. The sparse storage schemes that satisfy the above conditions does not necessarily satisfy these conditions for other matrices and hence there is a performance deterioration.

The sparsity structure of the sparse matrices varies from matrix to matrix, and application domain to domain. The performance of the SpMV kernel is dependent upon the sparsity structure of the matrices. We can deduce that none of the sparse representation schemes are perpetually superior rather the SpMV performance depends upon the sparse representation and the choice of sparse representation is based on the sparsity structure. The GPU architecture used for SpMV computations and the thread-block-warp configuration of the SpMV kernel are two other factors that affect the performance of SpMV kernel on GPUs.

1.2 Research Objectives and Contributions

The aim of this thesis is to design novel techniques that improve the performance of sparse Jacobi iterative linear solvers on Nvidia based GPUs. The following work has been carried out towards the stated aim.

1. A detailed review of the relevant literature is carried out to identify the challenges and the research gaps.

2. The review revealed that the matrix sparsity structures vary widely based on the application domains and this poses major challenges in obtaining consistent high performance from sparse iterative solvers on GPUs. These challenges include coalesced memory access to the sparse matrix and vector and load balancing among threads and warps.

3. We have developed a deep learning tool that uses an extended set of features to dynamically address these challenges and invoke the most suitable storage format for the iterative solution of sparse linear equation systems. For designing the deep model, we have considered COO, CSR, HYB, ELL, and HYB SpMV kernels based on CUDA architecture.

4. The iterative solver tool has been implemented on Nvidia Tesla K20 GPU and tested on matrices arising from real-world problems. Compared to other leading works, our tool demonstrated 25% or higher performance on average in terms of the execution time and GFLOPS.

The contributions of this thesis include

- A state of the art survey on sparse storage schemes.
- A deep learning based novel methodology using an extended set of sparse matrix features.
- A deep learning tool for the iterative solution of sparse linear equation systems.

1.3 Thesis Organization

The rest of the Thesis is organized as follows.

Chapter 2 presents the background material on the sparse matrix storage schemes and iterative methods for the solution of linear equation systems. A basic background on GPU and CUDA architectures is also provided. We discuss Deep Learning background in this section.

Chapter 3 reviews the literature on iterative solvers and sparse storage schemes on GPU. A discussion on the challenges in GPU implementation of sparse storage scheme and the iterative solvers is provided. The gaps in the current literature has been identified.

Chapter 4 discusses our proposed technique to improve the performance of Jacobi algorithm. We discuss the algorithms of Deep network based models and Iterative Jacobi solvers that we have proposed.

In Chapter 5, we analyze and compare the efficiency, accuarcy, and the performance of our scheme as compared to other existing schemes.

Chapter 6 concludes the our thesis and provides more discussion on further research areas.

Chapter 2

Background

2.1 Numerical methods

The system of Linear equations of the form Ax=b can be solved by direct methods or iterative methods [28, 29, 30, 31, 32]. The direct methods are based on factorizations. Gaussian elimination, LU factorization and QR [32] factorization are the major direct methods. When the matrix A is large, the computation becomes expensive due to fill-in phenomenon, which arises due to the spawning of new entries in the matrix A, resulting from the factorization process of direct methods. Whereas iterative methods do not alter the matrix A, they only use the matrix in the context of matrix-vector multiplication (MVP) [33, 34]. Iterative methods generate approximate solutions for linear systems by successive approximations at each iteration [35, 6, 33]. They start out with an approximate solution and iteratively modify the approximation until convergence. However, they do not guarantee a convergence, but they are faster than the direct methods. The iterative methods can be further classified into stationary iterative methods or pure iterative (based on matrix splittings) [28, 29, 36, 37, 32] and non-stationary iterative methods [38, 39, 32, 31, 44]. Non-stationary iterative techniques include Krylov subspace techniques such as Conjugate Gradient and Generalized Minimal Residual (GMRES). The amount of literature on Iterative methods are enormous [35, 31, 40, 32, 38, 44]. A survey on the existing iterative solutions for linear systems can be found in [41].

2.1.1 Basic methods

2.1.1.1 Jacobi Methods

Jacobi method is a stationary iterative method for solving linear systems of form $Ax = b$, where $A \in R^{n\times n}$, and $x, b \in R^n$. Generally, the stationary iterative methods can be given by the expression

$$x^{(k+1)} = P^{-1}(P - A)\, x^{(k)} + P^{-1}b, \tag{2.1}$$

Where P represents a preconditioner. The preconditioner is chosen such that it is very close to the initial matrix A and still allow fast computations. The preconditioner P, for Jacobi method, is the diagonal of the matrix A. Hence the (2) for Jacobi can be rewritten as

$$x^{(k+1)} = D^{-1}(D - A)\, x^{(k)} + D^{-1}b \tag{2.2}$$

where $D \in \{A_{ij}|i = j\}$, implying that D is the Diagonal of the matrix A. We can split A as $A = D - (L + U)$, where L is the lower triangular matrix with elements $l_{ij} = -a_{ij}$ if $i > j$, $l_{ij} = 0$ if $i \le j$ and U is the upper triangular matrix with entries $u_{ij} = -a_{ij}$ if $j > i$, $u_{ij} = 0$ if $j \le i$.

Hence, the$k+1$-th iteration in Jacobi method can also be expressed mathematically as

$$x_i^{(k+1)} = a_{ii}^{-1}(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}) \tag{2.3}$$

for all i,0≤i≤ n. a_{ij}reprsents i -th row and j-th column of the matrix A. $x_i^{(k+1)}$ and $x_i^{(k)}$ reprsents the i-th element of the iteration vector for $(k+1)$-th and k-th iterations respectively.

The Jacobi method does not converge very easily for all system of linear equations. The Jacobi method will only converge if every eigenvalue (λ) of $M = I - D^{-1}A$ satisfies the condition $|\lambda(M)| < 1$. If there is a convergence, then the rate of convergence is determined by the magnitude of the eigenvalues of M. The closer the eigenvalues are to one the slower the convergence would be. In case the Jacobi does not converge, it can be made to converge using the under-relaxation parameter in the Jacobi technique. Moreover, the convergence can be accelerated with the help of the overrelaxation parameter. This results in a technique called Jacobi overrelaxation method (JOR). Mathematically we can express this as

$$x_i^{(k+1)} = \omega a_{ii}^{-1}(b_i - \sum_{j=1,\, j\neq i}^{n} a_{ij}x_j^{(k)}) + (1-\omega)x_i^{(k)} \tag{2.4}$$

for all i,0≤i≤ n. ω is called the relaxation parameter and $\omega \in (0,2)$.For any $\omega \neq 0$ equation (5) is consistent. If $\omega > 1$, the method becomes overrelaxation and when 0< $\omega < 1$ it is called as under relaxation.

For the computation of the $x(k+1)$-th iteration vector, Jacobi only uses $x(k) - th$ vector. Hence, Jacobi method requires storage only for the two iterative vectors along with the matrix A and the diagonal elements A. Hence Jacobi and JOR methods are inherently parallel.

2.1.1.2 Gauss-Seidel

Gauss-Seidel method is an improvement to the Jacobi method wherein in the$(k+1)$-th iteration the available values of $x_i^{(k+1)}$ is used to update the solution. The preconditioner chosen is the lower triangle of the matrix A including the diagonal and can be represented as $P = D - L$, Hence Equation (2) can be written as

$$x^{(k+1)} = (D-L)^{-1}U\,x^{(k)} + (D-L)^{-1}b \tag{2.5}$$

which represents the solution at k+1-th iteration using Gauss-Seidel method. It can also be written as

$$x_i^{(k+1)} = a_{ii}^{-1}(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}) \tag{2.6}$$

The two major advantages of Gauss-Seidel over Jacobi are that it converges faster than Jacobi and that it requires only one iteration vector [6]. This reduces the storage required by large systems. However, the major disadvantage of this technique is that it is not inherently parallel due to the use of values from the current iterations. Even then they have been used for the parallel solution of Markov chains [45, 46].

Analogous to Jacobi iterations we can introduce a relaxation parameter $\omega \in (0,2)$

for Gauss-Seidel. The resulting technique is called Successive over-relaxation. Mathematically this can be represented as

$$x_i^{(k+1)} = \omega a_{ii}^{-1}(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}) + (1-\omega)x_i^{(k)} \tag{2.7}$$

2.1.2 Krylov methods

Zhong-Zhi Bai [47] discusses the motivations that lead to the use of Krylov Subspace methods and provides an insight into various techniques that come under Krylov subspace for solving linear systems. The authors discuss the advantages and disadvantages of direct techniques such as Gaussian Elimination, LU decomposition, QR decomposition, and efficient modification of Graham's determinant method. The author furthermore discusses Grahm-Schmidt technique for finding the orthogonal vectors for QR decomposition as it is an elementary ingredient in Krylov subspace techniques. The authors further discuss the two class of iterative techniques, Krylov subspace techniques, and over-relaxation techniques including Jacobi, and Gauss-Siedel. The Arnoldi process for finding the orthogonal columns and GMRES technique are also discussed by the author. The author also lists out the preconditioners that could be used in association with the Krylov subspace.

j-th order Krylov Subspace consists of a linear combination of b, Ab, $A^2b, \ldots,$ $A^{j-i}b$ and we denote it by $\mathcal{K}_\backslash$. The linear subspace

$$\mathcal{K}_|(A, b) = span\{b, Ab,\ Ab^2, ..., A^{j-1}b\} \tag{2.8}$$

is called the j-th order Krylov subspace.K represents the Krylov matrix that consists of the basis vectors from the Krylov subspace as its columns. The idea is to choose the best possible linear combination x_j from the Krylov subspace. Depending upon various definition of "the best possible combination" we can have various techniques to get x_j. The major four approaches for selecting x_j in $\mathcal{K}_\backslash$ is listed below.

1. Conjugate gradients: The residual $r_j = b - Ax_j$ is orthogonal to Krylov subspace$\mathcal{K}_|$

2. MINRES and GMRES: The residual r_j has minimum norm for $x_j$in $\mathcal{K}_|$.

3. Biconjugate Gradients: The residual r_j is orthogonal to different space $\mathcal{K}_|(A^T)$.

4. SYMMLQ: The error $e_j$has a minimum norm.

Further detailed discussions on Krylov methods can be found in [48, 49].

2.1.3 Multigrid and Hybrid Techniques

Jacobi and Gauss-Seidel produce smooth errors which result in the convergence requiring $O(N^2)$ iterations. The higher frequencies of the error are removed in the first few iterations, and the lower frequencies take much time to be removed. This is quite intolerable. Hence, general multi- grid techniques such as proposed by [50, 51] moves the system to a coarser grid wherein we consider smooth frequencies become rough frequencies. This leads to faster convergence. The coarser grid can be used to reduce a large chunk of error. We alternate between fine and coarse to achieve a convergence that is independent of the size of n and can be solved in fixed number of iterations.

Wen [52] propose a new accelerated two-level multigrid method for finding a solution for solving Markov Chains. He applies a Quadratic extrapolation to modify the two level multi-grid on the coarse level so that the system achieves faster convergence while calculating the stationary probability distribution of Markov chains. Even though the number of iterations is reduced the quadratic extrapolation increases the compute time of the system quite heavily, since it consists of computations such as QR decomposition (using Gram-Schmidt) and several other matrix operations.

Issues with fixed point techniques such as Jacobi, Gauss- Seidel, SOR, JOR as compared to Krylov subspace methods such as conjugate gradient [53] and GMRES (Generalized minimal residual methods) [54]:

- Large prefactors
- Poor scaling with system size

However, Jacobi is simple and inherently parallel. So researchers have tried to accelerate Jacobi using various techniques such as Chebyshev acceleration [38] and scheduled relaxation Jacobi [55]. However, [38] requires the extremal Eigenvalues of the Matrix and [55] be only suitable for solving second-order finite difference second order finite difference discrete elliptic equations. In this context, Pratapa et al. [56] proposed a new technique called Alternating Anderson –Jacobi (AAJ) method to solve large sparse linear systems. It accelerates the classical Jacobi method by speeding up the convergence by employing extrapolation at periodic intervals inside the classical Jacobi technique. The authors find that the proposed system scales well.

In general Anderson's extrapolation [57] is used to accelerate the convergence in nonlinear fixed point iterations. From the point of linear systems, it is similar to GMRES [58, 59, 60]. The authors compare the performance of the proposed techniques against the following techniques:

- Weighted Jacobi
- Anderson Jacobi
- Scheduled relaxation Jacobi
- GMRES (Based on Krylov subspace). Simple one without multigrid.

The following Problem model were considered by the authors:

- Laplace Equation
- Poisson and Helmholtz equation
- General Non-Symmetric matrices

This technique can be parallelized so as to decrease the computational time based on Markov chain application. Unlike multigrid method we discussed earlier the time overhead due to faster convergence theoretically seems to be lesser in the above method.

Touzene [61] propose a new parallel technique for solving Markov chain equations called PBA (Parallel Block Aggregated) iterative method. This technique is a combination of both the iterative techniques as well as direct techniques for solving linear systems. This technique is built on the aggregation of Markov chain states to blocks. Each processor in the parallel system updates the probabilities of a block containing the states to form a new aggregated Markov chain. The aggregated blocks are then solved using LU Factorization. Then the processors update the vector π. Once each processor has performed these, they exchange the sub-vectors to form a new updated vector π and this continuous until convergence. The authors compare this technique with MBJ (Modified Block Jacobi) and BV [62] schemes on a blade server and prove that it runs faster and converges faster than block Jacobi techniques.

Touzene [63] further proposes a new technique called a parallel sparse iterative method (PPSIA) for solving large-scale Markov chains. This technique is based on state isolation of Markov chain and aggregation schemes. Initially, the authors develop an algorithm named SIA (Sparse Isolation Aggregation) which has Gauss-Seidel effect. Since Gauss-Seidel is inherently not parallel, it is hard to parallelize this technique. The authors hence propose a pipelined based parallel version of SIA called PPSIA. In the SIA technique, the states are divided into two macrostates, Left (L) and Right (R).Moreover, for the i^{th} state, the $(i + 1)$-th state is isolated. This results in three states$L$, $(i + 1)$, $and\, R$. Later aggregated system parameters are calculated, and the aggregated system is solved. In the PPSIA the pipelining takes as follows:

- First iteration: one processor updates p_i components using SIA algorithm and the other processors use the conventional power method for the updates.
- Second iteration: First and second processor use SIA whereas the rest use the power method.
- Iteration p: all p processors use SIA for their updates. The authors test this technique using an MPI implementation and compare against GMRES.

2.2 CPU Sparse storage schemes

Several sparse storage schemes have been proposed by scholars to improve the performance of sparse matrix computations. Many of them have been designed for matrices with particular sparsity structure or from a given application domain. However, there are a few basic storage formats that have always been used to store sparse matrices. Most of the sparse matrices are stored using one of these formats. Optimized implementations of these basics sparse storage schemes for GPU based SpMV can be found in cuSPARSE [174] and CUSP libraries [64]. Figure. 2.1 shows an example matrix A, that we shall use to illustrate various sparse storage schemes.

2.2.1 Coordinate Storage (COO)

The coordinate (COO) [65] format is a simple sparse storage scheme. We use three arrays: *row*, *col*, and *val* to store the row indices, column indices, and values, respectively,

$$m\left\{\underbrace{\overset{\xleftrightarrow{\quad n \quad}}{\begin{pmatrix} 1 & 0 & 2 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 5 & 6 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 8 & 0 & 9 \\ 0 & 10 & 0 & 11 & 0 & 0 \end{pmatrix}}}_{n}\right. \Big\updownarrow m$$

Figure 2.1: The dense reprsentation of matrix A

val	1	2	3	4	5	6	7	8	9	10	11
row	0	0	0	1	2	2	2	3	3	4	4
col	0	2	5	4	0	1	5	3	5	1	3

Figure 2.2: The COO representation of matrix A

of the nonzero matrix entries. All the three arrays have a dimension of nonzeroes. The required storage is always proportional to the number of nonzeros. Both the row and column indices are stored explicitly in COO. Figure 2.2 illustrates the COO representation of the example matrix A. Given an 8-byte floating point number representation and a 4-byte integer representation the COO format requires $16 \times nnz$ bytes to store the whole sparse matrix. Implementation of this storage format on GPU requires atomic operations.

2.2.2 Compressed Sparse Row (CSR)

The compressed sparse row (CSR) format like the COO format, stores the column indices and nonzero values in arrays named *col* and *val* explicitly. A third array of row pointers,*ptr*, are used to find the rows. For an $m \times n$ matrix, *ptr* has length $m + 1$ and stores the offset into the $i - th$ row in$ptr[i]$. The last entry in *ptr*, which would otherwise correspond to the $(m + 1)$-st row, stores nnz, the number of nonzeros in the matrix. Figure 11 illustrates the CSR representation of an example matrix. The row pointers facilitate fast querying of matrix values and allow other quantities of interest, such as the number of nonzeros in a particular row ($ptr[i + 1] - ptr[i]$), to be readily computed. For these reasons, the compressed row format is commonly used for sparse matrix computations (e.g., sparse matrix-matrix multiplication) in addition to sparse matrix-vector multiplications. Generally, one thread is assigned per row for SpMV

val	1	2	3	4	5	6	7	8	9	10	11
col	0	2	5	4	0	1	5	3	5	1	3

ptr	0	3	4	7	9	11

Figure 2.3: The CSR representation of matrix A

operation and this technique is known as CSR scalar. To improve the performance one warp is assigned to a row and this technique is called CSR vector [177]. The CSR format requires $12 \times nnz + 4 \times (m + 1)$ bytes for storage.

2.2.3 ELLPACK (ELL)

ELLPACK [66] also known as ITPACK is a sparse matrix storage well suited for vector architectures including GPUs. In this storage format, a sparse matrix A with dimensions $m \times n$ is stored with the help of following data structures: (1) A 2-D float array, named data, of size $m \times max$, where max is the maximum nonzeroes per row, to store the values. The rows that contain less than max nonzeroes are zero padded. (2) Another 2-D integer array with dimension $m \times max$, named indices, to store the column indices. Again the rows are zero padded to a length of max. An example of a 4x4 matrix stored in ELLPACK format is illustrated in Fig. 2.4. We can observe that the row indices are implicitly stored whereas column indices are explicitly stored. The data are generally stored in column major format [67] that increases the performance in a GPU due to coalesced global memory access. ELLPACK is an efficient sparse matrix representation when maximum nnz per row does not vary considerably from the average. The performance of ELLPACK also reduces when the percentage of zeroes increases. The deterioration persists even after memory access for zero padded elements are restricted through conditional divergence. They are generally suited for structured matrices. The storage size of ELL format is $12 \times m \times max$ bytes, given that value is stored as 8-byte floating point number and the indices as 4-byte integer.

2.2.4 Hybrid ELL/COO

The hybrid format was proposed by Bell and Garland to eliminate the issues with ELL [67]. The major disadvantage of ELL was the performance degradation on varying

$$A = \begin{pmatrix} 1 & 0 \\ 3 & 4 \\ 6 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 12 & \end{pmatrix}$$

Figure 2.4: ELLPACK representation of matrix A

$$A = \begin{bmatrix} 1 & 0 \\ 3 & 4 \\ 6 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 12 & \end{bmatrix} \quad \begin{array}{r} \text{val} = \begin{bmatrix} 12 & 8 \end{bmatrix} \\ \text{col} = \begin{bmatrix} 5 & 4 & 15 \end{bmatrix} \\ \text{row} = \begin{bmatrix} 1 & 13 & 6 \end{bmatrix} \end{array}$$

Figure 2.5: HYB representation of matrix A

number of nnzs per row of the matrix. This storage scheme divides the matrix into two parts, a dense part and a sparse part. In the dense part , we store the typical number of nonzeroes per row in the ELL storage format and the rest of entries belonging to rows with higher nnzs in the COO format. The size of the ELL matrix needs to be determined normally from the input matrix. Generally, the ELL format is three times faster than the COO technique for matrices with rows greater than 4000. Hence, we can add a $k-th$ column to the ELL part if at least one third part of the matrix rows contains k non-zeroes. The remaining nnzs in the rows are stored using the COO format. Another way to compute k is to use histogram of nnz per row as seen in CUSP implementation. An example of HYB storage for a sparse matrix A is shown in Fig. 2.5. Hence, this storage technique use two 2-D arrays of size$m \times k$ for storing the ELLAPCK part, where k is the number of selected columns, and three seperate arrays to store the COO part. If the actual matrix is of size $m \times n$ then the value array of the COO part would have a dimension of $(n - k)$. Given that the value is stored as 8-byte floating point and the indexas 4-byte integer, the storage space required for this scheme is $12(m \times k) + 16(n - k)$ bytes.

2.2.5 Diagonal Storage (DIA)

Diagonal matrix is only useful for storing structured sparse matrices especially the ones containing diagonal. It uses two arrays, a value array for storing the values and an offset array that stores the offsets of the values from the main diagonal. The offset is

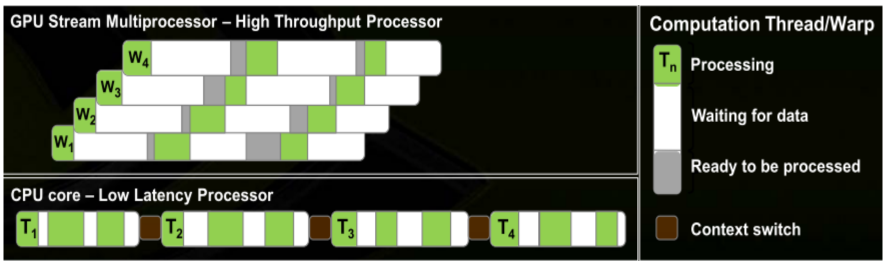


Figure 2.6: Latency hiding in GPU and CPU [1]

calculated from the main diagonal. The main diagonal has offset zero. A positive offset indicates super diagonals and a negative offset indicates sub diagonals. The size of offset array would be equal to the number of non-zero diagonals of the matrix. Implicit storage of row and column indices reduces memory footprint. the amount of data that needs to be transferred reduces. Another major advantage is that DIA format provides coalesced memory access to the sparse matrix A and the vectors x and y. The major disadvantage is that it stores the zero values in the diagonals and all matrices cannot be stored in this format.

2.3 GPU Architecture

GPUs are designed to produce a higher throughput unlike normal processors which are latency oriented. The CPU architecture generally minimize latency within each thread whereas the GPU hides latency with computation among the warps. This idea is illustrated in Fig. 2.6. The major two components of a GPU are : Global memory and SMs (Streaming multiprocessors).

The global memory is analogous to RAM in a CPU based system. It is a common memory accessible by both CPU and GPU. The SMs are the actual computational units. Each SM has its own control unit, registers, execution pipelines, and caches. There are 32 CUDA cores per SM. The Cuda cores have floating point unit and an integer units that performs computations. It also consists of a logic unit, move-compare unit, and a branch unit. The SMs have an L1 cache as well shared memory. L1 memory is hardware managed whereas shared memory is user managed. The hardware architecture has been illustrated in Fig. 2.7.

Logically, the threads are grouped into thread blocks, which can utmost be 3-dimensional and the blocks are further grouped into grids as shown in Fig. 2.8. A

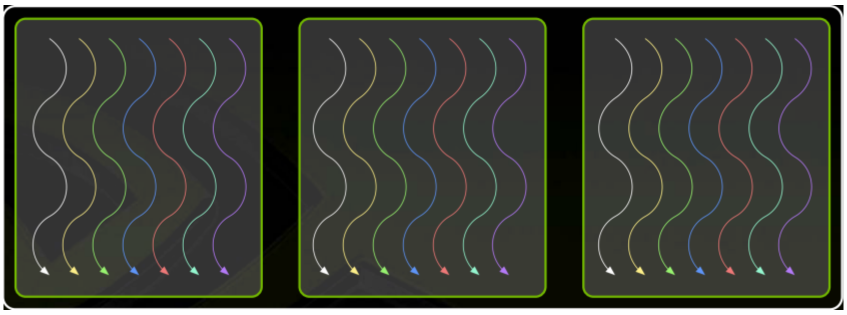


Figure 2.7: Hardware Architecture of CUDA based GPU [1]

CUDA kernel is executed as a grid of blocks of threads.

The hardware can assign any number of blocks to any processor at any time. A CUDA kernel can scale to any number of parallel processors. The blocks can execute in any order and are independent of each other. The threads are actually executed by the SM (Streaming multiprocessor), The SMs are similar to the cores in the normal processors. Threads are assigned to the SM in block granularity meaning that if a thread from a block is given to certain SM for execution, then the rest of the threads in that block also will be executed by that SM. In current CUDA architecture, we can assign 8 thread blocks to SM. This is a hardware restriction. A fermi SM can take 1536 threads per SMs. This can be either six blocks with 256 threads/block or 3 blocks with 512 threads per block. The SM is the one who maintains the thread and block Ids. SM also manages and schedules the thread execution. Each block is executed as a 32-thread Warps. Warps are the scheduling units in SM. Threads in a warp execute in SIMD. For example, if we have 3 blocks that are assigned to the SM and each of this block has 256 threads, then each block is divided into 256/32= 8 warps and since there are 3 blocks, we have a total of 8*3= 24 warps. All the threads in a warp are executed at the same time on the SM. When one warp is waiting for data the other warp with the data can execute on the SM. Switching between the warps does not inculcate any cost (it is zero overhead) and hence warps are the scheduling units of SM. So what should be the block dimension or the number of threads in the block for the best performance taking into consideration that 1536 threads can be executed by the SM. Let us consider three cases of block dimensions (1) 8x8, (2) 16x16, (3) 32x32.

In 8x8, we have 64 threads per block and considering that SM can have 1536 thread we have 24 blocks, but SM takes only 8 blocks. hence, if we use 8x8 we can only

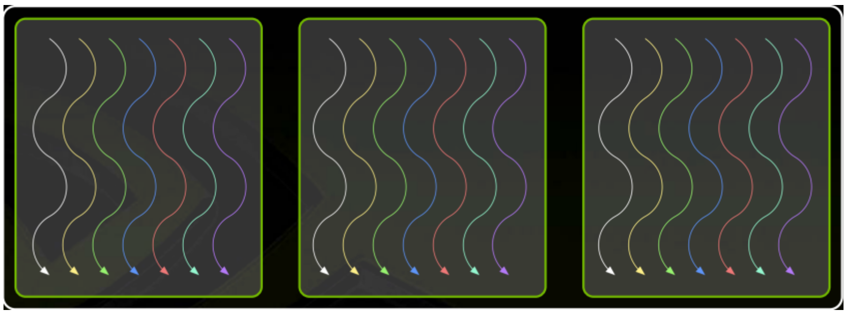


Figure 2.8: Grouping of threads in CUDA based GPU. [1]

have 512 threads at a given time in SM. For 16x16, we have 256 threads per block and considering 1536 threads supported by SM we can have 6 blocks which are at its maximum capacity of 1536 threads. In 32x32, we have 1024 threads per block and only one block will fit and hence only 1024 threads will fit into the SM, Hence, it will not achieve full potential. We can conclude that 256 threads per block run the maximum threads parallel in a Single SM.

Different GPU architectures exists due to multiple vendors such as Nvidia and AMD. For example, currently a warp in an Nvidia GPU currently consists of 32 threads whereas in an AMD GPU the wavefronts (warps) consists of 64 threads. Nvidia GPUs are the most used device and the first to introduce Compute Unified Device Architecture. Hence, following discussions will be based on Nvidia GPUs.

2.4 Machine Learning

The concept of machine learning was defined as Tom Mitchell [68] as follows:

A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P improves with experience E.

The learning process can be divided broadly into three 1) Supervised Learning, 2) Semi-supervised learning, and 3) Unsupervised Learning. Following sections give a detailed discussion on the three types of learning.

We shall establish a few notations that shall be used throughout our discussions. An input variable (input features) matrix is denoted by X and the output vector containing the target variables is denoted by Y. The pair $(x^{(i)}, y^{(i)})$ is a single training example.

The index i indicate that it is the i-th data sample, where $i\epsilon\{1, 2, ..., m\}$, where m is the total number of features for training (the number of training examples). In supervised learning, we learn a function $h : X \rightarrow Y$ such that $h(x)$ is a good predictor for the value of y. The function h is called the hypothesis function. When the target value y that we predict is continuous, then we term it as a regression problem. Else if the target y takes a limited range of discrete values we term it as a classification problem.

2.4.1 Linear Regression

We need to define a function $y = h(x)$, such that $y^{(i)} \approx h(x^i)$ for each training example. A major step in supervised Learning is representing the hypothesis function $h(x)$. For general linear regression problem we can define the function as $h_w(x) = \sum_{j=0}^{m} w_j x_j = W^T x$. The function $h_w(x)$ represents a family of functions represented by various parameter values of w. The function $h_w(x^{(i)})$ has to be as close as possible to $y^{(i)}$ and can be achieved by finding appropriate values for the weights w. We need to find values for w that minimises the difference between $h_w(x^{(i)})$ and $y^{(i)}$ or mathemathically we need to find w such that it minimizes the function $J(W)$.

$$J(W) = L(W) = \frac{1}{2}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2 = \frac{1}{2}\sum_{i=1}^{m}(w^T x^{(i)} - y^{(i)})^2 \tag{2.9}$$

This function is called the Loss function or the Cost function or the objective function. With appropriate values of w we can reduce the loss between $h_w(x^{(i)})$ and $y^{(i)}$. It represents the error in the prediction $y^{(i)}$ for a particular w.

Finding w is an optimization problem and one of the ways to solve this problem is using Gradient Descent. The gradient descent algorithm starts with an initial value of the weights w and iteratively updates the weight until it reaches the optimal value. The update is performed as in Equation 2.10:

$$w_j = w_j - \alpha \times \frac{\partial}{\partial w_j} J(W) \tag{2.10}$$

The updates are for each $j = 0, ..., m$ is performed simultaneously. α is called the learning rate. The partial derivative of our Loss function $J(W)$ (Gradient) can be derived as $\frac{\partial}{\partial w_j} J(W) = (h_w(x) - y).x_j$

Hence, Equation 2.10 can be rewritten as

$$w_j = w_j + \alpha \times (h_w(x) - y).x_j \tag{2.11}$$

This equation is also called as LMS update rule or Widrw-Hoff rule. For using all the input parameters on gradient descent we rewrite the above equations as

Repeat until convergence {

$w_j := w_j + \alpha \times \sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)}).x_j^{(i)}$ ($\forall j$ simultaneously)

}

The other way to solve for w is by using normal equations which does not require an iterative treatment. The normal equation is given by $X^TXW = X^Ty$. Hence,$W = (X^TX)^{-1}X^Ty$. The above equations are derived based on matrix derivatives.

2.4.2 Logistic regression

Logistic regression is a classification problem and hence only have a limited number of discrete outputs. The target output $y^{(i)}$ in logistic regression is called labels. In logistic regression $y^{(i)}\epsilon\{0,1\}$. In this technique we consider our $y^{(i)}$ to be the probability that it is $x^{(i)}$. Hence the hypothesis function needs to be redefined as follows,

$$P(y=1|x) = h_w(x) = \frac{1}{1+exp(W^TX)} = \sigma(W^TX) \tag{2.12}$$

The function $\sigma(W^TX)$ is called the sigmoid function or the logistic function. It confines the values to between $[0,1]$ for any given input.

Hence the cost function for logistic regression can be written as

$$J(W) = -\sum_i (y^{(i)}log(h_w(x^{(i)})) + (1-y^{(i)})log(1-h_w(x^{(i)})))$$

We solve for w as before using Batch gradient descent, stochastic gradient or by using normal Equations.

After finding the weights a threshold value can be used with $h_w(x)$ to determine wether output is 0 or 1.

$$y^{(i)} = \begin{cases} 1 & h_w(x) > 0.5 \\ 0 & h_w(x) < 0.5 \end{cases} \tag{2.13}$$

2.4.3 Multinomial Logistic regression (Softmax regression)

Softmax Regression problem is a logistic regression problem with multiple output targets y, so $y^{(i)} \in 1, 2, 3, 4, \ldots k$. k is the total number of classes. The response variable is still discrete but can take k different values. In this case the hypothesis should calculate the probability of y being k given when input is x. Mathematically we need an estimation of $P(y = k|x)$ for all $k \in 1, 2, 3, \ldots, k$. Hence the hypothesis will be a K-dimensional vector as in the following Equation.

$$h_w(x) = \begin{bmatrix} P(y=1|x;w) \\ P(y=2|x;w) \\ \ldots\ldots \\ P(y=k|x;w) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} exp(w^{(j)T}x)} \begin{bmatrix} exp(w^{(1)T}x) \\ exp(w^{(2)T}x) \\ exp(w^{(3)T}x) \\ exp(w^{(4)T}x) \end{bmatrix} \tag{2.14}$$

It is convenient to represent $w^{(k)}$ as an $m \times k$ matrix where $w^{(1)}$, $w^{(2)}$, $\ldots$, $w^{(k)}$ are the columns of the matrix.

The cost function is given by

$$J(W) = -\left[\sum_{i=1}^{m} y^{(i)} log(h_w(x^{(i)})) + (1-y^{(i)}) log(1-h_w(x^{(i)}))\right] \tag{2.15}$$

$$= -\left[\sum_{i=1}^{m} \sum_{k=0}^{1} 1\{y^{(i)} = k\} log P(y^{(i)} = k\} x^{(i)}; w)\right] \tag{2.16}$$

In Equation 2.16 1{.} indicates an indicator function. If the statement inside the curly brace is true then it outputs 1 else it outputs 0.

$$\mathbf{1}(x) = \begin{cases} 1 & \text{if x is true} \\ 0 & \text{otherwise} \end{cases}$$

and

$$P(y^{(i)} = k|x^{(i)}; w) = \frac{exp(w^{(k)T}x^{(i)})}{\sum_{j=1}^{k} exp(w^{(j)T}x^{(i)})} \tag{2.17}$$

the gradient can be given by

$$\nabla_{w^{(k)}} J(W) = -\sum_{i=1}^{m} \left[x^{(i)} \left(1\{y^{(i)} = k\} - P(y^{(i)} = k\} x^{(i)}; w) \right) \right] \tag{2.18}$$

With the gradient and the cost function W can be solved using gradient decent or other techniques.

2.4.4 Deep Networks

Due to the advancement of research in machine learning, there is a lot of overlap between Artificial Intelligence and machine learning. In most scenarios, the term can be used interchangeably while discussing Deep Learning. Deep learning can be classified as a part of machine learning wherein the data is abstracted with the help of multiple processing layers of hierarchy. In these layers, the data is transformed using various models and functions. Deep learning automatically learns features across the abstractions. This allows complex functions to map the input to the output directly from data, without depending entirely on features developed by humans [69]. The term, deep learning indicates large, deep neural networks. The significant difference between the neural networks in the 1980s and the current deep learning is that the data used for training has increased and the computers have increased the processing capabilities [70]. Moreover, the weights for the network are chosen cleverly than before. The old linear and non-linear algorithms didn't fare well when the size of the data increased. For higher data input the performance saturated as shown in Figure 2.9 [2].

When the hypothesis function $h_W(x)$ needs to be a non-linear, complex function to fit the data, we use Neural networks. Neural networks are based on the functioning of biological neurons in a human body. Neurons are the building blocks of a neural network. We shall not discuss the biological aspects of neural network so as not to confuse the readers. Basically a neuron is a computational unit that takes multiple input $x_1, x_2, x_3, ..., x_n$, the bias terms (the intercept terms, $+1$, generally), and outputs$h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{n} W_i x_i + b)$, where f is called the activation function. Various functions are used as activation functions. The sigmoid function is the most common activation function. The other functions used are tanh, ReLU, Leaky ReLU, and Maxout.

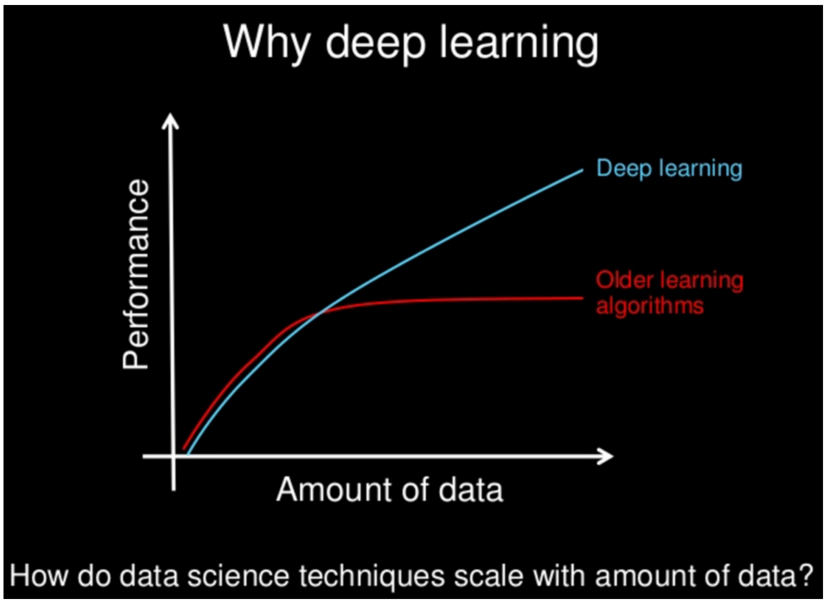


Figure 2.9: Performance comparison with data size for learning algorithms [2]

2.4.5 Deep network architectures

A neural network is created by putting together the neurons as an acyclic graph. This would mean that the output of a neuron is fed as input to other neurons. Fig illustrates a simple neural network.

A deep network is created by putting together the neurons as an acyclic graph. This would mean that the output of a neuron is fed as input to other neurons. Figure. 2.10 illustrates a simple deep neural network.

The circles in the image represent the neurons as well as the input to the neurons. The first layer is the Input layer, and the last layer is the output layer. All the layers in-between input and output is called the hidden layers. A deep network will have more than two hidden layers [69]. Each hidden layer may consist of a large variable number of neurons.

The circles in the image represent the neurons as well as the input to the neurons. The blue circle indicates the input to neurons. The circles with the $+1$ term are called the bias term or the intercept term. The leftmost layer is always the input layer, and the rightmost layer is the output layer. The output layer has been shown as a red circle in the Figure. The layers between the input layer and the output layer are called the hidden layers. In the illustration, we have one input layer (in blue), one hidden layer(in green), and one output layer (in red). We can see that the input layer has three neurons.

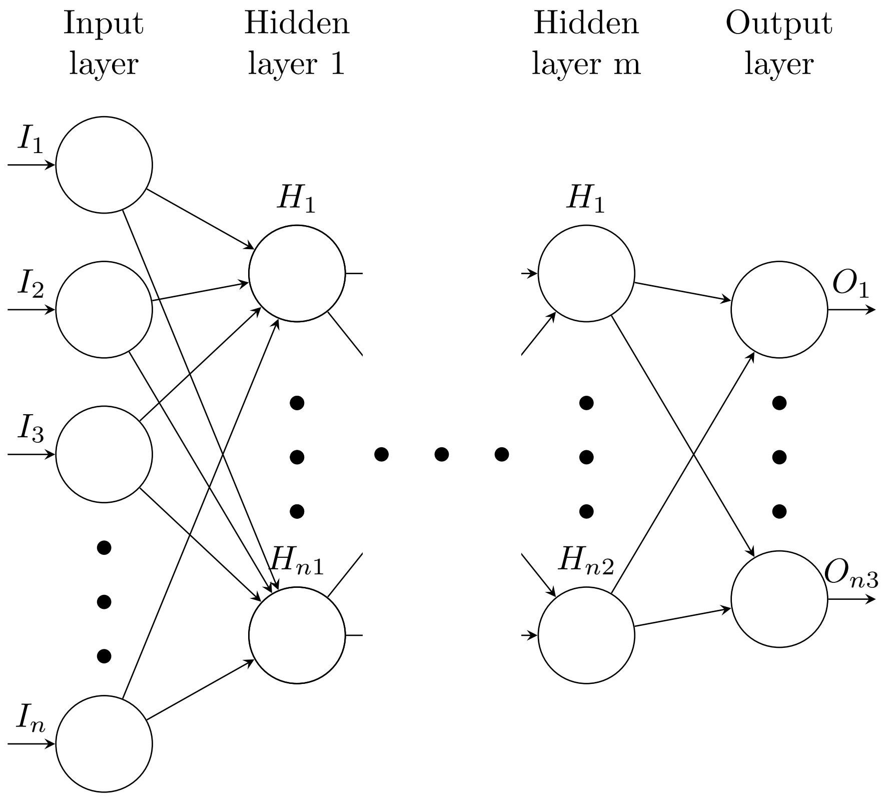


Figure 2.10: A Simple Deep Network

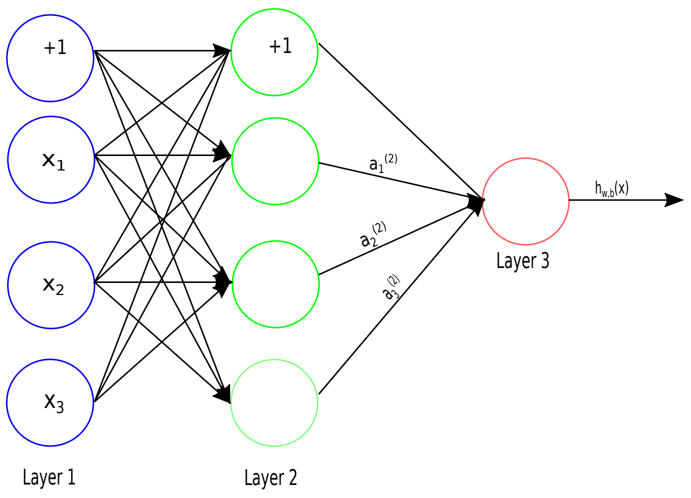


Figure 2.11: A Simple Artificial Neural Network

Hence we say it has three input units. The hidden layer has three units, and the output layer has one unit. Some of the conventions we use for our discussion has been listed below:

- n_l denotes the number of layers in our NN. In our example above $n_l = 3$
- The layers are labeled from L_1 (the input layer) to L_{n_l} (the output layer)
- The weights are represented by the letter W and the bias with b. $W_{ij}^{(l)}$ is used to denote the weight associated with the connection between the j-th unit in layer l and i-th unit in the layer $l + 1$. Hence in our example $W^{(1)} \in R^{3\times 3}$ and $W^{(2)} \in R^{1x3}$. $b_i^{(l)}$ denotes the bias associated with the i-th unit in layer $l + 1$.
- s_l is the number of nodes in layer l, without the bias nodes.
- $a_i^{(l)}$ denotes the activation of unit i in layer l. For $l = 1$, we assume that $a_i^{(l)} = x_i$. That is for the first layer we consider the first layer as the activation.

With these notations in place we can now define our hypothesis $h_{W,b}(x)$ that would produce a real number. The computation of hypothesis for our example is as follows

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \tag{2.19}$$

$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \tag{2.20}$$

$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \tag{2.21}$$

$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}) \tag{2.22}$$

If we denote the total weighted sum of inputs to unit i in layer l, including the bias term as $z_i^{(l)} = \sum_{j=1}^{n} W_{ij}^{(l)}x_j + b_i^{(l)}$. Hence we can write $a_i^{(l)} = f(z_i^{(l)})$. Extending the activation function $f(.)$ to apply to vectors in an element wise fashion would result in a more compact representation as:

$$z^{(2)} = W^{(1)}x + b^{(1)} \tag{2.23}$$

$$a^{(2)} = f(z^{(2)}) \tag{2.24}$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{2.25}$$

$$h_{W,b}(x) = a^{(3)} = f(z^{(3)}) \tag{2.26}$$

We can generalize this and compute the activation for a given layer $l + 1$ using the activations of the layer l as follows:

$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)} \tag{2.27}$$

$$a^{(l+1)} = f(z^{(l+1)}) \tag{2.28}$$

This process is called the forward propagation. We can have multiple input, hidden, and output layers. We start the computation of the activation from the first layer and use this further to calculate the activations of the next layer using forward propagation. Such neural networks are called feed-forward Neural network. Larger networks will always work better than smaller networks, but their higher model capacity must be appropriately addressed with stronger regularization (such as higher weight decay), or they might overfit.

2.4.6 Back propagation Algorithm

Given a training set of m training examples, $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, we initially define the cost function with respect to a single sample (x, y) as :

$$J(W, b; x, y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2 . \tag{2.29}$$

The above cost function is also known as squared-error cost function. The total Loss function for all m examples are calculated as

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 && (2.30) \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 && (2.31) \end{aligned}$$

The above equation consists of two terms, the first term is an average sum of squares error term. The second term is the weight decay term also called as regularization term that prevents overfitting by decreasing the weights. We observe that the bias term $b_i^{(l)}$ has not been regularized. λ is called the weight decay parameter. The above cost function is generally used for regression as well as classification. The cost function that we have used above is also called the Quadratic Loss, mean squared, maximum likelihood and Sum squared error or Cost. The gradient of the above cost function with respect to the output can be expressed as $\nabla_a J_{MST} = (a^{nl} - y^{(i)})$. There are other Loss functions that can be used. We list some of the major loss functions that could be used with a neural network in Subsection 2.4.8. Once we find the weights we can use a test set to predict the output values of new data.

2.4.7 Activation Functions

2.4.7.1 Sigmoid

The sigmoid function can be mathematically written as $\sigma(x) = 1/(1 + e^{-x})$. It takes a real-valued number and squashes the number between the range [0,1] as shown in Fig. 2.12. Hence the output from the sigmoid function is generally considered as the probability, $P(y_i = 1 \mid x_i; w)$, the probability that the output is true given the input x_i for the weights w. It can be seen from the figure that, large negative numbers become (approximates to) zero, and large positive numbers become 1. The sigmoid function has two major drawbacks [CS231].

1. Sigmoid saturation and death of gradients: When the output from sigmoid saturates at 0 or 1, the gradients at these regions becomes zero. During the back propagation phase, we compute the product of the local gradient and the gradient of the current neurons output. Precaution should be taken that the initial weights of the neurons should not be greater as higher initial weights saturate the neurons and the network will not learn.

2. Non-zero centered outputs from sigmoid: If the output from the following layers in the neural network is non- zero centered then later during gradient descent then the data input to neurons are positive, then during backpropagation, the gradients on weight w will all be positive or all negative. This issue is less severe than the saturation issue since the final update of weights still can have variable signs when the gradients are added across a batch.

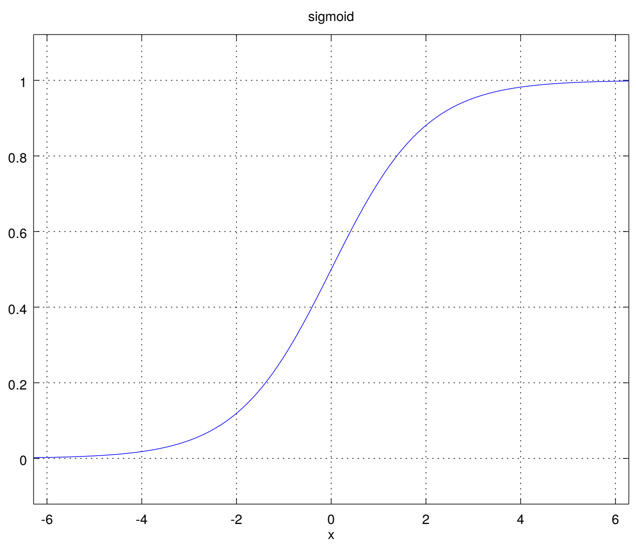


Figure 2.12: Sigmoid function

2.4.7.2 Tanh

The tanh is mathematically given by $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$. Where $\sigma()$ is the sigmoid function. The tanh function produces the output between the range [-1,1] as shown in the Fig. 2.13

2.4.7.3 ReLU

Recent researches [71, 72] point to the use of rectified linear units for better performance for deep neural networks. The ReLU is not bounded nor is it continuously differentiable. The gradient of the ReL function doesn't vanish as we increase x. In fact, for max function, the gradient is defined as $f(x) = \begin{cases} 1 & x > 0 \\ 0 & otherwise \end{cases}$. It can also be represented as $f(x) = \max(0, x)$. Figure. 2.14 represents the ReLU function.

The major advantages of ReLU are as follows

1. No gradient vanishing problem like sigmoid and tanh. It does not saturate and hence Accelerates the convergence of gradient decent compared to sigmoid and tanh functions.

2. ReLU doesn't have expensive operations. It consists of simple thresholds that is easy to implement.

3. Induces sparsity in the hidden Units.

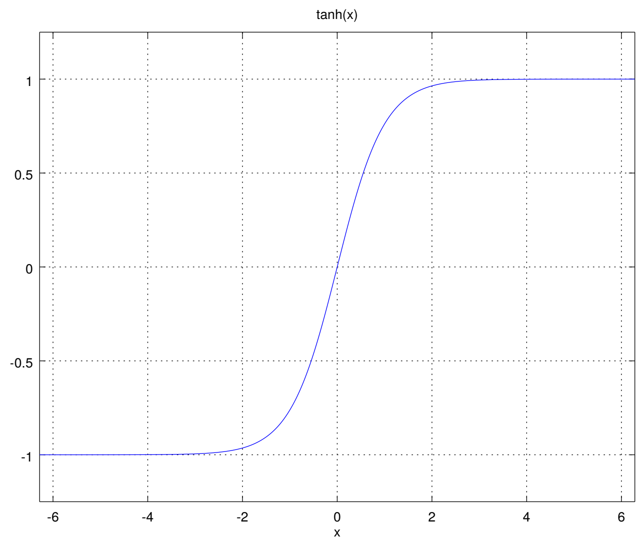


Figure 2.13: Tanh function

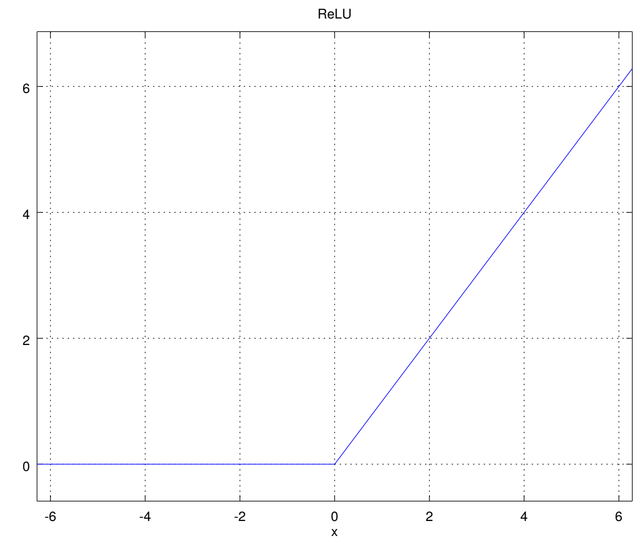


Figure 2.14: ReLU function

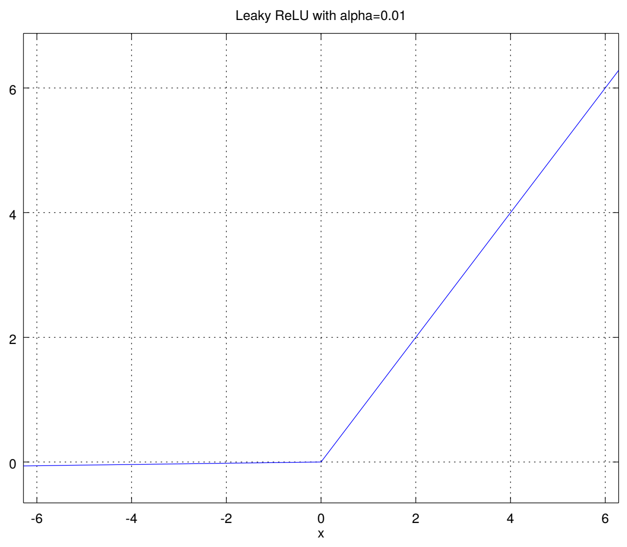


Figure 2.15: Leaky ReLU function

But a disadvantage of ReLU is that it can easily die during the training phase. A large gradient flowing through a neuron might result in the updating of weights in such a way that the threshold is never reached and it never gets activated. This kills of the neurons since all the gradient flow through it will be zero. However, this can be controlled by appropriately setting the learning rate. Higher learning rates may result in larger deaths in the network.

2.4.7.4 Leaky reLUs

Leaky ReLUs were proposed to rectify the dying issue of the ReLU [73]. Instead the function outputting zero when x<0, the leaky ReLU has a small negative slope ($\alpha = 0.01$). Hence Leaky ReLUs can be represented by $f(x) = 1x < 0(\alpha x) + 1x >= 0(x)$, where α is a small constant. Which is the same as $f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$. The negative slope can also be made into an input parameter for each neuron as in [74]. Such activation functions are called PReLU. However, the stability of this function needs to be further analyzed and studied. Leaky ReLUs are plotted in Fig. 2.15. Another variant of leaky ReLU is called the randomized ReLU [75]. In Randomized ReLU, the slopes of negative parts are randomized in a given range in the training, and then fixed in the testing.

2.4.7.5 Maxout

It generalizes the Leaky ReLU and ReLU. It was introduced in [76]It computes the function $\max(w_1^T x + b_1, w_2^T x + b_2)$. ReLU and Leaky ReLU are special cases of maxout. The only drawback is that it introduces a large number of parameters for each neuron leading to a high number of parameters.

2.4.8 Loss Functions

Cross-entropy cost They are also known as Bernoulli negative log-likelihood and Binary Cross-Entropy $-\sum\limits_j [y_j^{(i)} \text{ ln } a_j^{nl} + (1 - y_j^{(i)}) \text{ ln } (1 - a_j^{nl})]$ The gradient of this cost function with respect to the output of a neural network and some sample i is: $\nabla_a J_{CE}(W, b, x^{(i)}, y^{(i)}) = \frac{(a^{nl} - y^i)}{(a^{nl}+1)(a^{nl})}$

Exponentional cost This requires choosing some parameter β that you think will give you the behavior you want. Typically you'll just need to play with this until things work good. It is represented as $J_{EXP}(W, b, x^{(i)}, y^{(i)}) = \beta \text{ exp}(\frac{1}{\beta} \sum\limits_j (a_j^{nl} - y_j^{(i)})^2)$ where $\exp(x)$ is simply shorthand for e^x. The gradient of this cost function with respect to the output of a neural network and some sample i is: $\nabla_a J = \frac{2}{\beta}(a^{nl} - y^{(i)}) J_{EXP}(W, b, x^{(i)}, y^{(i)})$.I could rewrite out J_{EXP}, but that seems redundant. Point is the gradient computes a vector and then multiplies it by J_{EXP}.

Hellinger distance This distance is represented as $\frac{1}{\sqrt{2}} \sum\limits_j (\sqrt{a_j^{nl}} - \sqrt{y_j^{(i)}})^2$. This needs to have positive values, and ideally values between 0 and 1. The same is true for the following divergences. The gradient of this cost function with respect to the output of a neural network and some sample i is: $\nabla_a J = \frac{\sqrt{a^{nl}} - \sqrt{y^{(i)}}}{\sqrt{2}\sqrt{a^{nl}}}$

Kullback–Leibler divergence This technique known as Information Divergence, Information Gain, Relative entropy, KLIC, or KL Divergence. Kullback–Leibler divergence is typically denoted $D_{\text{KL}}(P \| Q) = \sum_i P(i) \text{ ln } \frac{P(i)}{Q(i)}$, where $D_{\text{KL}}(P \| Q)$ is a measure of the information lost when Q is used to approximate P. Thus we want to set $P = y^{(i)}$ and $Q = a^{nl}$, because we want to measure how much information is lost when we use $a_j^{(i)}$ to approximate $y_j^{(i)}$. This gives us $J_{KL}(W, b, x^{(i)}, y^{(i)}) = \sum\limits_j y_j^{(i)} \text{ log } \frac{y_j^{(i)}}{a_j^{nl}}$. The other divergences here use this same idea of setting $P = y^{(i)}$ and $Q = a^{(nl)}$. The gradient of this cost function with respect to the output of a neural network and some sample i is: $\nabla_a J = \frac{y^{(i)}}{a^{nl}}$

Generalized Kullback–Leibler divergence is given by $J_{GKL}(W, b, x^{(i)}, y^{(i)}) = \sum\limits_j y_j^{(i)} \log \frac{y_j^{(i)}}{a_j^{nl}} - \sum\limits_j (y_j^{(i)}) + \sum\limits_j (a_j^{nl})$. The gradient of this cost function with respect to the output of a neural network and some sample i is: $\nabla_a J = \frac{y^{(i)} + a^{nl}}{a^{nl}}$

Itakura–Saito distance TheLoss is given b $\sum_j \left(\frac{y_j^{(i)}}{a_j^{nl}} - \log \frac{y_j^{(i)}}{a_j^{nl}} - 1 \right)$. The gradient of this cost function with respect to the output of a neural network and some sample i is:

$\nabla_a J = \frac{y^{(i)} + \left(a^{nl}\right)^2}{\left(a^{nl}\right)^2}$,Where $\left(\left(a^{nl}\right)^2\right)_j = a_j^{nl} \cdot a_j^{nl}$. In other words, $\left(a^{nl}\right)^2$ is simply equal to squaring each element of a^{nl}.

Next we identify the parameters W and b that minimizes the cost function $J(W, b)$. Initially to train the neural we need to initialize the weights and bias. We don't initialize it to zero unlike linear regression or classification to avoid the problem of symmetric ways. Rather we initialize $W_{ij}^{(l)}$ to a random value in the range $[-\epsilon, \epsilon]$ or according to $Normal(0, \epsilon^2)$ distribution to break the symmetry. ϵ can be a small value like 0.01. Moreover the cost function $J(W, b)$ that we defined is a non-convex function that might lead to local optima. We can use any optimization algorithm to solve for W and b. We shall discuss Batch Gradient descent and we shall list out the rest of techniques that can be used to solve for W and b, while using back propagation. In the gradient descent algorithm we update the values of of the parameter W and b iteratively. Each iteration can be represented mathematically as

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \tag{2.32}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b) \tag{2.33}$$

Here α is the learning rate. Next we shall use the back propagation algorithm that effectively finds the partial derivatives required in the above equation. The derivative of the the overall Cost function $J(W, b)$ can be computed as

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \tag{2.34}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \tag{2.35}$$

Where $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y)$ and $\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y)$ are partial derivatives with respect to single example (x,y). We can see from the equation that once we calculate $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y)$ and $\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y)$ we easily calculate the derivative of the total cost. The derivative with respect to the bias term does not include the regularization term. When we feed input(x,y) at input layer, each layer after that computes the activations until the hypothesis is calculated in the final layer. This is called the forward pass. Now we compute the error $\delta_i^{(l)}$, for each node i in layer l . This would tell us the role of each node in introducing errors in outputs.

The Back-propagation algorithm can then be written as following:

1. Initially we perform a forward pass that computes the activations for the layers $L_2, L_3, \ldots.., L_{nl}$.

2. For each ouput unit i in layer assign,

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}) \tag{2.36}$$

   In the vector notation this can also be rewritten as

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)}) \tag{2.37}$$

3. For $l = n_l - 1,\ n_l - 2,\ n_l - 3, \ldots, 2$

   (a) For each node i in layer l set,

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)}) \tag{2.38}$$

   or in the vectorized form we have

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)} \right) \bullet f'(z^{(l)}) \tag{2.39}$$

4. Next we compute the required partial derivatives as

$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)} (a^{(l)})^T, \tag{2.40}$$

$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}. \tag{2.41}$$

The vectorized equations would be

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T, \tag{2.42}$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}. \tag{2.43}$$

In the above equations $\bullet$ denotes the hadamard product which is the element wise product operator. For example if $A = (a_{ij}) \in R^{m \times n} and B = (b_{ij}) \in R^{n \times p}$ then $A \bullet B = a_{ij}.b_{ij}$. Moreover in the above equations we have extended the definition of $f(.)$ and $f'()$ to mean element wise application to vectors. That is $f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$. Here $f(.)$ is the activation function.

2.4.9 Optimization functions

We list out techniques that can be used for parameter upgradation (W and b) that can be used in place of gradient descent. A detailed analysis of these techniques can be found in [75, 77]

- Stohastic Gradient Descent: [78, 79]
  - Vanilla Update
  - Momentum update
  - Nesterov Momentum
- Quasi-Newton methods [80]
  - BFGS [81]
  - L-BFGS [81]
- Adam (Adaptive Moment estimation) [82]
- Conjugate Gradient [83]
- Genetic Algorithm
- Twiddle (A realy simple but cool algorithm)
- Simulated Annealing.

Chapter 3

Literature Review

3.1 CPU Sparse Storage Schemes

3.1.1 Serial Schemes

Abdali and Wiset [84] performs quantitative analysis on the suitability of the usage of the quadtrees for the representation of both sparse and dense matrices as compared to the linear storage. They use the REDUCE [85] implementation of Quadtrees as well as linear representation of matrices. They perform arithmetic operations such as addition and subtraction as well as inversion of matrices using matrices stored linearly and in Quadtrees. They use dense, triangular, tridiagonal and diagonal matrices (both symbolic and numerical). They conclude that Quadtrees are suitable for sparse matrices and provide speedups in matrix operations and inversion. The authors do not analyze the space occupied these storage schemes. The authors conclude by saying that it is an efficient parallel structure and there is a need for optimal algorithms for Quadtree operations.

Dvorsky and Kratky propose [86] a Multi-dimensional sparse storage based on R-trees and BUB trees [87]. In this approach, space for storing the indexes are higher than CSR but have an a better access rate than them. A detailed comparison nor analysis is provided by the authors.

Balk et al. [88] propose a sparse matrix storage scheme using balanced binary search tree such as AVL Red-Black tree. Since these data structures have an access time of $O(ln(n))$, the authors consider it better than normal matrix representation. The authors compare the time taken for this scheme against traditional hash map memory allocation. The authors do not provide any details about the actual implementation of the scheme nor about the algorithm used for the multiplication. Moreover, they do not discuss the suitability of this scheme for parallel architectures.They also do not provide a comparison with other traditional sparse matrix storage such as COO and CSR schemes.

Abu Hanif and Azharul Hasan [89] propose an n- dimensional sparse array storage scheme called generalized Row/Column storage (GCRS/GCCS). Traditional storage schemes such as CSR and COO requires (n+1) arrays to store an n-dimensional array. When the number of the dimensions increases these data structures becomes unusable due to the increase in sparsity. Moreover, time and space complexity increases. The proposed scheme maps a multidimensional array with dimensions greater than two to a 2-D array using a mapping algorithm written by the authors. The total possible permutations of generating a mapped 2-D structure are n(n!). The authors fit in odd indices along the row and even indices along the column. Using the above rule they develop a mapping algorithm for array indices. This enables higher sequential access of the memory compared to traditional n-dimensional CSR/CCS. The resultant 2-D structure is stored like CSR/CCS technique which would imply that an n-dimensional sparse array can be stored using just three vectors. If we consider a normal multidimensional array of size n, then we can transform it into two dimensions by converting $\ulcorner n/2 \urcorner$ dimensions along the row and the other $n/2$ dimensions along the column. Since the multidimensional arrays are stored in a linear fashion in the memory it is easy to map the array indices to the memory address. For example, If we consider a for-dimensional array $A[l_1][l_2][l_3][l_4]$ then a tuple of indices $< x1, x2, x3, x4 >$ can be mapped as $f(x1, x2, x3, x4) = x_1l_2l_3l_4 + x_2l_3l_4 + x_3l_4 + x_4$. In the proposed algorithm the new dimensions $l_1^{'}$ consists of contribution from l_1 and l_3 and $l_2^{'}$ consists of contribution from l_2 and l_4. This is tested on normal Xeon CPU. The proposed technique has a better time and space complexity compared to CRS/CCS for n-dimensional arrays.

Simecek et al. [90] propose a minimal Quadtree format (MQT) for compressed sparse matrix storage. Quadtree based SpMV has not been explored. Rather the scheme has been proposed as an alternative to text-based storage on disk for the compressed sparse matrix. The authors create a sparse storage scheme based on Advanced Quadtree structure proposed earlier by the authors [91]. The major addition of this storage format is ease in adding and deleting values in the matrix and improves the memory utilization. The maximum size of the minimal quadtree for a sparse matrix of size N and with nnz non-zeroes is given as

$$4.nnz(\frac{1}{3} + log_4(N^2/nnz)) \tag{3.1}$$

The worst time complexity for the conversion from CSR to MQT is given by

$$\Theta(nnz(1 + \frac{N}{\sqrt{nnz}}) \times log_2 nnz.per.row) \tag{3.2}$$

Yuan et al. [92] propose two improved storage formats for sparse matrices based on DIA format. The proposed scheme is a combination of DIA and CSR techniques. The main objective of this scheme is to improve the DIA scheme for matrices that have less number of dense diagonals. The first proposed algorithm is called DDD-Naïve which omits the index array in the CSR and stores in the DIA format the diagonals. The second proposed technique called DDD-Split splits the sparse matrix, as well as the result vector into row blocks and the identical contiguous values in the diagonals, are compressed and each row can be further split into sub-block rows. This scheme has not been implemented on GPU. Both the schemes were implemented sequentially by the authors on the CPU.

Langr et al. [93] propose an Adaptive Hierarchical blocking format for the storage of sparse matrices. In HSF's (Hierarchical Storage Formats) the matrix is divided into blocks and these blocks can be stored as a two level hierarchical data structure. The first level, level-0, is used to store the nnz blocks, which are stored as sub-matrices of the original sparse matrix and level-1 consists of a block matrix that stores pointers to the blocks in the first level. Some of the storage formats that fall under HSF are BCRS [35, 94], BCSR [95], HiSM [94, 96], COOCOO [97], SBSRS [98], SPBCRSX [99], CSRCOO [97], CSB [100], etc. All these techniques named above produces variable performance in time and space depending upon the structure of the input matrix. For a single matrix use of different schemes to store different blocks results in an increase in performance. In the proposed technique for each block an efficient storage scheme is selected out of CSR, COO, dense and Bitmap and for the block matrix COO format is used. This produces a storage format that has lower space complexity than using CSR, COO, Dense, or Bitmap scheme individually. This storage scheme has not been tested using parallel SpMV based kernels or any other parallel architectures.

Vuduc [101] proposes a sparse storage scheme based on Diagonal storage format called the Row Segmented Diagonal storage format. In this format, the input matrix is divided into row slices such that each slice consists of diagonal elements more than one or equal to the number of rows in a slice. The total number of slices can be denoted by s. This storage scheme consists of an integer array, *slice_ptr*, which stores the starting index of each slice, and an integer array *num_ diag*, of size, s that stores the number of diagonals in each segment, an integer array, *src_ ind*, to store the starting column indices of the diagonals and a float array data that stores the *nnz* values in the matrix. Another parameter u, which represents the unrolling depth that is used while storing *nnz* in the data array. The first u values are initially stored from the first diagonal and the first u *nnzs* from the second diagonal and so on. Then the next u elements from the first diagonal are stored and the next u from second diagonal and so on until the final *nnzs* are stored.

Willcock and Lumsdaine [102] propose an improvement to the sparse matrix SpMV by proposing new sparse storage formats that are based on lossless compression. The first of such technique is called the Delta Coded Sparse Row (DCSR) that is based on byte-oriented delta encoding. This technique six commands to encode the matrix and each of the command code takes a parameter. The compressed matrix then can be represented as a list of <command, parameter> pairs. Three such pairs encoded into 32-byte word. A decompression algorithm is applied before SpMV operations. The second proposed scheme is called the Row Pattern CSR which provides a better compression, compression time, and performance as compared to DCSR. In this scheme, the delta values between the consecutive nnzs are grouped into intervals. The intervals are further grouped into multiple intervals in a group which are called patterns. These patterns are stored in the compressed matrix along with delta values for decompression.

Vuduc and Moon [103] extend the classical BCSR to store sparse matrices with unaligned multiple structures. They term this new format as UBCSR (unaligned block compressed sparse row). In this format the matrix is split as the sum of sub-matrices and each sub-matrix is stored using the UBCSR format. Initially variable block row format (VBR) [104, 105] is used to partition the matrix into rows of blocks. This is done to reveal the sub-structure of the matrix. The major difference between BCSR and UBCSR is that UBCSR relaxes row and column alignment. The blocks in UBCSR can have variable size. To handle this it uses one extra pointer array to point to the start of each block as compared to the traditional BCSR format for sparse matrices. This scheme was not designed for heterogeneous multiprocessor computing.

3.1.2 Parallel Schemes

Zhang et al. [106] Proposes a sparse matrix storage method based on Quadtrees and cache memory for the purpose of matrix-vector multiplication. The two features that the authors optimize in this scheme are:

- Data locality (for the multiplication)
- Parallel performance

In this technique, the authors' recursively subdivide the sparse matrix into smaller sub-regions using quadtrees so that the sub-regions can be stored in the cache to improve the data locality. The conversion complexity to COEQT from general methods such as COO, CSR can be generalized as $O(2n\,log_2 n + n\theta)$. We can see that the complexity of this conversion is higher than the complexity of matrix-vector multiplication for which it was proposed. We can also observe that the conversion cost is dependent on the size of the matrix. The authors further propose a Matrix-Vector multiplication based on this storage. It was also run in a distributed system consisting of 1140 blade nodes each having 32 GB memory. The authors conclude that the CEQT achieves 1.5 times the speed up for Matrix-vector multiplication as compared with CSR. The authors do not mention or perform any analysis on the memory consumed by CEQT.

Guo and Gropp [107, 108] propose a sparse storage for SpMV on IBM POWER architecture. The IBM POWER architecture consists of a special hardware component stream called the pre-fetch data stream that would increase the memory bandwidth significantly. The authors have developed a new sparse storage that efficiently use this data stream. Hence, the data is called as Stream-CSR (S-CSR). The proposed scheme is based on CSR. The main idea behind this format is to allow multiple groups to prefetch independent data streams without overlap so that the prefetch can be increased to accelerate the performance. This format is dependent upon the number of streams in the architecture.

3.2 FPGA schemes

Smailbegovic *et al.* [99] proposes a sparse matrix storage format called extended Sparse Block Compressed Row Storage (SBCRSx). The format is based on Block Row Compressed Format [109]. The format is designed keeping in mind the developments in FPGAs. The technique provides additional details about the sparse matrices to the hardware to assist efficient implementation of operations on the matrices. The main idea of this scheme is that the value and the indices should be accessed consequently unlike most current storage techniques in which the indices and values are stored separately. Hence, the authors propose that the values and the indices be stored in a list. The elements in the same row will have their individual lists linked to each other. This results in the creation of chunks or short vectors with known lower and upper bound addresses. The number of chunks that can fit on the chip at any given time cannot be fixed due to the differences in the length of different chunks. Hence, the authors augment an extra bit to each list to indicate whether there are extra elements following it in a given row. This bit is called chaining bit. A chain bit of 0 indicates that no elements are there in the row after the current element. The authors provide an algorithm to convert a normal dense sparse matrix to this new storage. They don't provide any qualitative or quantitative analysis on the effectiveness of this scheme.

3.3 MIC schemes

Lu and Vinter [110] propose a storage format for Sparse matrices from the point of view of matrix-Vector multiplication called as CSR5. The 5 in the name indicates that there are five arrays stored unlike the three in the classical CSR technique. The main objective of proposing this technique was to improve the performance of Matrix-Vector multiplication. The designed data storage scheme is insensitive to the sparsity of the data structure provides similar performances for sparse as well as dense matrices. It has a lesser conversion cost when converting from existing techniques such as COO and CSR. This technique uses a combination of row block method [111] and segmented sum method [112]. This technique provides proper load balancing and reduces overhead due to memory access and synchronization. They do not provide any discussion or analysis on the space occupied by this storage scheme. They show that this scheme accelerates the matrix-vector on various platforms such as on GPUs and CPUs as compared to CSR, ACSR, ESB, and pOSKI.

Kreutzer [113] proposes a unified sparse matrix storage for all modern processors with wide SIMD units. They propose a storage format that would provide efficient performance in all processors such as Intel multicore processors, Intel MICs, and CUDA architectures. The proposed format is based on the sliced ELL combined with SIMD vectorization and is called the SELL-C-σ. The C (chunk) indicates the width of the SIMD registers in the x86 architectures and on CUDA architecture it is the number of threads per warp. σ is called the sorting scope that is the number of rows that will be sorted to reduce the overhead and increase the performance. The rows within a chunk are stored column wise and the chunks are stored one after another. The rows in chunks are padded to that of the largest row similar to Sliced ELL. To avoid this head selective sorting is applied using σ parameter. But this does affect the performance of the iterative solvers. The CUDA based implementation has not been discussed in detail. Only the results are shown. Whereas, the parallel implementation discussed is mainly that of OpenMP.

Liu *et al.* [114] propose an efficient sparse matrix-vector multiplication on Intel Xeon Phi (MIC) coprocessor by developing a new sparse storage format called ELLPAK Sparse Block (ESB). The proposed new format is an extension of ELLPACK format that overcomes the issues faced when running CSR based SpMV on Xeon Phi coprocessor. In addition to the regular ELLPACK format, column blocking is used to improve locality of memory access. Moreover, in each of the column, a finite window sort is applied to improve the Single Instruction Multiple Data efficiency. To reduce the bandwidth, the authors encode the indices of the non-zeroes into a bit array. The use of bit arrray degrades the performance for some matrix. The performance is still dependent upon the structure of the matrix.

3.4 GPU Sparse Storage Schemes

In this section we shall discuss the state-of-art Sparse matrix storage schemes desinged for SpMV on GPUs.

ELL-R was introduced in [115] to improve the performance of the ELL storage

$$A = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 0 & 4 & 5 & 0 \\ 6 & 0 & 7 & 0 \\ 0 & 2 & 8 & 9 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 2 & 3 & * \\ 4 & 5 & * & * \\ 6 & 7 & * & * \\ 2 & 8 & 9 & * \end{bmatrix} \quad indices = \begin{bmatrix} 0 & 1 & 3 & * \\ 1 & 2 & * & * \\ 0 & 1 & * & * \\ 1 & 2 & 3 & * \end{bmatrix} \quad rl = \begin{bmatrix} 3 & 2 & 2 & 3 \end{bmatrix}$$

Figure 3.1: ELL-R representation of matrix A

format on GPUS. Similar to the ELL format it has float MxK array to store the values and an integer $M \times K$ array to store the column indices. In addition, it has an integer array,$rl[]$, of size M to store the length of each row. This additional data structure helps in the reducing the performance degradation due to thread divergence caused by non-zero entries in the ELL storage format. The row length information enables the skipping of non-zero values without performance degradation. The total number of rows (N) has to be a divisor of the block size for coalesced global memory access. Hence extra rows with zero values are padded are achieve coalesced memory access. There is no thread divergence within a warp of threads which improves the performance. If the new row size after padding is N', then the total number of bytes required to store ELL-R format for double- precision valued matrices is: $2NK * 8bytes + N' * 4bytes$. An example of this storage format for a 4x4 sparse matrix is illustrated in Fig.3.1.

JAD format [116, 117] can be considered as an optimization of ELL format on GPUs. Initial preprocessing is required to store a sparse matrix in JAD format. The sparse array A is sorted initially based on nnz per row in the descending order. After sorting, the zeroes are removed from the array and the non-zero elements are shifted left. The columns of the newly form structure are called the Jagged diagonals. Four arrays are used to store a matrix in JAD format: (1) A float array, data, stores the non-zero elements in column-major, (2) An integer array indices is used to store the corresponding column indices of data, (3) an integer array, ia is used to store the starting indices of each jagged diagonal and, (4) an integer array perm, to store the row permutations. JAD provides better performance in SpMV kernel than ELL, but the occupancy in JAD is lesser than ELL. An example of JAD storage format is illustrated in Fig.3.2.

ELLR-T is based on GPU kernel modification of ELLR [118, 115]. In ELLR format,

$$A = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 0 & 4 & 5 & 0 \\ 0 & 0 & 7 & 0 \\ 1 & 2 & 8 & 9 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 2 & 8 & 9 \\ 1 & 2 & 3 & * \\ 4 & 5 & * & * \\ 7 & * & * & * \end{bmatrix} \qquad indices = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 3 & * \\ 1 & 2 & * & * \\ 2 & * & * & * \end{bmatrix}$$

$$ia = \ 0 \quad 4 \quad 7 \quad 9 \qquad perm = \begin{bmatrix} 3 & 0 & 1 & 2 \end{bmatrix}$$

Figure 3.2: JAD representation of matrix A

each row is handled by a thread during SpMV, whereas in ELLR-T, T number of threads operate on the row at the same time. Here T can be 1, 2, 4, 8, 16, or 32. To achieve this, preprocessing is performed on ELL-R format, which includes permuting the non-zero values and their column indices and adding zero padding so that each row is a multiple of 16. This ensures coalesced global memory access. The addressing scheme in ELLR-T and sliced ELLR are quite different even though in both schemes multiple threads operate per row. To calculate the value of $x[i]$, the$i - th$ row is split into T parts and T threads compute $\ulcorner rl[i]/T \urcorner$ iterations of the inner loop of SpMV. The partial computation performed by each thread is stored in the shared memory. Reduction in shared memory is performed compute the final $x[i]$.

Monakov et al. [119] proposed Sliced ELLPACK to reduce the redundancies in the ELL format. The input matrix A is partitioned into S slices with each slice consisting of a fixed number of adjacent rows. Each slice in this format is stored separately in ELL format. The slicing reduces the difference between the shortest and longest row resulting in a reduction of redundancy in zero padding. One CUDA thread block can be assigned to each slice for CUDA implementation.

Lu and Vinter [110] propose a storage format for Sparse matrices from the point of view of matrix-Vector multiplication called as CSR5. The 5 in the name indicates that there are five arrays stored unlike the three in the classical CSR technique. The main objective of proposing this technique was to improve the performance of Matrix-Vector multiplication. The designed data storage scheme is insensitive to the sparsity of the data structure provides similar performances for sparse as well as dense matrices. It has a lesser conversion cost when converting from existing techniques such as COO and

CSR. This technique uses a combination of row block method [111] and segmented sum method [112]. This technique provides proper load balancing and reduces overhead due to memory access and synchronization. They do not provide any discussion or analysis on the space occupied by this storage scheme. They show that this scheme accelerates the matrix-vector on various platforms such as on GPUs and CPUs as compared to CSR, ACSR, ESB, and pOSKI.

A modification of existing HYB format from the CUSP library [64], for storing sparse matrices, named HYBIV (The HYB with Indexed Valued format) is proposed by Bylina et al [120]. The major change they try to inculcate in the algorithm is the sparse access of the slow GPU and put a transition matrix as large as possible in the GPU memory. The conventional HYB format is a combination of both the COO technique and ELLPACK technique [121]. Hence, it consists of 5 arrays to store the matrix (3 from COO and 2 from ELLPACK). In this format, we store all the unique elements in the transition matrix in GPU read memory as a new vector, data. The row and columns indices are stored in separate arrays, like the HYB and COO. A memory and time comparison was performed against ELL and CSR. It was found that the memory required for larger matrices are reduced by half for HYBIV as compared to HYB. They store Markov chain transition matrix and perform SpMVM with a vector.

Neelima and Raghavendra [122] propose a sparse matrix representation scheme aimed at improving the performance of sparse matrices on GPU named Column only SPaRse Format (CSPR). In total, this scheme uses two data structures, one for storing data and the other for storing indices of data. The authors use a single data structure to embed both the row and column information. The row and column indices are combined and represented as a single value as in Row-major representation of the matrix. The major advantage of this technique is the use of one data structure for sparse matrix access but the disadvantage is that the decoding of the data structure into row and column takes extra computation and time. Rather than using a CUDA thread for a single row the authors use a thread each for each element in the matrix so as to increase the continuous memory access. The authors don't use texture memory. CSPR is only applicable for unstructured matrices which don't have uniform row lengths. This storage technique only seems to perform well when the matrix has a large number of rows with less non-zero elements per row and in the absence of dense row at the center of the matrix. If these conditions are satisfied then this storage scheme provides a better performance, CPU to GPU memory transfer speed, and throughput compared to COO scheme. Again the authors don't discuss the memory comparison with other techniques.

Wijs and Bosnacki [123] propose three schemes to store Markov chain transitional matrices aimed at GPU. They also discuss SpMV multiplication algorithms for these schemes. To access continuous memory in the GPU, they propose enumerating the non-zero elements of the sparse matrix using breadth-first search. They assign indices to the non-zero elements of the matrix using a heuristics based on BFS. In one of the scheme, they use a single thread for a single row whereas in the second scheme they use two threads parallel for a single row. The authors plot various transition matrices generated using PRISM [124] and come to a conclusion that the values are mostly diagonally located. The authors modify existing MSR storage scheme so that it supports continual memory access. In CUDA architecture, the threads that are in the same warp access continuous memory. Hence, the authors propose modifying MSR such that it is possible. The threads in warps are explicitly partitioned to enable this. The matrix is segmented into various partitions such that each segment consists of rows equal to warp size. This is implemented by adding an extra data structure that stores the start of the segment. This technique is called as full Warp technique. The other technique proposed by the authors implement half warp in which two threads are executed for a row in parallel. The authors provide a comparison with COO in terms of performance and memory.

The memory-boundness is a major issue in the development of GPU-based parallel sparse linear solvers. Many authors have proposed a number compression schemes for compressing the sparse matrices but these schemes are not quite feasible due to the serial nature of most of the compression and decompression algorithms. Hence Tang et al. [125] propose a family of bit representation-optimized (BRO) sparse storage scheme for sparse matrices representation on GPUs. The proposed formats are BRO-CSR, BRO-ELL, and BRO-HYB. They improve the SPMV on GPUS by reducing the memory traffic by compressing the index data. They further propose two other BRO Hybrid scheme that has significant performance improvement. The authors use lossless compression to reduce the amount of storage required by the matrices and hence reduce the memory bandwidth. These proposed techniques have been tailored especially for performance in GPUs. The main idea of this technique is to compress the indices. There is a hefty redundancy in the indices which is reduced by the authors by representing the indices using just enough number of bits. One of the goals of the BRO scheme is that the decompression on the GPU matches the Warp execution model in GPU. Initially, the matrix rows are grouped together such that each group has a fixed number of rows omega, which is set as same as the warp size of the GPU. For each warp block, the authors apply delta encoding to column indices of each row. Delta encoding is a compression scheme normally used for signal compression. Next we determine the minimum number of bits required to represent each column that is encoded by delta encoding and store in an array named "bitalloc". The delta values of each warp block are packed together using the bit allocation information. For any given warp block the number of bits required to store the indices is the same. The compressed indices for each row are then multiplexed with the symbol length. The symbol length is a thread's data access granularity during the decompression phase. This multiplexing is to enable coalesced memory access. The compressed indices are then stored in an array named compstr. In addition, to compstr and bitalloc we have three more vectors namely val, cptr, and bptr to store the non-zero elements, to provide an index to cptr and bptr respectively. The authors apply this scheme to existing sparse storage techniques to derive the BRO version of each of these schemes. The authors test these proposed schemes against conventional CSR, HYB, and ELLPACK using Matrix-vector multiplication. Further they test the proposd storage with parallel Conjugate Gradient solver. They achived a speedup of 1.4x to 2.2x using an Nvida Tesla K20.

Koza et al. [126] discusses some of the major sparse matrix structures and their implementation in GPU, they discuss sparse structures such as COO, CRS, ELL, and HYB.

Dziekonski et al. [127] propose a new sparse matrix storage called Sliced ELLR-T. The storage scheme was designed specifically to solve complex-valued sparse linear equations in computational electromagnetics. The proposed technique is based on Sliced ELL and ELLR-T storage schemes. In this technique, the matrix A is divided into slices with S row per slice as in ELL-R. Moreover, multiple numbers of threads (T= 1, 2, 4, 8, 16, 32) operate on a single row. This format provides higher performance due to coalesced memory access, multiple threads per row and use of shared memory. This technique uses less memory for storage since each slice in the matrix is stored in the

ELLR-T format. The amount of space required by this scheme is less than ELL-R and ELLR-T but more than CSR. The bigger the matrix the better the performance of the system. The authors also try out the new concurrent feature of the Fermi GPUs which produced higher performance.

Yan et al. [128] propose a new storage form based on the HYB format, to utilize the higher bandwidth of the GPUs. For some matrices, the authors note that the number of non-zeroes in the COO part of HYB is higher than that in the ELLPACK part of the HYB. Hence, they propose a new storage scheme called HYB-R (HYB-recursive) which recursively partitions the matrix into COO and ELLAPCK. This results in higher number of non-zeroes in ELLPACK part of the HYB-R format than HYB format. Initially, we create two parts as before, the COO part and the ELLPACK part. Then non-zeroes are recursively added to the ELL part. Each new recursion is stored as a new ELL part. Therefore, the ELL part consists of smaller ELL parts and the rest will be stored in the COO part. The sparse matrix is sorted initially according to the number of non-zeroes before the recursive addition to ELL part. Hence, the structure of the data storage would look like

Each of this format is executed with different kernels. Since HYB format has a good performance on the GPU increasing the nnz in the ELL part of HYB increases the performance. The authors obtain a speedup of 10% on average. The biggest matrix that uses had a size of one million.

Dang and Schmidt et al. [129] proposes a sliced COO algorithm that gives a higher performance than the COO kernel for SPMV as in [67] and CUSP library. In this technique, in addition to the three array containing the value, column indices and row pointer we have an extra that contains pointers to the slices. Initially, the matrix is sorted based on the row weights. The values are then stored in column major format that ensures that the elements in the same columns are contagiously located. Storing the columns indices in the increasing order increases the regular access to the input vector that would increase the performance. But this results in an inability to use parallel reduction as the rows are accessed randomly. Hence, the authors sort the local non-zero elements inside each row and the slicing is done using consecutive rows. The kernel designed by the authors for SPMV utilize one CUDA block for each slice. Thus, the blockId is the same as the slice index. An atomic add operation is used to add the results in shared memory. Atomicadd synchronizes the access if more than one thread is accessing the shared memory in parallel. Reducing the size of slices results in two elements having same row indices to be processed resulting in serialization. To avoid this, multiple lanes are provided in the shared memory. The authors also propose a faster heuristic based partitioning algorithm for the matrix A into SCOO format. For most of the tests, HYB showed a good performance, for two of the test matrices the performance of HYB was better than SCOO. The space occupied by SCOO is less than HYB and COO but more than CSR.

Maggioni and Berger-Wolf [130] propose an architectural optimization based on a heuristics, capable of reducing the cache memory access within a warp, to improve the performance of SpMV kernels. They also propose am an improvement to sliced ELL, based on warp granularity and reordering. The proposed technique aims to improve the coalesced memory access and reduce the cache requests, by changing the order in which the threads compute the non-zeroes in the array. The sliced ELL originally proposed by Monakov et al. [119] assumes that each slice S, will be assigned to a CUDA block of threads for execution. i.e., they do not make a distinction between the block-size and the slice size. The slice size which is equal to the warp size produces the best performance. In such cases, assigning a slice to a CUDA block results in one SM having 8 blocks with each block having 32 (warp size) number of threads, a total of 256 threads. But an SM is capable of executing 1536 threads in an SM. This a gross underutilization of resources. To rectify this issue the block size and the warp size is decoupled from each other. The block size is set to 256 and the slice size is set equal to warp size. This leads to storing the warps offsets and the maximum row size of a slice in addition to the two arrays for value and the columns as stored in ELL format. The warp offset is stored in an integer array of size S, where S is the number of slices and the maximum row size of a slice is also stored in an integer array of size S. This results achieving a better occupancy and data-structure efficiency. The execution of a memory instruction (warp level) is converted into an L1 cache line requests in order to satisfy the request of each thread in a Fermi based architecture. The order of the non-zeroes in the ELL format affects the amount of L1 cache transactions during SpMV. The cache transaction minimization problem is the minimization of the function $z(S) = \sum_{j=0}^{k-1} |\cap_{i=0}^{w-1} S_{i.j}|$, where $z(S)$ is the sum of cache line intersections along the vertical direction in the scheduling table S.

Using greedy approach the authors find an S that minimizes z(S). The time complexity of the proposed heuristics is $O(k^2n)$.

Yin et al. [131] provides an optimization for CSR based storage format on GPUS. The CSR scalar kernel provides poor performance due to non-coalesced memory access and the CSR kernel doesn't properly utilize the resources when the number of nnz per row is less than 16 or is not a multiple of 16. In this technique, the sparse matrix A is divided into fragments. Each of these fragments is a multiple of threads in the thread block and are assigned to a single block for SpMV calculation. The partial sum of each fragment is stored in shared memory and later through reduction is written to the main memory. The authors add an extra int2 array named Bound of size S, where S is the number of fragments of A (which is equal to number of thread blocks), Bound[i] represents the block Id and Bound[i].x represents the row number of the first element in the block and Bound[i].y represents the row number of the last element in the fragment/block. Two kernels are used for SpMV calculations. The initial kernel calculates the partial results and the second kernel writes the final result.

Bisection ELLPACK [132] can constructed from a sparse matrix A as follows:

1. Split the matrix into slices of size warp size (32 threads). If the total number of rows in the matrix is not a multiple of 32, then zero padding is applied to ensure it.
2. The rows inside each strip is sorted within the slices in descending order according to the size of rows.
3. The nnz elements are shifted to the left as in JAD and ELL.
4. The columns inside the slices are divided into (log2 warpsize+1), i.e.,6 groups.
5. Each of these groups are stored in the ELL format.

Four arrays are used to represent a sparse matrix in BiELL format. A float array *data* is used to store the data and an integer array *indices* is used to store the corresponding columns of the array. Another integer array *ia* is used to store pointers to the index of the first element of each group in each slice beginning in the data array. The array *ia* has a length of $\lfloor n/warpsize \rfloor * (log_2 warpsize + 1) + 1$. Moreover, an integer array *perm* is used to store the row permutation index. The bisection of the slices into group improves the load-balancing and thus optimize the performance of SpMV. Similarly, the authors propose the bisection of the JAD format. BiELL SpMV Kernel Implementation is as follows: One warp is assigned to one slice of BiELL. Initially, each row of a group will be assigned a thread from the warp. These threads iterate over the other groups in the same slice in each iteration. Once it completes it is reassigned to the other slices. Each thread computes the partial result and stores in the shared memory. The final result is obtained with the help of reduction.

Xu et al. [114] propose a cache blocking method for optimizing sparse matrix-vector multiplication on GPUs. Initially, the sparse matrix is partitioned into multiple sub-blocks and each sub-block is stored in CSR format. This storage scheme uses four arrays to store the sparse matrix. A float array, data, of size nnz, is used to store the non-zero values, an integer array col, that stores the column index of the corresponding non-zero values, an integer array, block_ptr, that points to the start of each block row, and an integer array, row_ptr, that points to the start of each row in a sub-block. When the column size of a sub-block is small then it can be reused in the GPU cache. One row of blocks is considered as a CUDA block and threads are assigned per row in this row of sub-matrix blocks. If the columns of the block are quite large then it cannot be wholly cached and hence there is a degradation in the performance.

Zardoshti et al. [133] an auto-tuning framework for SpMV kernels on GPU which selects the best possible matrix format for the input matrix keeping in mind the architecture of GPUs. The selection of a fixed format for all matrices leads to inefficient performance. They study the effects of various GPU parameters on the performance of Sparse-vector multiplication. The studied parameters are the number of threads per block, number of registers, and memory hierarchy configurations. These parameters are analyzed for various input matrices to determine the best suitable format for that matrix. The proposed framework is a run -time adaptive system that chooses the optimal storage and SpMV kernel. During the initial training phase, an input matrix is divided into five parts and ten percentage of each sub-part is stored using all possible input storage format, and associated SpMV operation is performed. The one that provides the best performance is selected to store the entire matrix. The authors experiment and determine the best suitable matrix storage format for a number of well-known matrices.

Ashari et al. [111] discusses a new optimization of SpMV for the CSR storage technique that reduces, both space and time overhead. This new technique is named as Adaptive Compressed Sparse Row (ACSR). This technique takes into consideration, issues such as thread divergence. It also reduces extra computations as compared to ELLPACK and other storage formats that require padding. The synchronization overhead imposed by reduction operations are also reduced as compared to COO. The rows of the sparse matrix are moved into bins depending on the number of nonzeroes per row. The rows with nearly equal nnz per row are binned together. For example, the rows with one or two non-zeroes are stored in Bin-1, the rows with 3 or 4 nnzs are stored in Bin-2, etc. Each Bin is assigned a specific kernel for their execution. Each kernel will be configured to efficiently execute SpMV on those bins. For the bins that contain large number rows with nnzs, a master kernel is invoked. The master kernel doesn't perform any computation but in a nested dynamic fashion, it invokes kernels for each row in the bin. This dynamic kernel invocation is available in all Nvidia GPUs with a compute capability higher than 3.5. The bins are used to configure the number of threads per row in a kernel. A reduction operation is required to compile the partial results calculated by different threads in a row for different kernels. But this is quite fast as it is based on intra-warp and not inter-warp synchronization.

Koza et al. [134] propose a new sparse storage scheme called compressed multi-row sparse format (CMRS). In this technique, the matrix is divided into multiple strips with each strip containing a constant amount of rows. It consists of four arrays and an integer parameter. A float array data is used to store the non-zero values, an integer array index is used to store the column indices of the corresponding non-zero values in data, an integer array strp_ptr that is similar to row_ptr in CSR but rather stores the index of the starting element in each strip, and an integer array row_in_strip, of size nnz, holds the row index of each non-zero element in data within each strip. The parameter height indicates the number of rows in a strip. The conversion between CSR and the proposed format is trivial. CSMR can be considered as an extension of the CSR format. This format dynamically assigns thread per row and each strip is executed by an SIMD unit. They also implement a number of optimizations such as use of texture memory for input matrix, enlarging the shared memory by sacrificing the L1 cache, reordering the CMRS array by sorting indices array by strip indices and then by column indices within the same strip, and accessing the non-aligned portion of the and later the aligned portion.

Feng et al. [135] propose a new storage format called Segmented Interleave combination (SIC) which is based on CSR storage format. Similar to CSR the SIC format also has the same three arrays for storing the sparse matrices, namely data, indices and row_ptr to store the non-zero value, column index nd row pointers respectively. The major idea in this storage format is to combine certain CSR rows to achieve the new SIC row. For a sparse matrix of size N, c contiguous rows are selected to be combined to form the new SIC row. This will largely reduce the uncoalesced global memory access. Unlike CSR, only one warp and one global memory access are required to perform on multiplication and addition. Moreover, it is highly suitable for parallel reduction operation. The SIC matrix is sorted according to the row length and converted into segments and each segments can also be stored as SIC and different GPU kernel can be launched for each segment. Each warp will be processing c rows. This technique hence reduces the thread divergence. The number of rows executed by a warp is fixed in this format unlike [134] which dynamically decides the rows.

Guo and Wand [136]propose an auto-tuning framework that can analyze an input matrix and automatically select parameters for CUDA kernel for SpMV to optimize the performance on specific GPUs. The authors investigate the effect of a number of threads, block size and warp size on the performance of SpMV kernel. The best thread size as been deduced between [Max threads/2, Max threads], where Max threads are the total number of maximum threads supported by the GPU. The block size selection is made depending upon the model of GPU and the warp size is selected as 16 threads per warp. The matrix is read in and converted to CSR format and the framework analyzes and deduces the best possible parameters and execute the kernel with these parameters.

Yan et al. [137] introduces block COO algorithms named Blocked compressed COO (BCCOO) and Block compressed COO plus (BCCOO+). BCCOO format is an extension of BCOO format. In BCOO format, the matrix is divided into blocks and each non-zero block is stored consecutively. This results in a block having the same row and column index. This reduces the amount of space required to store the column and row indices. Similar to COO format, it stores the non-zero value, the column index and the row index in three arrays respectively. The BCCOO format introduces a new array called Bit_flag which is used to compress the row indices. The Bit_flag array is created by performing a difference operation on the row index array. A bit value of 1 indicates that it is the last element in the array whereas a bit value of 0 indicates that there are more elements in the row. This eliminates the requirement of condition check during the reduction when the partial sums of dot products are gathered. In BCCOO+ format, the sparse matrix is initially sliced vertically and the slices are aligned in top-down order. BCCOO is then applied on this newly obtained sliced matrix excluding the column index array. The column index array is generated based on the original unmodified array. The segmented scan is used by the SpMV kernel developed by the authors to get the partial results. The partial results are then added in the global memory.

Monakov and Avetisyan [138] study blocked storage of sparse matrices for sparse matrix-vector multiplication. The first technique is a combination of BCSR (Block CSR) and BCOO (Block COO) and the second technique is a combination of the initially proposed Block format and ELL scheme. In the initial block based format, the sparse matrix is divided into strips with each strip containing S number of consecutive rows. Within each slice (strip) the blocks are formed and are stored in BCOO format. The authors store the block column index and offset from top row strip in one word. For each strip, the index of the first block in the strip is stored. Each of this strip is assigned a thread block. The threads in a CUDA block is logically sub dived among the blocks in each strip and each thread execute. After the execution of all the blocks, the threads synchronize to calculate the final result. Since the blocked format requires storing zero elements, which results in degradation of performance. In order to diminish this, they propose the use of Block technique along with the ELL format. To reduce the overhead in ELL caused by extra zeroes the authors use a simple matrix reordering scheme to reorder rows with similar nnz into a single strip and they allow the number of nnz elements per row to vary between strips.

Greathouse and Daga [139] proposes a new CSR based storage format performing SpMV on AMD based processors. The proposed algorithm is named as CSR-Stream. Since the majority of the storage format use CSR format, they develop a storage format with low transformation overhead and no extra storage. The number of non-zero values that will be processed by the wavefronts (warps) is fixed and the values are streamed into local scratchpad memory. The efficiency of this technique reduces when the number of non-zeroes per row increases. Moreover, it becomes ineffective if the size of non-zero per row is more than the scratchpad memory size. To overcome this they dynamically determine whether a set of rows will be executed BY SCSR or traditional CSR. A combination of these techniques is named CSR-Adaptive.

Wieczorek et al. [140] propose a compact storage scheme for transitional matrices of Markov models. Due to the presence of high redundancy in the transition matrices, the authors propose a compression scheme to store the matrices. When the column indices are encoded differentially with respect to the row index it can be observed that many of the rows are same which leads to storing the rows in a dictionary of unique differentially encoded rows. Moreover, it was observed that the index sequence has fractal-like similarities. The authors propose a Meta run length encoding algorithm to store these sequences. Decompression is performed symbol by symbol depending upon the required values for operation. They solve for the transient state vectors of Markov chains using Uniformization algorithm on GPU.

Feng et al. [141] propose a new storage format called the Segmented Hybrid ELL + Compressed Sparse Row (SHEC) to improve the throughput and memory footprint on Nvidia GPUs. The proposed storage format provides coalesced memory access, manages the overhead for each element in the matrix and balances the load among the threads. Interleaved combination is used by the authors to provide coalesced memory access reduce the load overhead. Initially, all the *nnz* values are removed from the sparse matrix and are shifted to the left resembling the ELL format. Next, c number of rows are selected and combined by interleaved combination to form a new SHEC row. Each of these original c rows can be considered as to be stored in ELL format with values and column indices arrays. When these c original rows are combined together to form the SHEC row, an extra row pointer is required to point to the first non-zero in each new row. So it is identical to the CSR technique with a value, indices and a row_pointer array. If the matrix size is not divisible by c then the matrix is zero-padded to achieve this. The value of c will be a power of 2 but will not exceed 32. c cannot be more than 32 since a warp cannot handle more than 32 threads. If c is 1 then it becomes the CSR format and if c is 32 it becomes ELL format. The authors reorder and segment the matrix to reduce the overhead and to balance the load on the threads. An interleaved combination of variable *nnz* per row results in wastage of storage space and can create load imbalance. Hence to avoid this the row11s are sorted using the quick sort algorithm but this introduces but this might still result in load imbalance. To avoid this the sorted rows are segmented into a maximum of 6 segments. The first segment contains rows with size not less than 32, the next segment with *nnz* per row between 32 and 16, the next between 16 and 8 and the last segment contains *nnz* per row less than zero. The GPU kernel for the SpMV operation using this kernel is similar to the CSR format. One warp is assigned to the *c* rows or one SHEC row. A parallel reduction is required to write the result from the shared memory to the global memory.

New sparse matrix storage formats introduce the problem of converting the matrix from existing format to the new proposed format and generally this results in an overhead and in several cases it undermines the reported performance. Since a large number of the matrices are stored in CSR format Liu and Vinter [142] propose a new SpMV algorithm based on the CSR storage format using speculative segmented sums. In this technique the authors, initially, perform a segmented sum operation on the GPU to generate possibly correct results and later the partial sums are rearranged on the CPU to obtain the correct result vector. The authors consider three parameters to improve the performance of segmented sum: reduction of the overhead due to the global memory access by generating the segment descriptor during run-time, Processing of empty rows without any pre or post processing, and using heterogeneous processors (CPU+GPU) for the processing. The proposed algorithm mainly consists of two steps: the speculative segment execution stage on the GPU and the prediction checking stage on the CPU. In the first stage, the algorithm runs on GPU and performs speculative SpMV to produce conceivably incorrect results of the vector y. Incorrectness does not denote numerical incorrectness rather it signifies the error in the layout of elements in the result vector y. Row pointer, tile offset, and the boundary of the tiles are stored in separate arrays. An array named segment_descriptor is used to indicate whether the rows are completely empty or not, a synchronizer array is used to synchronize the elements, and a dirty counter array is used to signify the correctness of the generated results in the first phase. In the second phase, the prediction check phase, the dirty bit is verified. If the value is one then the results from the initial phase are assumed to be true else the position of the tiles are corrected in the result vector. Segmented sum techniques on GPUs have been also covered in [143, 112, 144].

A blocked row-column (BRC) storage format that utilizes two dimensional blocking for the storage of sparse matrices [145]. This technique reduces thread divergence and improves load balancing. Thread divergence is achieved by sorting and reordering the rows and combining them into various groups. Load balancing is accomplished by partitioning the rows into a number of blocks with each block containing a constant number of nnz's. The row sparse matrices are permutated like JDS and is combined with the ELL storage mechanism. All the rows are not padded as in the ELL format. Instead, the rows are grouped into blocks and then for each block we pad the rows. This reduces the amount of padding required. Moreover, the sparse matrix is again blocked along the column dimension. Therefore, the matrix is initially scanned row-wise and then column-wise to create blocks. This results in the scheme being adaptive to the structural characteristics of the matrix. CUDA implementation of BRC uses multiple threads per row and an atomicAdd() operation to sum the results.

Maggioni and Berger-Wolf [146] propose a storage scheme based on ELL called Adaptive ELL that distributes variable threads to the rows depending upon the computational load which results in balanced warps for SpMV in GPUs. They propose a novel heuristic for assigning workload to threads. Adaptive ELL adapts itself to the structure of the input matrix and is based on the ELL format. It is based on adaptive warp balancing that balances the warps. The proposed data structure is similar to the sliced-ELL. This scheme like Sliced ELL consists of slices that are stored in the ELL format. Therefore, it will have an array for storing value and a second array to store the column indices. In addition to that, it requires an offset array of size equal to the number of working warps, to store the starting location of each ELL substructure. Moreover, another array of the same size is required to store the keep track of each local workload. Another array of the same size is required to store the reduction map that will decide the threads that write to the result vector, which would result in coalesced memory access. Finally to keep track of the row offset we require another array of the same size. An atomic operation is performed to sum up the final result. The authors also discuss a heuristic for proper load assignment to the warps along with a loop unrolling heuristics.

Maggioni and Berger-Wolf [147] extend their work, Adaptive ELL, by adding compression of indices using delta encoding and warp granularity to increase the performance SpMV on GPUs. This modified scheme is named as Compressed Adaptive ELL

(CoAdELL). Differential encoding is used between contagious nzz values so that the column width can be represented with less number of bits. The major difference between AdEll and CoAdell is the compression of column indices. Instead of directly storing the column indices, delta indices are stored after differential encoding.

Choi et al. [148] implements the blocked version of CSR algorithm (BCSR) on GPU. Furthermore, they also propose a second sparse storage structure, which is the blocked version of ELL called the blocked ELLPACK (BELLPACK). A matrix of size $m \times n$ in CSR is converted into BCSR by partitioning the matrix into sub-blocks of size $(m/r) \times (n/c)$ sub-blocks of size $r \times c$. The nnz values are stored block wise contiguously in a float *data* array. The block column are stored in an integer *index* array and the starting index to each block is stored in the *bptr* array. For creating the BELLPACK format initially the matrix is converted into sub-blocks similar to the BCSR. Then the block rows are sorted depending upon the number of blocks per row. After which the matrix is subdivided into matrices and each of these matrices are stored in the ELLPACK format.

Ekambaram and Montagne [149] proposes a new sparse storage scheme inspired by JDS (JAD) format called Transposed Jagged Diagonal. A major advantage of this format is that the permutation array in JAD is not required. In this format, the initial sparse matrix is shifted up instead up of shifting left by removing the *nnz's*. Then each row elements are sorted internally inside the row in the descending order and it is shifted left. The *nnzs* in the resulting array is stored in a float array , *data*, row by row. Each of these rows is called the transposed jagged diagonal. Another integer array called *row_ptr* stores the indices to the start of each *nnz* row in the original matrix. The pointers to the starting of each transposed jagged diagonal are stored in an integer array called *start_pos*. Each transposed jagged diagonal is separated with the help of a semi-colon in the data array. There are four load operation and one write operation which is similiar to that of the JAD format. The amount of storage taken by TJAD format is given by $(N_{nnz} * 1) * 8 + N_{tjd} * 4$ bytes. Where, N_{nnz} gives the total number non-zeroes in the matrix and N_{tjd} is the number of transposed jagged diagonals.

Kreutzer [113] proposes a unified sparse matrix storage for all modern processors with wide SIMD units. They propose a storage format that would provide efficient performance in all processors such as Intel multicore processors, Intel MICs, and CUDA architectures. The proposed format is based on the sliced ELL combined with SIMD vectorization and is called the SELL-C-σ. The C (chunk) indicates the width of the SIMD registers in the x86 architectures and on CUDA architecture it is the number of threads per warp. σ is called the sorting scope that is the number of rows that will be sorted to reduce the overhead and increase the performance. The rows within a chunk are stored column wise and the chunks are stored one after another. The rows in chunks are padded to that of the largest row similar to Sliced ELL. To avoid this head selective sorting is applied using σ parameter. But this does affect the performance of the iterative solvers. The CUDA based implementation has not been discussed in detail. Only the results are shown. Whereas, the parallel implementation discussed is mainly that of OpenMP.

Oberhuber *et al.* [150] propose a new storage format based on CSR and Sliced ELLPACK called the Row grouped CSR (RgCSR). This method has been proposed to rectify the un-coalesced memory access in GPU's. The proposed technique stores the elements in each row in a coalesced manner. Initially, the matrix is divided into groups of a certain number of rows. For each group the nnz values are stored as follows:

1. First, the first element from the first row is stored followed by the first element in the second row and so on until the first element in the nth row. Here n is the number of rows in a group.

2. Then the second element of the first row is stored followed by the second element of the second row and so on until the second element of the nth row

3. The above process continues until we store the nth element in the nth group

If the number of elements in the rows of a group is not equal, they are padded with zeroes. Therefore, the *data* array that stores the *nnzs* will contain extra padded zeroes. Similarly, the *index* array that stores the column indices of the *nnzs* will also have extra zeroes corresponding to the padded zeroes. The other major differences from the CSR is the storage of the length of each row in a new integer array, *rowLength* and a *groupPointe*r array that points to the first indices of the group instead of the *row_ptr* in the CSR format. This technique reduces the thread divergence inside a CUDA kernel due to the use of row length array. Each of the group is assigned to a CUDA block and each row is accessed by a thread. It doesn't perform well for all matrices due to the complicated structure of the *nnzs* in the sparse matrices.

Yang *et al.* [151] propose an improvement to the CSR format called Improved CSR aimed at SpMV on CUDA-based GPUs. The authors solve the issue of serialization, in the CSR storage, caused by uneven nnz elements in a row by padding the rows with zeroes which result in aligned access to the global memory. However, the discussed technique increases the space complexity and the cause the threads iterate over extra elements that degrade the performance.

As seen before one of the major issues with the ELLPACK-R format is the presence of non-variable nnzs in a row which reduces the performance of SpMV on CUDA based GPUs. Whereas, in the JAD format, the disadvantage is the non-coalesced memory access. Cao et al. [152] proposes a new hybrid model which is a combination of ELLPACK-R and JAD named ELLPACK-RP. The storage format can be constructed from ELLPACK-R format by permuting the rows in descending order. A permutation of the row length array of ELLPACK-R is also required. A separate array is used for the storage of the permutations. The implementation of the authors uses a thread per two rows.

Heller and Oberhuber [153] extend the work on RgCSR [150] to develop a new sparse storage format called the Adaptive RgCSR. The main disadvantage of the RgCSR was that the threads per row were fixed (It was using a single thread per row) which resulted in the serialization of the computation due to variable elements in rows. Like the RgCSR, the rows are divided into groups and each group has the following information associated with it: the first row of the segment, the total number of rows, the first element in the group and the size of the group, which indicates the number of elements in the group. In this technique, each group is assigned a CUDA block and four threads (Multiple threads) are assigned to each row. The partial sum is calculated and the final sum is obtained with the help of reduction.

Abu-Sufah and Abdel Karim [154] introduce an improvement to the Transpose Jagged Diagonal Storage (TJDS) by the usage of blocking technique, which is called Blocked Transpose Jagged Diagonal Storage (BTJDS). This storage scheme is very similar to the TJDS except that the matrix is divided into blocks, and each block is stored in the TJDS format. The arrays required for the storage of this scheme similar to that of

TJDS except an integer array that stores the starting index to each block. The authors assign a fixed number of warps to a single block. Each column in a matrix is assigned a thread, and they iterate until all columns are traversed. Therefore, one thread computes the partial sum of a row from the actual matrix. These partial sums are combined in the global memory. The variable number of nnzs affect the performance of this storage format in SpMV because it causes a load imbalance among the warps. This technique does not achieve a good memory bandwidth utilization as compared to that of other techniques such as HYB and ELL.

Kreutzer et al. [155] propose a sparse matrix storage scheme to reduce the space overhead of extra padded zeroes in the ELL format and improve the performance of SpMV on GPUs. The suggested format is similar to ELLPACK and JDS. This format is quite similar to the already proposed sliced ELLAPACK. In this format, the sparse matrix is shift left after removing the nnzs as in ELL format. Moreover, then the rows are sorted based on the descending order of nnzs per row. The sorted rows are then sorted into blocks, and each of these blocks is assigned a CUDA block so that the load is balanced among the warps. The major disadvantage of this technique is that the multiplication is performed by permutation, which destroys the dense blocks and diagonal elements. Therefore, cache reuse is prevented which degrades the performance. The authors also test this format by performing SpMV on a cluster of GPUs using Asynchronous communication.

3.5 Iterative Solvers on GPUs

An image reconstruction process can be reduced into a set of sparse linear equations. Florian and Koch [156] has solved equations thus generated using Krylov based Conjugate gradient (CG) algorithm on a CUDA based GPU. By default, they use the CSR storage format to store the sparse matrices. They form a many experiments by varying the storage structure of sparse matrix A to other existing formats such as HYB and ELL. Moreover, they use the shared memory of the GPU to have a coalesced access to the memory. They also try a variation of the algorithm that uses back projection but reached a conclusion that it is hard to implement on the current GPU architecture. They find that a configuration that consists of a good sparse structure in addition to the use of the shared local memory of the GPU produces the best result. They also try to run them on FPGAs and Cell broadband engines but reject them due to unpromising results.

Khodja et al. [157] implement the generalized minimal residual iterative method (GMRES) on a cluster of GPUs utilizing communication reduction techniques for solving large-scale sparse linear systems and compare it to a corresponding CPU- cluster implementation. One of the major issue with implementations on GPU clusters is the data communication between the GPUs. It would consist of transferring information from one GPU to the CPU, sending of MPI communication between the CPU and the GPUs, and transferring the data to the second GPU. To reduce this communication issue, the authors propose a new compressed format for sparse vectors. The authors note that there is no need to transmit the whole vector x to all nodes, and only the required parts of the vector are needed to be passed, and they propose the compressing technique based on this objective. The authors use the hypergraph partitioning on the sparse matrix to partition the matrix to run on the various GPU nodes. Hypergraph partitioning is an NP-complete problem, but various heuristic techniques exist such as Zoltan. The authors conclude that the performance on the GPU cluster is much better than the GPU cluster.

Chou and Liao [158] proposes a new hybrid algorithm to solve Markov chains, and the proposed algorithm is a combination of Jacobi and Gauss-Seidel iterative methods. The authors use CUDA and MPI to implement it on a GPU cluster and a CPU cluster. They solve for a Markov chain with 1000 states. The proposed algorithm is discussed below:

The vector x can be portioned into subsets that are meaningful. Correspondingly, we can partition the matrix Q^T and we can solve the equations using these partitions. We can divide e the matrix Q^T into partitions of size K^2 where each partition has the size $(n/k)*(n/k)$ and the vector x can be subdivided into k sub-vectors each of sizen/k. We know that the Jacobi can be represented by the equation as given in

$$x_i^{(k+1)} = H_i \times x_i^{(k)} \tag{3.3}$$

where $H = D^{-1}(L + U)$ as discussed before in previous sections. Equation 3.5 is inherently parallel and be solved by a processor or a GPU block.

Moreover, each GPU block or a processor can solve then/k elements of $x_i^{(k+1)}$ using gauss seidel as follows:

$$x_{i,j}^{(k+1)} = H_{i,1} \times x_{i,1}^{(k+1)} + ... + H_{i,j-1} \times x_{i,j-1}^{(k+1)} + H_{i,j+1} \times x_{i,j-1}^{(k)} + ... + H_{i,n/k} \times x_{i,n/k}^{(k)} \quad (3.4)$$

$X_{i,j}^{(n+1)}$represents the j-th element of i-th solution vector that is obtained at the (k+1)-th iteration by the i-th processor or GPU block. The authors does not do any analysis on the speed of the convergence.

High accuracy surface modeling (HASM) has been developed to assess the anthropic impact on the environment and for catchment hydrologic modeling. HASM consists of a large number of sparse linear equation systems that needs to be solved. Yan et al. [159] propose the parallelization of the preconditioned conjugate gradient based on the Krylov subspace for solving linear equations related to HASM on GPU based on CUDA.

More techniques based on GPU based conjugate gradient (CG) and Bi-conjugate algorithm can be found in [160, 161, 162, 163]. Solution of GMRES can be found in [164, 165, 157, 166, 116, 167]. A GPU solution of Jacobi and Gauss-Seidel can be found at [168]. Iterative methods that use GPU clusters can be found in [157, 169, 170].

Bosnacki et al. [171] propose a modification of the Jacobi algorithm for calculating the steady state probability vector for Markov chains using backward segmented scan. Backward segmented scan computes the sum of all the indicated elements in the array in parallel. Once the products of the matrices are calculated the row sums are computed, using backward segmented scan in which all the elements in a segment are added, and later written to the start of each segment. The segmented scan algorithm is provided in the CUDA Data Parallel Primitives Library (CUDPP) [172]. The technical implementation details of the segmented scan on GPUs can be found further in [143].

Ahamed and Magoules [173] conducts analysis on the performance of various iterative methods, mainly the Krylov methods, using various well-known data formats and data structures for sparse matrices on GPUs. They develop a library that provides Krylov iterative solutions, named Alinea which would be an enhancement on existing libraries such as Cusp [64] and cuSPARE [174]. The tested Krylov methods are Stabilized BiConjugate Gradient (BiCGStab), Stabilized BiConjugate Gradient L (BiCGStabl), Generalized Conjugate Residual (P-GCR), BiConjugate Gradient Conjugate Residual (P-BiCGCR), and Transpose-free Quasi-Minimal Residual (P-tfQMR) for the unstructured matrices. Conjugate Gradient (CG) is used to find solutions for equations with symmetric matrices. CSR, COO, ELL, and HYB are the storage matrices that are utilized by the authors to test these iterative solutions and compare with the implementations in both the Cusp and CUSPARSE library.

Cormie-Bowins [175] implements a parallel version of both Jacobi iterative method and biconjugate gradient stabilized (BiCGStab) [176] using Nvidia GPUs for computing the reachability probability in Markov chains. The parallel implementation of Jacobi is quite straight forward with a single GPU kernel for an iteration, but the BiCGStab implementation consists seven different kernels as there are multiple matrix-vector multiplications. Implemented techniques are used in solving probability matrices randomly as well as generated with a probabilistic model generator. The general conclusion is that for sparse matrices, the higher the sparsity, the better the performance of Jacobi method. For, small sparse matrices, the serial BiCGStab produced a better performance than parallel implementation.

Wang et al. [167] solve a system of linear equations using Parallelized GMRES on a Tesla GPU. The major contribution of Wang et al. is the parallelization of the ILU (Incomplete LU) preconditioner. They parallelize a block version of ILU called as block-ILU [44]. The matrix A is subdivided into blocks, and ILU is applied to each block separately. The rest of the GMRES is parallelized and run on GPU. Implementation details about parallel GMRES are by parallelization of matrix-vector multiplication as discussed by Garland in [177]. Block-ILU increases the time for convergence as compared to normal ILU. The above technique was tested on a Tesla T10P GPU using oil field data provided by ConcoPhilips. The largest sample to be run was of size 1,122,000 and took 22 seconds to run it as compared to the serial version that took 436 seconds which implies a speedup of approximately 20x.

3.6 Challenges in GPU Implementation of Sparse Solvers

Iterative solvers mainly consist of multiple numbers of sparse matrix-vector computations. Implementation of SpMV on GPU has many issues, which are discussed in this section. Specialized storage structures are used to improve the performance of sparse SpMV. These structures have design issues for translating it to GPUs. In this section, we shall see the problems faced in implementing parallel sparse iterative solvers and translating sparse structures to GPU.

1. Coalesced Memory Access to both the sparse matrix A and the vector y. Consecutive threads should access the consecutive memory location. The access to scattered locations results in memory divergence, which results in a single memory access per thread. The global memory consists of sequences of 128-byte segments. At a time, 16 threads (half -warp) access the memory. The number of memory transaction for a half-warp depends upon the number of memory segments touched by the address of the half-warp [67]. If the half-warp touches a single 128-byte segment, we call the memory access coalesced. Therefore, the threads with adjacent indices should access contiguous memory words. Non-contiguous access increases the memory transaction, therefore reducing the bandwidth and hence the performance.

2. Load balance among array threads and warps. The threads and warps should have the same load. In instances where a single thread is used for calculating the product of a row with variable nnzs, this results in the warp having threads that would finish executing at different instances of time. Hence, all the threads in the warp will take time equal to the longest row. Hence, load balancing among the threads and warps are necessary to increase the performance of SpMV.

3. Avoiding thread divergence. Even though each thread in a warp is allowed to have its execution path, this reduces the performance of SpMV. The threads of a warp should follow the same execution path for a majority of computation.

4. The Performance of SpMV and storage schemes for sparse matrices based on the number of nnzs per row. There is a variation of nnzs per row among the sparse matrices. It is observed from the literature that the most of the scheme

do not cater to both these natures of matrices. Some matrix storage format and associated SpMV produces an excellent performance for structured matrices while the others produce the same for unstructured matrices. The variation of sparsity in the matrices determines the suitability of sparse storages and SpMV schemes.

5. Pre-processing Involved to store a sparse matrix in a specific scheme and Conversion cost: A large number of sparse storage formats involve pre-processing. Care should be taken that this pre-processing should not be expensive. Expensive pre-processing negates the speedup achieved during the SpMV operations. In some scenarios, the preprocessing takes a larger amount of time than SpMV. Such pre-processing has to be avoided. The cost of converting from a standard sparse storage to the new sparse should also not be expensive.

6. Byte/ float ratio: The GPU based sparse iterative solutions are memory bound than computationally bound. Some bytes that not to be fetched for a single floating point operation are higher and is represented by Byte/float ratio. There are two ways to increase the performance of the sparse SpMV. The first is to reduce the computations or time for computations. The second technique is to reduce the memory bandwidth which would mean to lessen the amount of bytes required or should be transferred to perform sparse SpMV.

7. Irregular Algorithms or algorithms that use structures such as lists, trees, linked lists does not have a good SpMV that has high throughput and less memory bandwidth.

Additional complications arise because the sparsity pattern of the matrix, i.e., the position of the non-zero entries, can have considerable impact on spMVM performance due to indirect access to the right hand side (RHS) vector; this makes it difficult to understand or even predict performance via simplistic bandwidth-based modeling. And finally, the sparse matrix storage format has a considerable performance impact and the optimal choice is known to be very sensitive to the underlying hardware.

The threads in a warp need to access contiguous locations of the global memory to avoid memory divergence. The access to non-contiguous memory locations by the threads in a warp requires on memory access per thread, whereas if the memory locations accessed by the threads are sufficiently close enough, the memory access can be coalesced to improve the efficiency of the memory access. Theoretically, the global memory consists of segments of 128 bytes sequentially arranged [67]. A half-warp (16 threads) is allowed to access the main memory at a time. The number of memory segments touched by the addresses used by the threads provides the number of memory transactions for the half-warp. If the half-warp touches exactly one segment, then the memory is said to coalesce fully, and if any of the threads touch other segments then we call it uncoalesced access. Uncoalesced memory access decreases the bandwidth efficiency and is reduces the performance of memory-bound kernels [67].

3.7 Gaps in Literature.

TJDS is only suitable for matrices that have power law characteristics. The technique requires column-based ordering. The pre-processing phase involved would be expensive. The technique does not always guarantee the blocks to have the same nnzs per block, this would result in improper load balancing. Moreover, this storage scheme and associated SpMV is not a suitable general technique.

ESB (ELLPACK Sparse Block) performs a lot of redundant computations. The technique has overheads in the form of sorting within each column partition, memory overhead and a load of permuting the row over column partition.

Block ELLPACK and BCSR techniques are not suitable for matrices that follow power-law due to higher memory overhead. It gives a good performance for matrices that has a dense block substructure. However, the auto-tuned BELLPACK and BCSR performs poorly for unstructured matrices. BCRS and Bellpack consist of additional padded zeroes per sub-block that do not contain adequate nnzs. The pre-processing involved in creating the proposed storage schemes are complex. BCSR requires more access to the global memory that decreases the performance.

ICRS has poor load balancing among the threads and the warps as the rows are treated non-preferentially for matrices with variable nnzs. The performance of this scheme for matrices with variable row distribution is poor. The performance of the scheme is reduced as compared to CSR-vector for variable rows. Moreover, the execution unit is underutilized. HYB-R Moving larger amount of data into the ELLPACK kernels results in the technique to have the same deficiencies as that of ELLPACK technique.

The only advantage is that the overhead caused by the COO part is reduced. Another major disadvantage is the usage of multiple kernels that would result in higher overhead. The scheme does not reduce redundant data transfers and extra padded zeroes. The technique also requires costly pre-processing. Moreover, the conversion from standard formats to the proposed technique is expensive.

The BRO-formats reduces the memory bandwidth by lossless compression. Reduces the bytes that are transferred for the access of the matrices. Higher compressible matrices produce better performance. For a good performance the matrices should be highly compressible, large coverage factor, and a large number of rows in the matrix. Higher variance in the row distribution results in a poor performance by the scheme. The technique focuses on improving the performance by reducing the memory bandwidth. Hence it does not provide warp granularity or coalesced memory access.

Sliced COO contains pre-sorting which would increase the overhead. Assigning one thread block per slice results in thread divergence. For larger matrices the technique uses up more memory. The technique does not guarantee a coalesced memory access.

BiELL and BiJAD are designed for sparse matrices with no specific structure. Good load balancing among the warps and the threads. BiJAD reduces the padded zeroes as compared to BiELL but the memory access pattern in BiJAD does not coalesce. Both the scheme produces a good performance for matrices with a large number of nnzs and when the standard deviation is high. However, the generator matrices of Markov chains have a small number of nnzs with a large number of rows. The performance of the technique deteriorates when the standard deviation decreases. Moreover, the technique requires higher pre-processing. BiJAD also performs poorly for SpMV with structured matrices.

Sliced Ellapck doesn't have good load balancing. Requires extra padding. Will result in thread divergence.

BRC reduces the thread divergence by reordering and grouping the rows of the input matrices with the almost same amount of nnzs to the same warp. A warp will then execute an nnz with the same nnzs per row. BRC lacks generality and performs well for certain matrices with specific characteristics. The technique provides poor performance for matrices that are highly sparse (average nnz=4), where most rows have nnz=2 and nnz=3. Some of the matrices that show poor performance are EPI, ECO, CIR, and

Web. BRC can be used as a substitute for COO but not fro ELL. If the original matrix doesn't have a block sub-structure it is a time-consuming process to convert to the new format.

ACSR is suitable for graph mining applications. Does not contain high pre-processing since preprocessing is not suitable for graph mining applications. Not much suited for general matrices. The technique is slower than HYB when using matrices such as Wik and EUZ. The ACSR storage format is slower when the average nnz per row and the standard deviation is lower. All the matrices do not use the dynamic parallelism which results in poor performance. Irregular matrices with a small number of long rows also show bad performance.

BCCOO and BCCOO+ techniques are only good for matrices that show dense sub-block structure. The overhead in generating the format from the standard format is higher. The above technique with the associated SpMV only works on the Nvidia GPUs using CUDA. It doesn't work on AMD GPUs because the non-portable communication causes a deadlock in AMD GPUs. All the block based sparse storage format such as BCOOCO, BCOOCOO+, BRC, and CSB relies on the sparsity structure. Therefore, they are highly application dependent.

CMSR is dependent upon the hardware for the performance. Due to the limited amount of shared memory per multiprocessor, it is not feasible for all the matrices. For better performance, the matrix has to be large enough. This technique provides poor performance when the matrices have empty rows.

SIC technique requires the reordering of rows and segmentation to reduce the imbalance among the warps. The technique does not perform well when the interleaved rows are greater than eight which is due to insufficient parallelism as the rows of the SIC format only occupies 125 warps. The matrix webbase does not work well this format due to an uncoalesced memory access to the vector x and due to less locality in reference to the vector which results in higher level of the cache miss. Speculative segmented sum All the matrices yet still do not perform well. In some cases, HYB has a better performance than the speculative segmented sum.

This technique is used to store Markov chain generator matrices. However, the technique is inefficient due to non-coalesced memory access and the time is taken to decompress the different rows take different time which results in serialization. The number of idle threads due to this is high. Hence, the technique does not provide a good load balancing nor a solution for thread divergence.

SHEC has a pre-processing that takes time, and the high amount of padding is required. There is a reduction in performance due to cache miss to vector y that reduces the performance. Moreover, the access to the vector y is non-coalesced.

When the total number of iterations are less (When the rows are few) the CSR-5 does not provide a good performance as compared to techniques based on CSR.

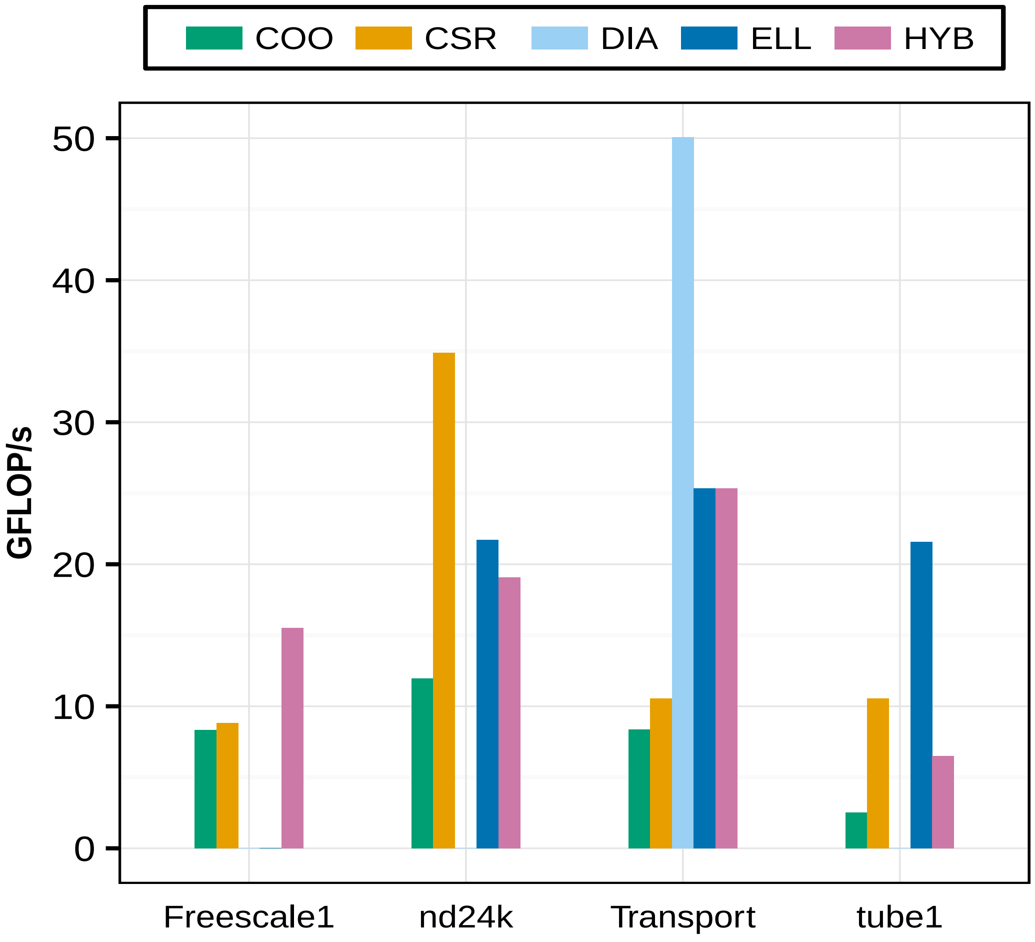
Chapter 4

Proposed Technique

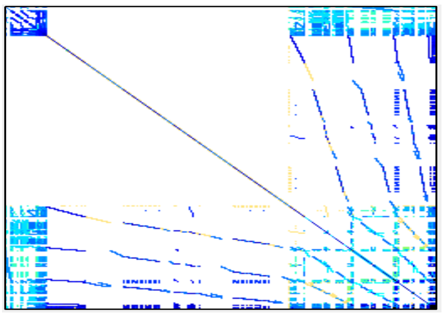
Memory bound algorithms such as Sparse Iterative Solvers exhibit superior performance on GPUs due to the high bandwidth memory hierarchy in cache based GPUs [35,36]. PageRank algorithm, HITS, and Random walk with a restart on GPUs have reported a performance improvement of 18 to 32 times. [42] show that the GPU-accelerated LOBPCG based on SpMM kernel is 3 to 5 times faster than multicore CPUs with the same power draw, which further indicates that the energy efficiency of executing the SpMV kernel on GPU is more than that of CPU [42, 43].

The sparsity structure of the sparse matrices varies from matrix to matrix, and application domain to domain. Figure. 4.1(b), (c), (d), (e) shows the structure of four sparse matrices from the University of Florida Sparse Matrix collection. Figure. 4.1(a) illustrates the performance of SpMV kernel under five different sparse storage formats, i.e. COO, CSR,ELL.HYB, DIA, for these four matrices. Figure. 4.1 indicates that the performance of the SpMV kernel is dependent upon the sparsity structure of the matrices. We can deduce that none of the sparse representation schemes are perpetually superior rather the SpMV performance depends upon the sparse representation and the choice of sparse representation is based on the sparsity structure. The GPU architecture used for SpMV computations and the thread-block-warp configuration of the SpMV kernel are two other factors that affect the performance of SpMV kernel on GPUs.

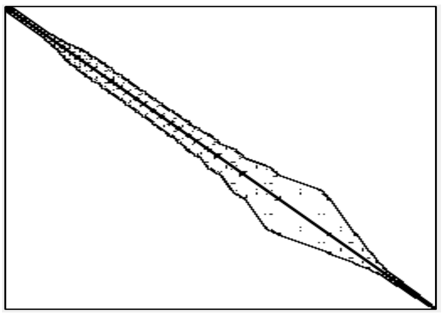
Hence we propose an Adaptive Deep Learning model based iterative solver that gives the best performance by adapting the best storage scheme possibble for a given sparse matrix. Deep Networks are much more attractive when the problem has non-linear nature as compared to shallow networks. A shallow network should have more number



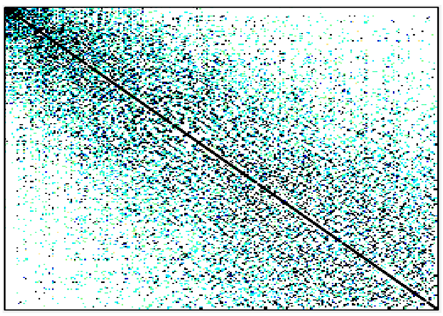
(a)



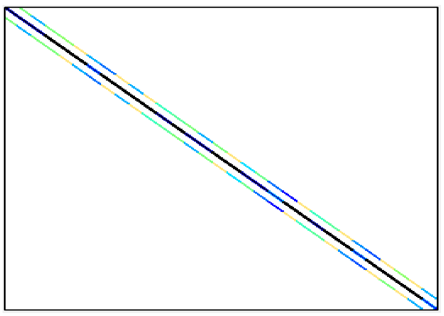
(b) Freescale1



(c) tube1



(d) nd24k



(e) Transport

Figure 4.1: Sparsity Structure of some sparse matrices

of connections to get the same level of performance as Deep networks . More number of connections imply higher number of floating point computations that increases the cost of operations as compared to similar Deep Networks. The functions that can be compactly represented by an architecture of depth k requires an exponential number of computational elements to be represented by a depth k − 1 architecture.

Practically Kernel methods are expensive as the number of support vectors grow the training set also increase and hence for larger datasets it takes large amount of time. Whereas in Deep Neural networks the time is independent of the training set, rather it depends on the number of connections between the neurons.

Hence, we propose a dynamic sparse representation framework for linear solvers that adapts the sparse storage representation based on the sparsity structure of the matrix. The framework relies on creating a model using Deep-Learning that maximizes the performance of sparse Jacobi iterative solver by using the best sparse representation.

We have designed a deep architecture that has six layers, which consists of an input layer, four hidden layers, and output layer. The input layer has 14 neurons corresponding to the number of input features. The second layer has 100 neurons, the third layer has 50 neurons, the fourth layer has 25 neurons, the fifth layer consists of 10 neurons and the last layer or the output layer has five neurons (corresponding to the number of classification classes). All four layers use ReLU as the activation function. The cost function of the deep network is minimized using Adams optimizer [178]. These configurations provide us with best possible accuracy for the deep network. We tried out various network configurations to achieve the best performance in terms of accuracy of the model. The selected parameters provided the best results in our experiments. The major advantage of the deep network over shallow techniques such as SVM and decision trees is that the deeper layers of the network automatically find new features which cannot be obtained using the shallow networks and this leads to higher accuracy in classifying the data. Another main benefit of a neural network is that it requires very less pre-processing of the input features as the features are learned in the network. The shallow techniques such as are excellent at memorization, but not so good at generalization. So, if we train the network with every possible input value, a shallow network shall memorize the input-output pairs without regarding the entire scenario. A large number of layers enable feature learning at distinct levels of abstractions. The ability to learn intermediate features between a given input and the final high-level classification makes Deep learning a better alternative for generalization.

4.0.1 Dataset Formation

We require a dataset for training our predictive model. Hence, we created a dataset utilizing the University of Florida Sparse Matrix repository [179]. We analyzed a large number of matrices from various application domains and selected matrices spread across the domains. We applied the following criteria while selecting the matrices for the dataset.

1. The matrices are not complex matrices, i.e. They are Real matrices.
2. The matrix can be square or rectangular.
3. The matrix can be symmetric or non-symmetric.
4. The sparse matrix fits in the global memory of GPU, such that the total occupied space by the sparse matrix is less than or equal to 80% of the total memory. The size of the sparse matrix in this regard is calculated as $16 \times nnz$ bytes, which is the size of the matrix in COO format. The global memory of the GPU used has been provided in Table. 4.4.
5. The total number of rows in the sparse matrix should assure minimum warp-level concurrency, which implies that the minimum number of rows in the matrix should be equal to warp-level concurrency. Warp concurrency is the ratio of the total threads across all the multiprocessors to warp size, i.e.$warp\,concurrency = (Threads\,per\,SM * SM)/warpsize$. Table. Y provides the hardware configuration of the GPU used in our experiments.

To determine the class/label of the dataset, we ran SpMV kernels using all the five storage formats (COO, CSR, DIA, ELL, and HYB) for each matrix. The storage format that recorded the best performance in terms of GFLOP/s was selected as the best storage format for a given matrix.

We analyze the observed performance of each sparse representation for all the matrices in our dataset. The performance of the SpMV kernel was measured using GFLOP/s.

Table 4.1: Distribution of class on dataset for SpMV kernel on Tesla K20c

COO	CSR	DIA	ELL	HYB
1%	57%	6.25%	22.25%	13.5%

All reported performance measurements only include the kernel execution time and do not include the data transfer time between the CPU and GPU.

Table. 4.1 provides the distribution of classes in our dataset. It shows the percentage of matrices that performed the best for a given sparse matrix representation out of 1056 matrices in our dataset.

We can observe that the CSR format has a higher share as compared to other storage formats. The COO format performs the worst, and only 1% of matrices recorded the best timings with COO. The ELL and HYB performed moderately well. The distribution of the classes on the dataset does not provide us with the complete picture, rather the performance difference between the each sparse format provides more insight into the performance aspects of these storage formats. Table. 5.3 provides the average slowdown of the sparse representations while using a single sparse storage format as compared to the utilization of the best possible representation of the sparse matrices for all 1035 matrices in our dataset. For example 1.5x for CSR implies a loss of 33% when using the CSR format for representing the dataset entirely as opposed to using the best representation for each matrix in the dataset. We can observe that the csr performs the best, but ELL and HYB are quite close behind. There is only a difference of 5% between ELL and CSR. The difference between ELL and HYB is quite narrow. DIA has higher loss of performance as compared to COO due to its inability to store all matrices. The inability of a sparse representation scheme to store a matrix was recorded as 100% loss.

4.0.2 Feature Selection

In the Introduction to this chapter we discussed how sparsity structure affects the performance of SpMV kernels. Hence, we need to gather relevant information about the sparsity structure of the given matrix. Therefore, we compute various features of the matrices. After many experiments and analysis, we selected the significant features that perceive the structure of the given input matrix efficiently. Table. 4.2 shows the

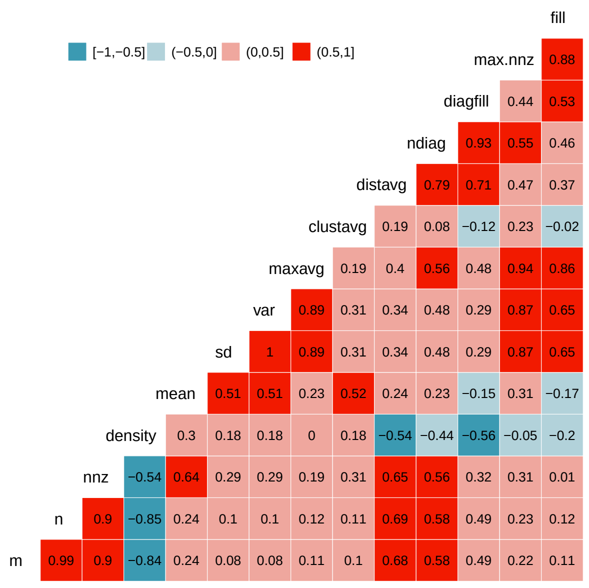


Figure 4.2: The correlation matrix for our selected features

final feature set that was chosen.

All of our features are simple to calculate.m, n, nnz, $density$, $mean$, and $variance$ all have a time complexity of $O(1)$. m, n, and nnz are usually stored in the input sparse matrix file. The mean non-zeroes per row ($mean$) is easily calculated from m, n, and nnz. Standard Deviation (sd) has a complexity of $O(2n)$. Variance can be calculated with a complexity of $O(1)$ once we have $mean$ and sd. For finding the maximum number non-zero elements in a row ($max - nnz$), we need to traverse each row once. The average distance between the first and the last non-zero value per row ($distavg$) can be calculated at the same time as $max - nnz$. $clusteravg$ calculates the average number of consecutive non-zero elements per row. $ndiag$ is the number of diagonals in the matrix and $diagfill$ is the product of the number of diagonals to the size of diagonals. $ndiag$ and $diagfill$ was added to improve the conversion rate of nonconvertible matrices to DIA sparse storage. The feature $fill$ is used to determine whether a matrix can be feasibly stored using the ELL storage format.

4.0.3 Feature Distribution Analysis

Figure. 4.2 shows the correlation matrix for the selected feature set that was indicated in the Table. 4.2. The lower triangle of the matrix illustrates the feature distribution of

Table 4.2: Features Extracted From Sparse Matrices

Feature	Description	Time complexity
m	The number of rows	$O(1)$
n	The number of columns	$O(1)$
nnz	The total number of non-zero elements	$O(1)$
$density$	The density is defined as $nnz/(m * n)$	$O(1)$
$mean$	Mean non-zero elements per row	$O(1)$
sd	Standard deviation of non-zero elements per row	$O(2m)$
var	Variance of non-zero elements per row	$O(1)$
$max - nnz$	The maximum number of non-zero elements in a row	$O(m)$
$maxavg$	The difference between the maximum number of non-zero elements per row and average non-zero elements per row	$O(m)$
$distavg$	The mean of the distance between first and last non-zero elements in each row	$O(m)$
$clusteravg$	The mean of the number of distinct consecutive non-zero elements per row	$O(nnz)$
$ndiag$	The number of diagonals in a matrix with non-zero elements	$O(nnz)$
$diagfill$	The diagonal fill-in ratio. It is calculated as $(ndiag \times m)/nnz$	$O(1)$
$fill$	The fill-in ratio for matrices calculated as $(m \times max - nnz)/nnz$.	$O(1)$

Table 4.3: Summary of our feature set

	Min	Max	Average
m	838	8.34M	138k
n	292	8.34M	142k
nnz	1074	229M	194k
density	9e-07	0.12	3.9e-03
mean	0.59	424	27.13
sd	0.00	3594	40.3
var	0	12M	29669
max-avg	0	2.3M	9067
clusters	0	126.64	6.585
distavg	0	4M	34724
ndiags	1	10M	79643
fill-in	1	259k	972
max_nnz	1	2M	9094
diagfill	1	978k	9236

a given feature with every other feature. We can observe that the correlation between the features is quite low. There are only four features that have a correlation above 0.5 with respect to nnz. These are n, m, distavg, and mean. These correlations are due to the linear nature of this features, i.e. when m and n increases nnz also increase. n and m show a higher correlation as we have a large number of square matrices in our dataset. Moreover, if we observe the correlation between density and other features we can observe that these are highly uncorrelated which implies that each feature contributes distinct information. The distavg has a higher correlation with the other features, and we observed we could achieve a higher performance even without distavg, but the inclusion of distavg increases the performance of the classifier. We can see that the rest of the features are clearly uncorrelated or correlated by minuscule degree.

Table. 4.3 gives a quantitative summary of the features of the matrices in our dataset.

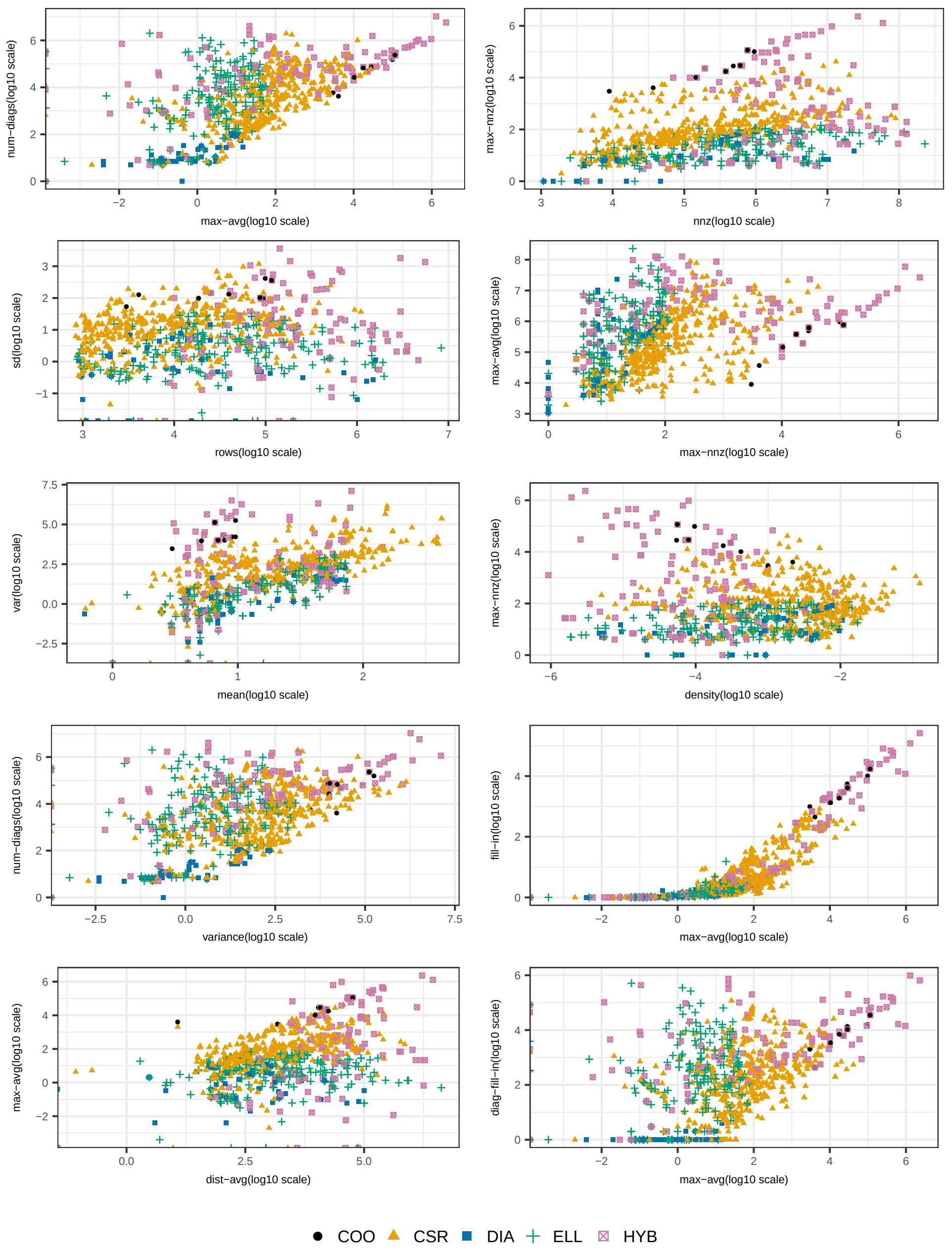


Figure 4.3: Projection of points in various 2-D planes

4.0.4 SpMV Performance Analysis on feature set

We use the selected sparsity features to observe and explain the performance of SpMV kernels on GPU. Due to the curse of dimensionality [Domingos] it is hard to depict, analyze, and organize high dimensional data (14 dimensions in our case). However, we show selected 2-D planes in Figure. 4.3 where various features from our feature set are projected. The projected points are coloured according to the sparse representations that reported the best performance (GFLOP/s).

In the max-avg vs ndiags plane, we can observe that the DIA class dominates the lower left corner whereas the CSR class is in the middle. This clearly indicates that higher number of diagonal does not mean DIA would perform best.

Algorithm 1 The Training Phase of Deep Networks

```
Input: $nfeat$
Output: $\{W_1....W_L\} \in R^2,\ \{b_1...,b_L \in R\}$
 1: function TRAINING(nfeat)
 2: Initialisation : $W \in [-1,1], b \leftarrow 1, a^{(1)} \leftarrow nfeat$
 3:     while $cost > minimumcost$ do
 4:         for $l = 1$ to Total number of Layers $L$ do
 5:             $a^{(l+1)} \leftarrow ReLU(W^{(l)}a^l + b^{(l)})$
 6:         end for
 7:         For the output layer $l$ find the derivative of the Loss function as
 8:         $\delta^{(n_l)} \leftarrow -(y - a^{(n_l)}) \bullet ReLU'(z^{(n_l)})$        ▷ ReLU applied Elementwise and $\bullet$
    indicates elementwise Hadamard product
 9:         for $l = L - 1$ to Layer 2 do        ▷ Compute hidden layer Cost derivative
10:             $\delta^{(l)} \leftarrow W^{(l)}\delta^{(l+1)} \bullet ReLU'(z^{(l)})$
11:         end for
12:         Compute the partial derivative for each layer l as:
13:         $\nabla_{W^{(l)}} \leftarrow \delta^{(l+1)}(a^{(l)})^T$
14:         For each Layer $l$ Update the weights using ADAM optimization
15:         Compute the new Cost
16:     end while
17: end function
```

In the max-nnz vs. nnz plane, the HYB format dominates the areas with both higher nnz and max-nnz. The ELL and DIA format cannot be seen after a threshold max-nnz. Due to higher values of max-nnz, matrices fail to convert to both DIA and ELL from the COO format in which the matrices are stored. Whereas, the CSR class performs best when the max-nnz is neither too low or nor too high. ELL, CSR, DIA classes performs well for all nnz values too whereas most of HYB class is at higher nnz.

Algorithm 2 Adaptive Deep Learning Jacobi

```
Input: $A$, $b$
Output: $x$
1: function Main(A,b)
2:     $feat \leftarrow ExtractFeatures(A)$
3:     $nfeat \leftarrow log_{10}(feat)$
4:     if Model not trained then
5:         $W, b \leftarrow Training(nfeat)$
6:     end if
7:     $sparseFormat \leftarrow DeepModel(nfeat)$
8:     $x \leftarrow Jacobi(A, b, sparseFormat)$
9: end function
```

Algorithm 3 The Deep Network predictive model

```
Input: $nfeat$, $W$, $b$
Output: $sparseFormat$
1: function DeepModel(nfeat)
2: Initialisation : $a^{(1)} \leftarrow nfeat$
3:     for $l = 1$ to Total number of Layers $L$ do
4:         $a^{(l+1)} \leftarrow ReLU(W^{(l)}a^{l} + b^{(l)})$
5:     end for
6:     return $a^{(L)}$
7: end function
```

The CSR class dominates the area of moderate sd and a low number of rows. The HYB class has a better performance when the number of rows is larger and at higher standard deviation than CSR. The ELL class provides the best performance for all row sizes but only at lower sd of non-zero values.

In the max-avg vs. max-nnz plane, we can see a clear distinction between CSR, HYB, and DIA. The CSR exhibits the best performance when the max-avg and max_nnz neither too low nor too high.

Algorithm 4 Dynamic Sparse format Jacobi Solver

Input: A, b, x, $x1$, $sparseFormat$

Output: x

```
1: function Jacobi($A, b, x, x1, sparseFormat$)
2: $Initialisation : error \leftarrow 1.0, \epsilon \leftarrow 10^{-6}, k \leftarrow 0$
3: $Initialisation : x \leftarrow (1/size(A)), x1 \leftarrow 0$
4:     switch $sparseFormat$ do
5:         case 0:
6:             Initialize and move $A$, $b$, and $x$ to $gpu$
7:             $JacobiCoo <<< gs, bs >>> (A, b, x, x_1)$
8:         case 1
9:             Convert $A$ to CSR
10:            Initialize and move $A$, $b$, and $x$ to $gpu$
11:            $JacobiCsr <<< gs, bs >>> (A, b, x, x_1)$
12:        case 2
13:            Convert $A$ to HYB
14:            Initialize and move $A$, $b$, and $x$ to $gpu$
15:            $JacobiHyb <<< gs, bs >>> (A, b, x, x_1)$
16:        case 3
17:            Convert $A$ to ELL
18:            Initialize and move $A$, $b$, and $x$ to $gpu$
19:            $JacobiEll <<< gs, bs >>> (A, b, x, x_1)$
20:        case 4
21:            Convert $A$ to DIA
22:            Initialize and move $A$, $b$, and $x$ to $gpu$
23:            $JacobiDia <<< gs, bs >>> (A, b, x, x_1)$
24:     return $a^{(L)}$
25: end function
```

Again, here we can observe that both DIA and ELL performs best for smaller nnz-max. max-avg do not quite affect the performance of DIA and ELL. Whereas, Majority of HYB class has good performance at higher max-avg and max-nnz.

In the plane mean-var, we observe that CSR dominates the area where the mean is high, and var is in the lower range. This implies that for matrices that have the best performance in CSR format, we have properly balanced thread distribution in GPU.

Max-nnz vs. density plane indicates that HYB format performs well for matrices that are large but has high sparsity. Large matrices with low density excel in performance using the HYB format. CSR performs better for denser matrices, and that has lower max-nnz than the HYB format. The DIA format is adversely affected by max-nnz.

Table 4.4: GPU Configurations

GPU Model	Tesla K20c
Chip	GK110
Compute Capability	3.5
Number of SMs.	13
Number of cores	2496
ALUs per SM	192
Warp size	32
Threads per CTA	1024
Threads per SM	2048
Shared memory per CTA (KB)	48
L2 cache (KB)	1536
Global memory (GB)	5
Memory bandwidth (GB/s)	208

Matrices that exhibit a lower number of diagonals and lower variance can be classified as DIA as seen in max-avg vs. variance plane. The HYB class has a higher number of diagonals with nnz along with higher variance. Whereas, ELL has a large number of diagonals and low variance.

Fill-in and max-avg are linearly related. HYB class is dominant in areas with higher max-avg and fill-in. DIA is dominant at low fill-in and low max-avg. CSR performs best when max-avg is not too high and fill- in is not too low.

In the max-avg vs. dist-avg plane, HYB follows the general trend in having the best performance for larger max-avg and dist-avg. DIA and ELL perform best when the max-avg is lower than CSR. ELL performs better than DIA when dist-avg is greater.

In max-avg vs. diag-fiill we can observe that the DIA performs best when the diagfill is low and max-avg is low. ELL, on the other hand, performs best when diag-fill is higher than DIA but with almost the same max-avg threshold. Whereas, the HYB performs better at higher diag-fill and higher max-avg.

On further analysis, we observed that the class that dominates a given area has higher performance gain as compared to other storage formats in the same region. In the areas where a single class does not dominate, we find the performance difference between the classes is quite low. Another major observation is that for all matrices which were possible to convert to ELL format the performance difference between HYB is quite low. These observations lead us to an assumption that the feature set selected can be effectively used for predicting as decision boundaries can be easily placed in areas where there are performance differences. In later sections, we prove that our results validate this assumption.

Chapter 5

Analysis and Evaluation of proposed technique

5.1 Quantitative Analysis of Deep Model

In this section, we shall evaluate the performance of our deep learning based classification model. The results reported are average from the 10-fold cross-validation performed using the model. To understand the performance of the model we calculate the accuracy of the classifier as in 5.1. Accuracy is defined as the ratio of sparse representations identified correctly by the model to the total number of matrices in the testing set. *Count Accurate* represents the count of matrices that has been correctly identified. We also compute the average performance loss of the Deep model compared to the best sparse representation (LCB) as in 5.2. The LCB value indicates how close to the best possible representation is our deep model regarding the performance. The lower the LCB, the better the model performs. Since the CSR format has higher cardinality in our dataset and as HYB is considered as an excellent storage for sparse SpMV kernels [67], we compute the gain obtained our deep model as compared to using a single sparse representation (CSR or HYB) as illustrated in 5.3 and 5.4. To further illustrate the performance gain we show the gain obtained by our deep model as compared the worst performing sparse storage for each matrix 5.5. The higher the gain, the higher the performance of our scheme.

$$Accuracy = \frac{Count\,Accurate}{|Test|} \times 100 \tag{5.1}$$

Table 5.1: Evaluation of Classifier

Accuracy	LCB	GCSR	GHYB	GELL	GDIA	GWorst
81.59%	4.4 %	154.5%	822.25%	∞	∞	1123.69%

$$LCB = \frac{1}{|Test|} \sum_{i \in Test} ((best_i - deep_i)/best_i) \times 100 \tag{5.2}$$

$$GCSR = \frac{1}{|Test|} \sum_{i \in Test} ((deep_i - csr_i)/csr_i) \times 100 \tag{5.3}$$

$$GHYB = \frac{1}{|Test|} \sum_{i \in Test} ((deep_i - hyb_i)/hyb_i) \times 100 \tag{5.4}$$

$$GWorst = \frac{1}{|Test|} \sum_{i \in Test} ((deep_i - worst_i)/worst_i) \times 100 \tag{5.5}$$

Table. 5.1 illustrates the evaluated metrics for our deep model. We achieve an accuracy of 81.59% predicting the best sparse representation based on the sparsity structure. We achieve an LCB of 4.4%, which indicates that we have reached 95.6% of the maximal possible performance. On further analysis, we note that our deep model does not classify a matrix as DIA or ELL if it is not possible to convert to it. We note that misclassifications happen when the performance of the matrix is very close to the best performing matrix. For many matrices, the performance difference between HYB and ELL is less than one GFLOP/s. This happens for the smaller matrices that can be easily converted to ELL. Compared with the use of a single format, particularly the CSR format, our model has a gain of 154.5% gain. This indicates 2.5x speedup for our model as compared to the single use of CSR throughout. Similarly, as compared to solely using HYB format our model has a speedup of 9.2x. A comparison with the worst storage schemes further demonstrates that the dynamic selection of sparse storage format improves the performance. Table 5.5 gives the performance of the deep model when using a simpler set of features. We can observe that even with the help of simple feature set we can achieve a performance greater than the ones provided using any single format for SpMV. The extra features improve the accuracy and performance and can be utilized depending on the scenario.

Table 5.2: Count of matrices with Loss $>$ 5x and $>$2x

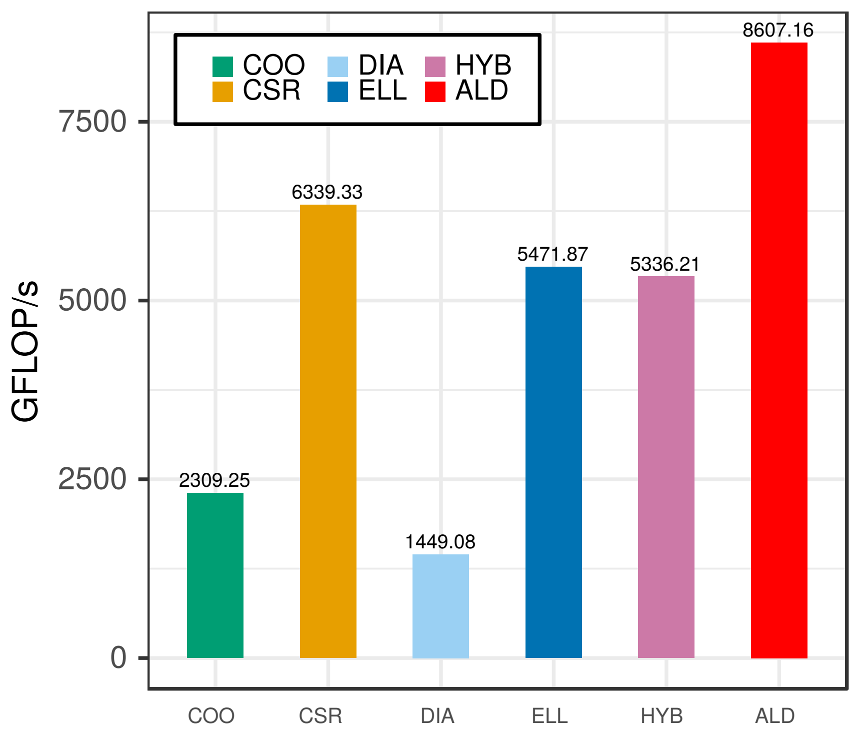
	COO	CSR	DIA	ELL	HYB	ADL
>5x	650	27	553	219	502	8
>2x	937	157	722	389	679	25

Table 5.3: Average slow down of sparse representations vs Deep learning model

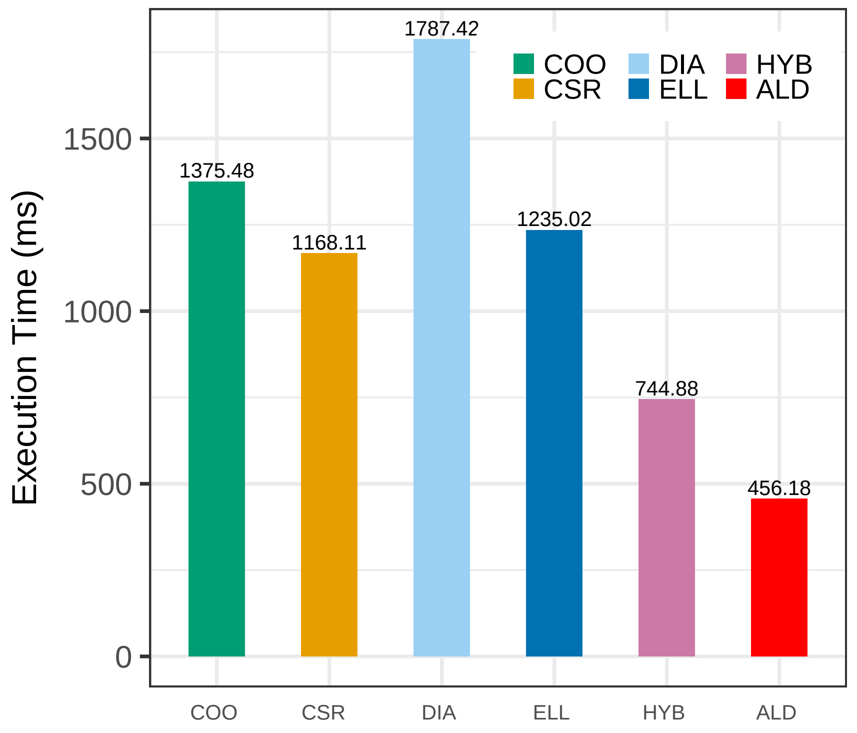
COO	CSR	DIA	ELL	HYB	ADL
3.86x	1.4x	6.15x	1.63x	1.67x	1.06x

Since we have a large dataset the average performance loss conceals the magnitude of performance loss observed for each matrix. Hence, we shall analyze the storage schemes based on the extent of slowdown. Table. 5.2 represents the total number of matrices that has more than 5x slowdown and 2x slowdown. We can clearly see that the number of matrices with more than 2x slowdown is quite high for all sparse representation schemes. We can clearly observe that our Deep model scheme improves the total performance of the sparse SpMV on GPUs. These results clearly dispels the idea that a single storage format can be used for SpMV kernel on GPUs. Rather it provides further motivation to design a storage scheme that dynamically selects the best representation of a given matrix to achieve reliable performance. We can also observe in Table. 5.3 that the total improvement comapred to the Ideal scenario is quite high using our technique.

Figure. 5.1a illustrates the total overall performance in GFLOPS as well as based on execution time achieved using our Adaptive Deep Learning based model as compared to the use of a single storage scheme. We can observe that we achieve a gain of 26.4% GFLOP/s improvement as compared to the next best scheme CSR. The DIA format due to its inability convert is the scheme that provides the least performance. The HYB format does not perform well for smaller matrices and since we have large number of smaller matrices the performance of HYB is smaller than expected. This behavior of HYB can be further illustrated using Table. 5.1b which shows the time taken for execution. We see that our proposed model executes 38.8% faster than the HYB scheme. We can observe that the HYB format has the second best execution time rather than CSR, whereas CSR had a better GFLOPS than HYB. This is because as the size of the matrices increase, the CSR performs deteriorates steeply and HYBs performance increase steeply and vice versa for smaller matrices CSR performs much better than



(a) Total GFLOP/s acheived for SpMV kernel



(b) Total execution time for SpMV kernel

Figure 5.1: GFLOPS and Execution Time comparison of various Sparse matrix storage schemes

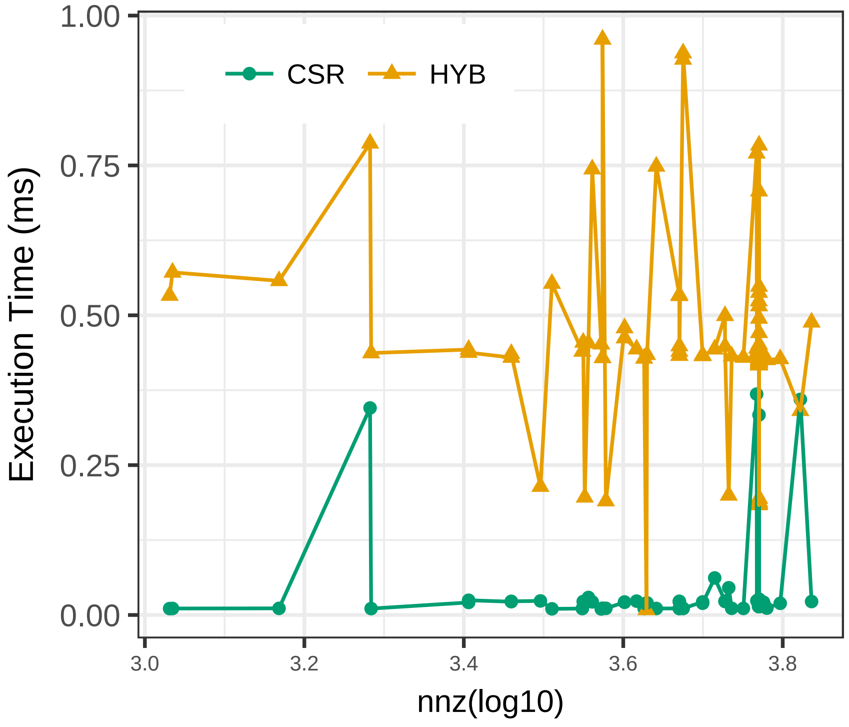
Table 5.4: Count of the best performing hundred largest and hundred smallest matrices in Dataset.

	COO	CSR	DIA	ELL	HYB	ADL
Largest 100	0	25	5	29	41	98
Smallest 100	0	76	9	14	1	97

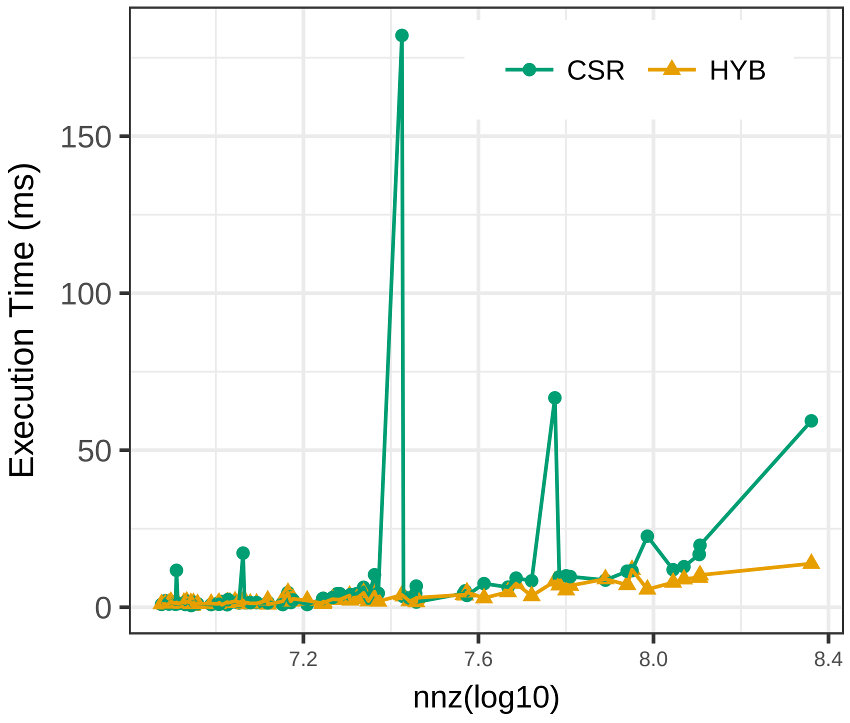
than the HYB scheme. The difference in the ratio of number of non-zeroes to the execution time is very large for CSR, for the biggest matrices as shown in Figure. 5.2a and Figure. 5.2b. As the size increases the performance of HYB increases and CSR decreases but the difference between the nnz to execution time ratio for CSR and HYB is very small. This results in HYB having a better execution time but still lower GFLOP/s than CSR. We can conclude that CSR is better suited for smaller matrices and HYB for larger matrices.

We can observe that that top half of the larger matrices all performs best with HYB, whereas CSR can be seen performing only for matrices with the smaller size. Another observation to be made is that the DIA representation gives the highest overall GFLOP/s for many of the matrices for which it is suitable. The performance gain for a matrix that has the best storage in DIA is higher than CSR or HYB by a huge factor. Hence the DIA matrix also plays a major role in achieving the overall best performance. The ELL format largely produces the best performances for mid-sized matrices in our dataset. We further provide the statistics for the hundred largest and smallest matrices to understand the relationship between size and performance. Table. 5.4 shows the number of matrices that performed the best for each sparse representation among the hundred biggest and hundred smallest matrices from our dataset. We see that our technique outperforms all other storage formats.

Figure. 5.3a and 5.3b compares our Adaptive Deep model storage scheme against the five stoarge schemes COO, CSR, DIA, ELL, and HYB for all matrices from our dataset based on the domain. On an average we can observe that our scheme has an improvement of 200 GFLOP/s as compared to other schemes. Application Domains such as 2D/3D, CFD, Structure, Circuit, Least Square, Linear programming has more than 200 GFLOP/s as compared to other schemes. Our Scheme has a slight slow down three application domains, Graphics, Statistics and unweighted graphs. But the average slowdown in these three application domain does not exceed 4 GFLOP/s. Moreover, we

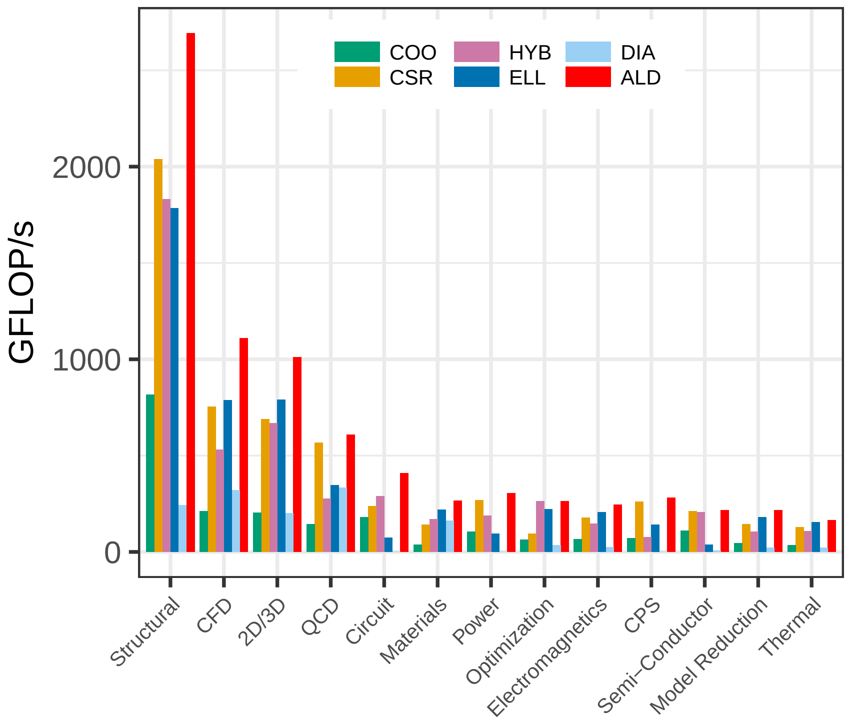


(a) Largest 100 matrices in our dataset

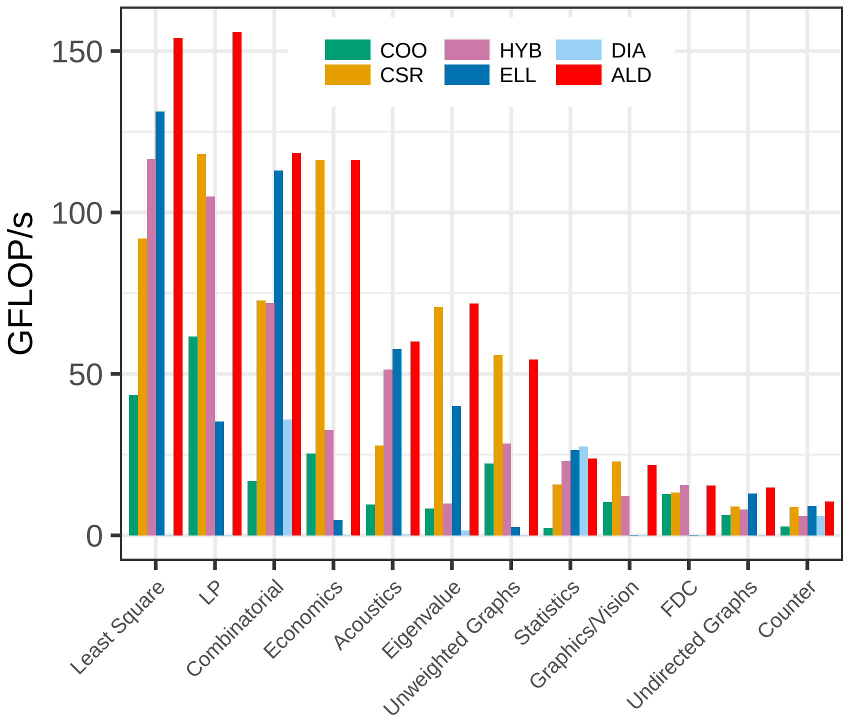


(b) Smallest 100 matrices in our dataset

Figure 5.2: Anomalies

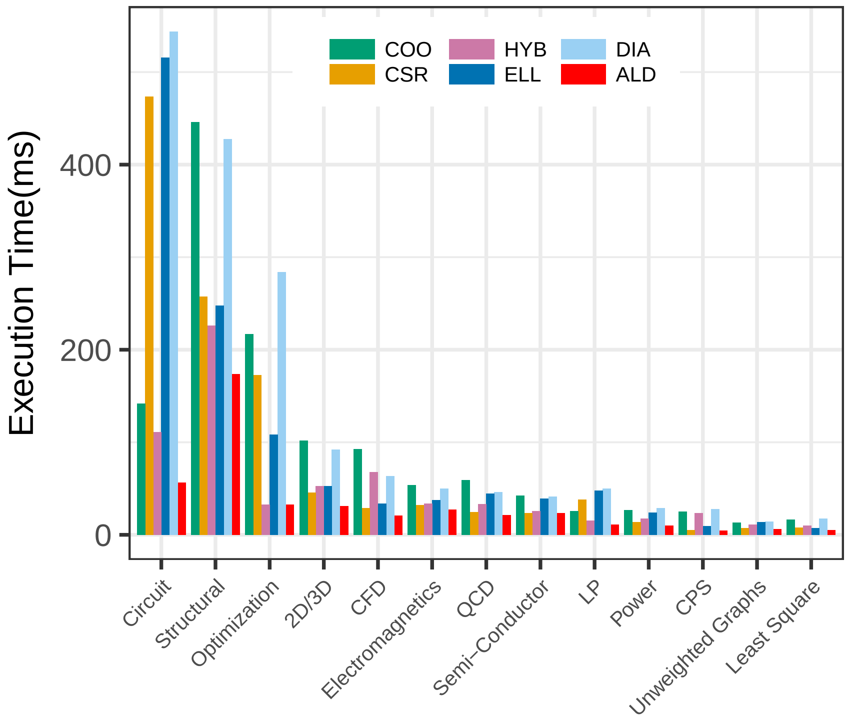


(a)

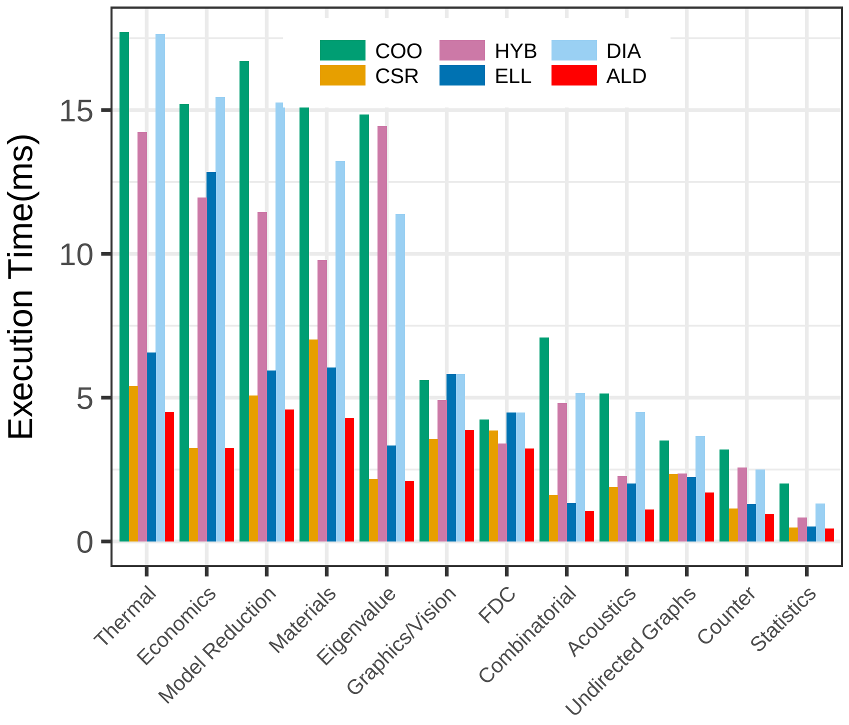


(b)

Figure 5.3: GFLOPS for various selected application domains



(a)



(b)

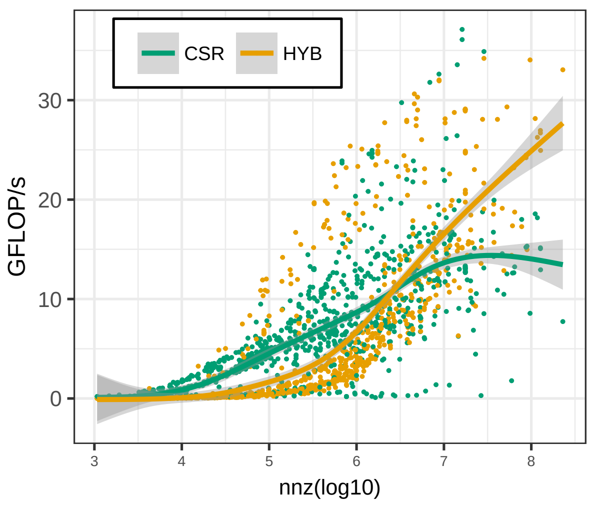
Figure 5.4: Execution time for various selected application domains

only consider three matrices for computations from the graphics/vision domain. Figure. 5.4a and 5.4b shows the run time in millisecond for sparse SpMV kernel using the five discussed formats as compared to our scheme. It can be observed that for every single application domain our scheme produces the lowest amount of time for execution.

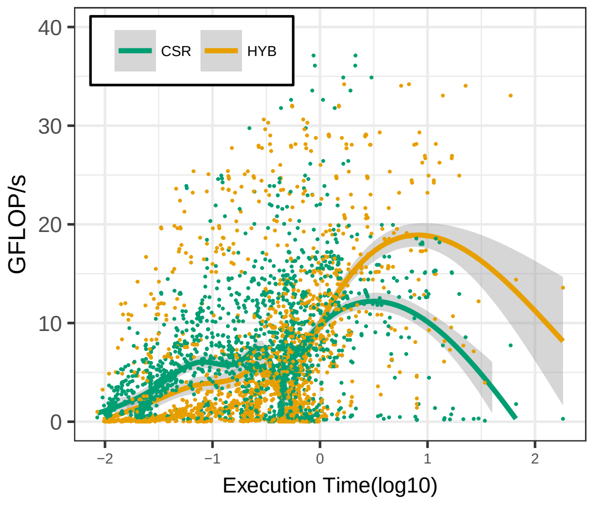
Figure. 5.5a shows the plot of the GFLOPS/s performance against the total number of non-zeroes for the HYB and CSR schemes. We can observe that the CSR scheme has a higher GFLOP/s initially for the smaller matrices and as the matrix size increases the performance also increase until a threshold where it saturates. After which it shows a slow descend in performance. The rate at which the performance deteriorates is quite small as compared to the initial rate of increase in performance. But the HYB scheme, initially lower than CSR scheme catches up with the CSR scheme and the rate of performance increase after the threshold size is quite large. We can see the difference in execution time and the difference in the GFLOPS between the CSR format and the HYB format in Figure. 5.5b.

We can observe that the difference between the two curves before the size threshold is quite low and the difference after the threshold is quite high. This is a reason why the CSR format has a greater execution time but a slightly higher GFLOP/s than the HYB format. Figure. 5.5c further illustrates the time taken against the non-zero values in the matrices. We can see that for a very few matrices the total execution time is quite high for the CSR format as compared to the HYB format. For the biggest hundred matrices the HYB format reports a total performance of 1925 GFLOP/s whereas CSR only produces 1323 GFLOP/s, whereas for the smaller matrices the CSR outperforms the HYB format.

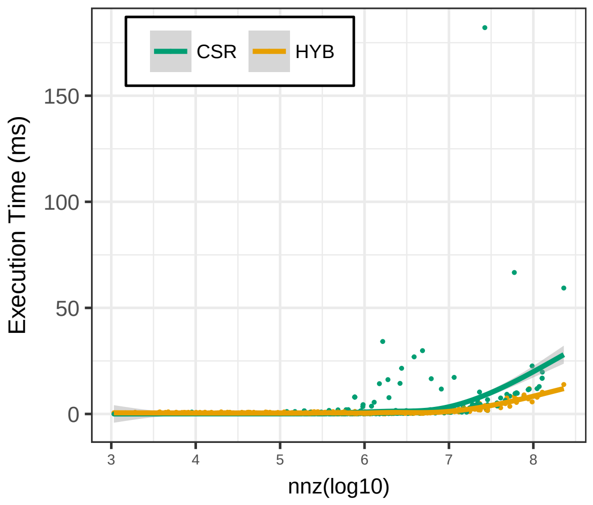
Figure. 5.6 illustrates the density plot of the GFLOP/s against the size of matrices in terms if the size of the matrices. The size of the matrices where calculated as the coo storage size, $16nnz$ bytes. In this plot the we can see the concentration of various sparse storage format depending upon the size and performance in GFLOP/s. We can clearly observe that for higher matrices HYB gives the higher GFLOP/s. However, the highest overall GFLOP/s is provided by the DIA storage format. The CSR format give excellent performance for small to medium sized matrices. The ELL format provides similar performace to that of HYB, but both ELL and DIA is not suitable for all the matrices since it becomes quite difficult to store it. Hence HYB provides a better



(a) GFLOPS vs nnz



(b) GFLOPS vs Execution time



(c) Execution time vs nnz

Figure 5.5: Anomalies Explaination

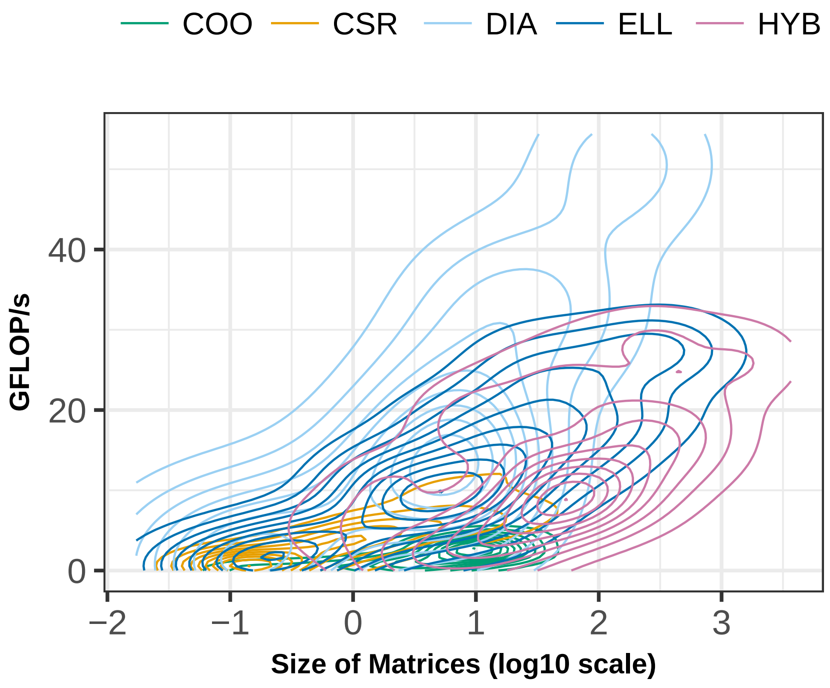
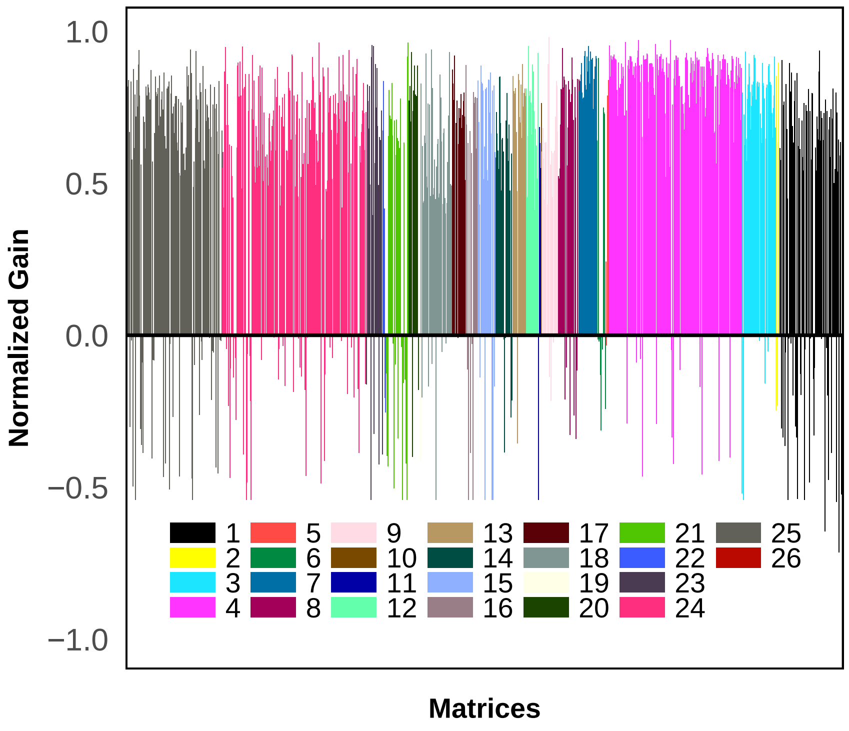


Figure 5.6: Density plot of GFLOPS/s against the size of matrices

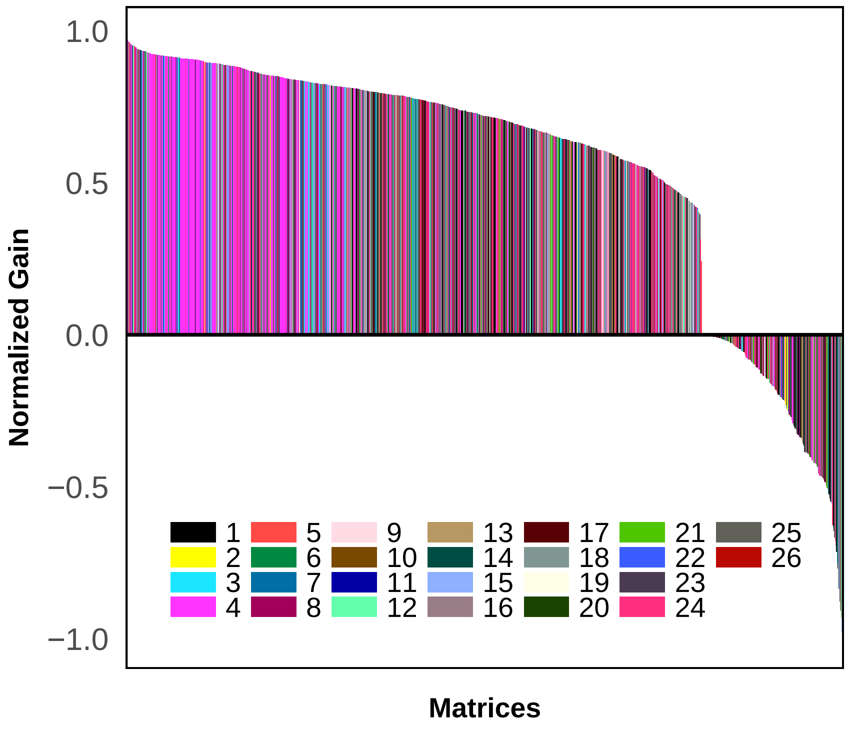
solution in such cases.

Figure. 5.7a shows the overall gain and loss of performance while using our ADL based model for all the 1056 matrices we used, sorted based on the twenty six application domains of the matrices. We can observe that our finding of 96% gain is corroborated by the figure. The positive y- axis indicates the gain and the negative y-axis indicates the loss. The width of the application domain indicates the number of matrices in that domain. Figure. 5.7b further shows the gain obtained by our ADL scheme sorted by the amount of gain. This image clearly illustrates the gain of our scheme. Figure. 5.8 shows the performance improvement that is achieved as the dimensions of the matrix increases.We extrapolate the results to show that even for higher matrices our Adaptive Deep Model attains a higher GFLOPs. We can clearly see that we can run higher matrices and get better performance. We can also use higher end GPUs like K-80 with larger amount of memory to improve the performance of SpMV in linear Iterative solvers.

Next, we test our Adaptive Deep Learning based Jacobi solver on some of the real matrices obtained from various application domains. We have selected eight matrices. Our Deep Learning-based model correctly identified seven of the eight matrices. The



(a) Gain sorted by Application domain



(b) Gain by Matrix ID

Figure 5.7: Total Gain and Loss incurred by our proposed Adaptive Deep Learning based stoarge technique.

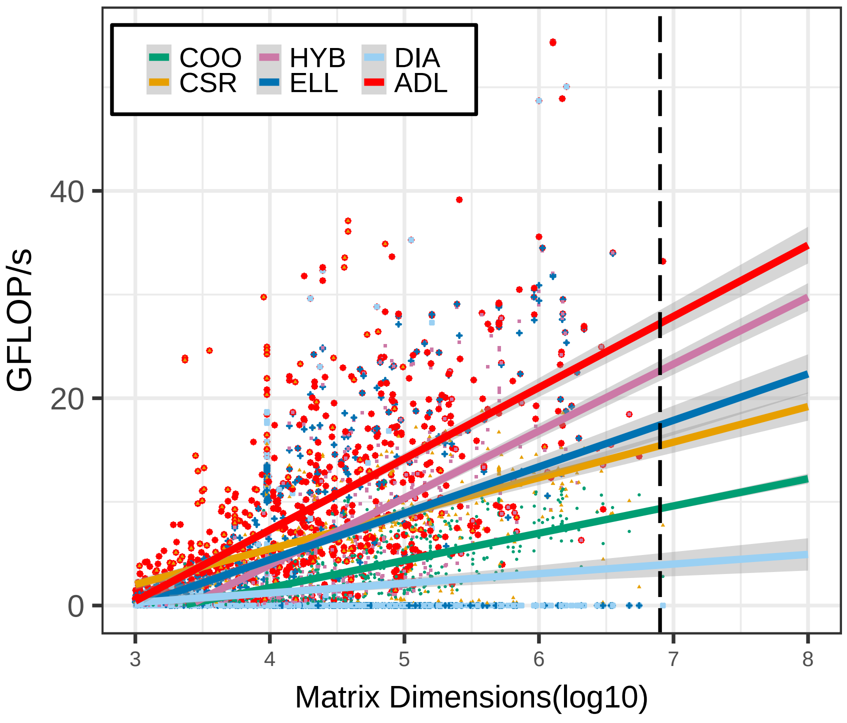


Figure 5.8: Extrapolation of the GFLOP/s performance with size of matrices

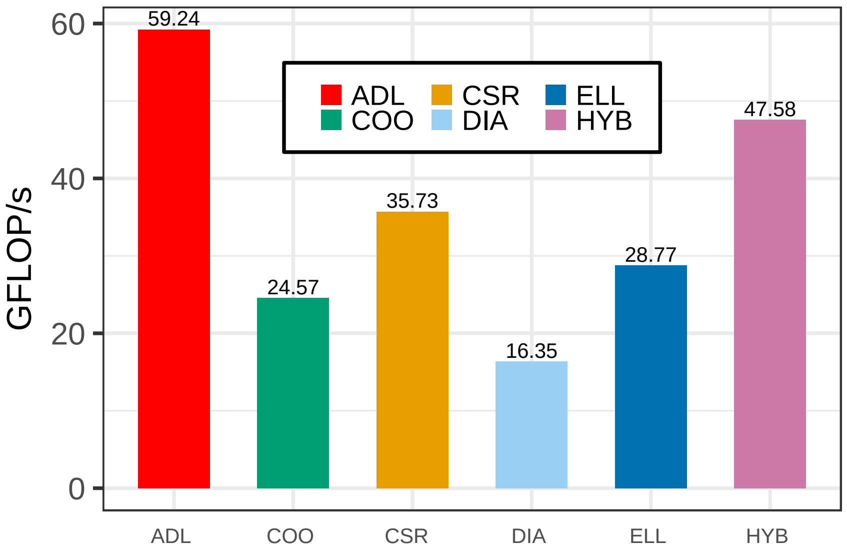


Figure 5.9: Total GFLOP/s obtained from Jacobi execution

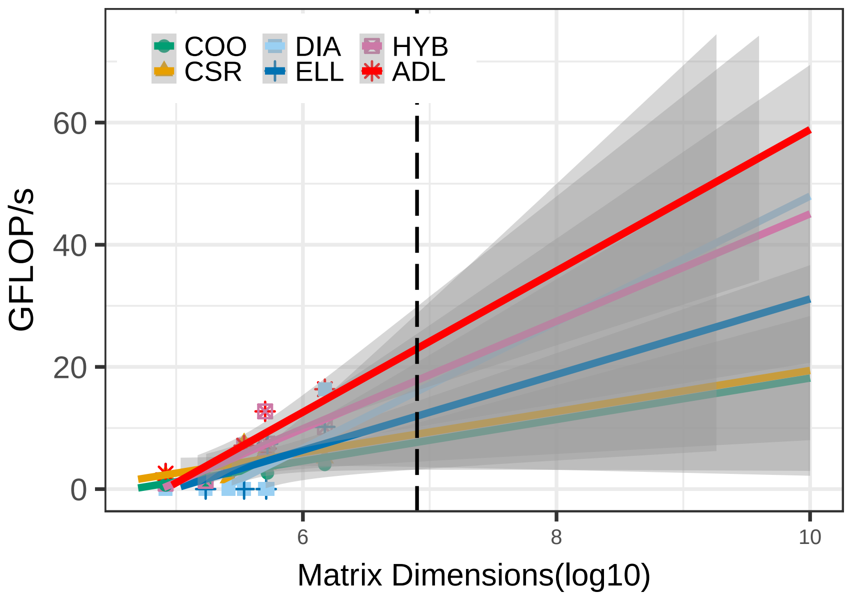


Figure 5.10: Extrapolating the GFLOP/s performance for jacobi as size increases

matrix F1 which performed well with CSR is classified as HYB. However, the difference in execution time is trivial. Figure. 5.12 gives the execution time for all the tested matrices. We can clearly observe from the results that our proposed technique clearly improves the performance of linear solvers as compared to using a single sparse storage scheme.

Figure. 5.9 shows the total GFLOP/s obtained using each of the seven matrix. Our format clearly provides the best performance. We can clearly see that our scheme is better than the HYB scheme, which is the second best scheme by 20% . We can also observe that the HYB scheme performs better than the CSR scheme. The DIA GFLOP/s reported is the GFLOP/s obtained from a single matrix, particularly, atmospheric modeling matrix. Figure. 5.10 illustrates the performance our model as compared to other models for various matrix dimensions. It can be clearly observed in the figure that our scheme's performance increases as the dimensions of the matrices increases. Moreover, we extrapolate the performance over the dimensions to indicate that even when the matrix size increases further our scheme has the better predicted performance.

Figure. 5.11 and Figure. 5.12 shows the GFLOP/s and the execution time for the seven selected matrices. We can observe that for all the matrices except the F1 matrix our scheme was able to execute with the best possible performance. In case of F1 matrix the difference between the selected HYB and the ideal CSR is very narrow. Hence does

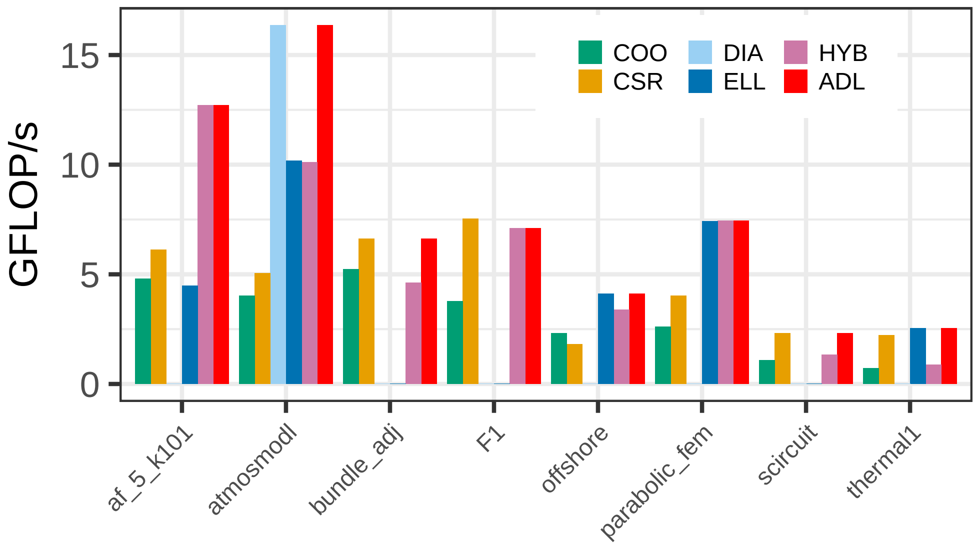


Figure 5.11: GFLOP/s obtained for Jacobi on selected matrices

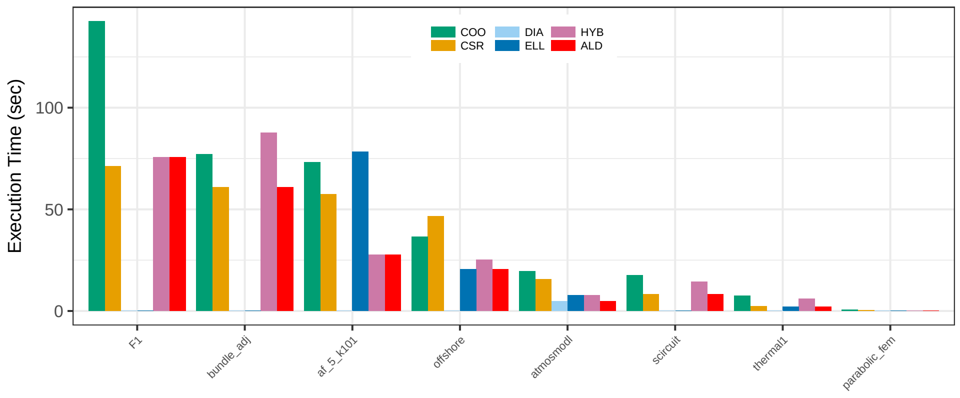


Figure 5.12: Jacobi timings for selected matrices

not affect the performance.

5.2 Comparison with state-of-art

Sedaghati et al. [180] has presented decision tree based classifier for the selection of the best sparse storage for a SpMV kernel. They have proposed the use of two set of features. A set of features that consists of simple features particularly, nnz, density, and mean. The second feature set consists of complex features that have a higher computational complexity which leads to higher overhead. Benatia et al. [181] uses multi-class SVM to perform the prediction.

We implement both models and compare with our DL based model. Decision Tree models were implemented using the Weka software libraries. We used two implementations of Decision Tree, SimpleCart and BFTree, which were used in [180]. For both the Decision tree-based models, we set the parameters to the default values, minNum=2 and numFold=5. Multiclass SVM model was implemented using the libSVM package. We used the RBF (Gaussian Radial Basis Function) as the kernel. The two parameters c and gamma were selected using the cross-validation technique provided in the package.

Table 5.5: Comparison of Deep technique Accuracy with shallow techniques Accuracy

Features	Deep	SVM	Decision Tree	
			Simple Cart	BFTree
3	10.1	10.626	10.2797	11.90
8	5.46	5.97	6.842	7.27
14	4.74	5.0	5.1082	5.876

Table 5.6: Comparison of Deep technique LCB with shallow techniques LCB

Features	Deep	SVM	Decision Tree	
			Simple Cart	BFTree
3	73.08	70.171	69.50	67.90
8	79.15	78.011	76.28	75.53
14	81	77.911	77.17	77.27

Table. 5.5 and 5.6 provides a comparison of our deep model with the above two schemes. We provide comparison using three feature sets, each consisting of three, eight, and 14 features respectively. In all the three cases, we can see that our model provides a better accuracy than the multiclass SVM or the Decision Tree-based models.

Decision Tree based classifier using Simplecart performed better than BFTree for all set of features. The addition of extra features has improved all the models by decreasing the LUB. The decline in performance for smaller feature sets is mainly due to the misprediction of the model about nonconvertible matrix formats. When a matrix that cannot be converted to DIA or ELL is predicted as either of these two, the performance of the model decreases. However, it can be observed that even when we use just three features, our Deep Learning-based model has a better accuracy and performance speedup. When we used feature, set 3, out of all 1056 matrices only two matrices were classified as DIA or ELL when the conversion was not possible. Overall our Adaptive

Deep Learning model surpass other techniques based on improving the performance of Sparse SpMV.

Chapter 6

Conclusion

Sparse Matrix-Vector multiplication (SpMV) is one of the key operations in linear algebra that lies at the heart of scientific computing and engineering. They play a significant role in solving linear system of equations using iterative methods. Sparse Kernels are level-2 BLAS operations on matrices whose entries are mostly zero so that computations with and storage of these zero elements may be eliminated. The emergence of parallel architectures, especially GPU, while offering higher computational performance, has led to the redesign of existing algorithms to suit the architecture.

In this thesis we designed novel techniques that improve the performance of sparse Jacobi iterative linear solvers on Nvidia based GPUs. A detailed review of the relevant literature was carried out to identify the challenges and the research gaps. The review revealed that the matrix sparsity structures vary widely based on the application domains and this poses major challenges in obtaining consistent high performance from sparse iterative solvers on GPUs. These challenges included coalesced memory access to the sparse matrix and vector and load balancing among threads and warps. We developed a deep learning tool that uses an extended set of features to dynamically address these challenges and invokes the most suitable storage format for the iterative solution of sparse linear equation systems. The iterative solver tool was implemented on Nvidia Tesla K20 GPU and was tested on matrices arising from real-world problems. Compared to other leading works, our tool demonstrated 25% or higher performance on average in terms of the execution time and GFLOPS.

Our approach is not limited by any GPU language nor restricted by a specific GPU architecture. It can be extended to any model of GPUs.

In our future work, we plan to implement the deep model with raw matrices sans features. We also plan to extend the tool to handle multiple GPUs.

LIST OF REFERENCES

[1] C. Woolley, “Cuda overview,” *Developer Technology Group, NVIDIA Corporation. Available online: http://www. cc. gatech. edu/˜ vetter/keeneland/tutorial-2011-04-14/02-cuda-overview. pdf (accessed on 10 September 2015)*, 2011.

[2] A. Ng, “What data scientists should know about deep learning?.”

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The landscape of parallel computing research: A view from berkeley,” Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] D. L. Golovashkin, D. G. Vorotnokova, A. V. Kochurov, and S. A. Malysheva, “Solving finite-difference equations for diffractive optics problems using graphics processing units,” *Optical Engineering*, vol. 52, no. 9, pp. 091719–091719, 2013.

[5] C. Yan, G. Zhao, T. Yue, C. Chen, J. Liu, H. Li, and N. Su, “Speeding up the high-accuracy surface modelling method with gpu,” *Environmental Earth Sciences*, vol. 74, no. 8, pp. 6511–6523, 2015.

[6] R. Mehmood and J. Crowcroft, “Parallel iterative solution method for large sparse linear equation systems,” *Computer Laboratory: University of Cambridge*, 2005.

[7] R. Garrappa, I. Moret, and M. Popolizio, “Solving the time-fractional schrÃ¶dinger equation by krylov projection methods,” *Journal of Computational Physics*, vol. 293, pp. 115 – 134, 2015. Fractional PDEsTheory, Numerics, and Applications.

[8] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web.," 1999.

[9] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub, "Extrapolation methods for accelerating pagerank computations," in *Proceedings of the 12th international conference on World Wide Web*, pp. 261–270, ACM, 2003.

[10] A. N. Langville and C. D. Meyer, "A survey of eigenvector methods for web information retrieval," *SIAM review*, vol. 47, no. 1, pp. 135–161, 2005.

[11] J. A. Buzacott and J. G. Shanthikumar, *Stochastic models of manufacturing systems*, vol. 4. Prentice Hall Englewood Cliffs, NJ, 1993.

[12] B. Kim and J. Kim, "Stability of a two-class two-server retrial queueing system," *Performance Evaluation*, vol. 88, pp. 1–17, 2015.

[13] B. Kim and J. Kim, "A single server queue with markov modulated service rates and impatient customers," *Performance Evaluation*, vol. 83, pp. 1–15, 2015.

[14] P. Buchholz, "A class of hierarchical queueing networks and their analysis," *Queueing Systems*, vol. 15, no. 1-4, pp. 59–80, 1994.

[15] W. K. Ching, "Iterative methods for queuing systems with batch arrivals and negative customers," *Bit Numerical Mathematics*, vol. 43, p. 285, June 2003.

[16] W.-K. Ching, X. Huang, M. K. Ng, and T.-K. Siu, "Queueing systems and the web," *Markov Chains*, Jan. 2013.

[17] W.-K. Ching, X. Huang, M. K. Ng, and T.-K. Siu, "Manufacturing and re-manufacturing systems," *Markov Chains*, Jan. 2013.

[18] W. J. Stewart, K. Atif, and B. Plateau, "The numerical solution of stochastic automata networks," *European Journal of Operational Research*, vol. 86, no. 3, pp. 503–525, 1995.

[19] R. H. Chan and W. K. Ching, "Circulant preconditioners for stochastic automata networks," *Numerische Mathematik*, vol. 87, no. 1, pp. 35–57, 2000.

[20] H. Heffes and D. Lucantoni, "A Markov modulated characterization of packetized voice and data traffic and related statistical multiplexer performance," *IEEE Journal on Selected Areas in Communications*, vol. 4, pp. 856–868, Sept. 1986.

[21] Y. H. Kim, B. C. Shin, and C. K. Un, "Performance analysis of leaky-bucket bandwidth enforcement strategy for bursty traffics in an atm network," *Computer Networks and ISDN Systems*, vol. 25, no. 3, pp. 295–303, 1992.

[22] J. Bylina and B. Bylina, "A markovian queuing model of a wlan node," *Computer Networks*, Jan. 2011.

[23] J. Bylina, B. Bylina, and M. Karwacki, "A markovian model of a network of two wireless devices," *Computer Networks*, jan 2012.

[24] G. Bianchi, "Performance analysis of the IEEE 802.11 distributed coordination function," *IEEE Journal on Selected Areas in Communications*, vol. 18, pp. 535–547, Mar. 2000.

[25] P. Park, P. Di Marco, P. Soldati, C. Fischione, and K. H. Johansson, "A generalized markov chain model for effective analysis of slotted ieee 802.15. 4," in *Mobile Adhoc and Sensor Systems, 2009. MASS'09. IEEE 6th International Conference on*, pp. 130–139, IEEE, 2009.

[26] A. Bustamam, K. Burrage, and N. A. Hamilton, "Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ellpack-r sparse format," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 9, pp. 679–692, May 2012.

[27] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," 2001.

[28] R. S. Varga, *Matrix iterative analysis*, vol. 27. Springer Science & Business Media, 2009.

[29] D. M. Young, *Iterative solution of large linear systems.* Academic Press, 1971.

[30] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices.* Clarendon press Oxford, 1986.

[31] O. Axelsson, *Iterative Solution Methods.* New York, NY, USA: Cambridge University Press, 1994.

[32] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU Press, 2012.

[33] R. Mehmood, "Serial disk-based analysis of large stochastic models," in *Validation of Stochastic Systems*, pp. 230–255, Springer, 2004.

[34] R. Mehmood, "Serial disk-based analysis of large stochastic models," *Validation of Stochastic Systems*, Jan. 2004.

[35] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*, vol. 43. Siam, 1994.

[36] Z.-Z. Bai, G. H. Golub, and M. K. Ng, "Hermitian and skew-hermitian splitting methods for non-hermitian positive definite linear systems," *SIAM Journal on Matrix Analysis and Applications*, vol. 24, no. 3, pp. 603–626, 2003.

[37] Z. zhi Bai and T. zhu Huang, "On the convergence of the relaxation methods for positive definite linear systems," *Journal of Computational Mathematics*, vol. 16, no. 6, pp. 527–538, 1998.

[38] Y. Saad, "Iterative methods for sparse linear systems. society for industrial and applied mathematics (siam), april 200 3," 2013.

[39] Z.-Z. Bai, "Sharp error bounds of some krylov subspace methods for non-hermitian linear systems," *Applied mathematics and computation*, vol. 109, no. 2, pp. 273–285, 2000.

[40] G. H. Golub and J. M. Ortega, *Scientific computing: an introduction with parallel computing.* Elsevier, 2014.

[41] Y. Saad and H. A. van der Vorst, "Iterative solution of linear systems in the 20th century," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1 - 2, pp. 1 – 33, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra.

[42] H. Anzt, S. Tomov, and J. Dongarra, "Energy efficiency and performance frontiers for sparse computations on gpu supercomputers," in *Proceedings of the Sixth*

International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '15, (New York, NY, USA), pp. 1–10, ACM, 2015.

[43] H. Anzt, S. Tomov, and J. Dongarra, "On the performance and energy efficiency of sparse linear algebra on gpus," *International Journal of High Performance Computing Applications*, 2016.

[44] Y. Saad, *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, second ed., 2003.

[45] M. Kwiatkowska, D. Parker, and Y. Zhang, "Dual-processor parallelisation of symbolic probabilistic model checking," in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pp. 123–130, IEEE, 2004.

[46] Y. Zhang, D. Parker, and M. Kwiatkowska, "A wavefront parallelisation of ctmc solution using mtbdds," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pp. 732–741, IEEE, 2005.

[47] Z.-Z. Bai, "Motivations and realizations of krylov subspace methods for large sparse linear systems," *Journal of Computational and Applied Mathematics*, vol. 283, pp. 71–78, 2015.

[48] G. Strang, *Computational science and engineering*, vol. 1. Wellesley-Cambridge Press Wellesley, 2007.

[49] R. Mehmood, "A survey of out-of-core analysis techniques in stochastic modelling," tech. rep., School of Computer Science, University of Birmingham, UK, August 2003.

[50] P. Vaněk, J. Mandel, and M. Brezina, "Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems," *Computing*, vol. 56, no. 3, pp. 179–196, 1996.

[51] A. Brandt, S. McCoruick, and J. Huge, "Algebraic multigrid (amg) f0r sparse matrix equati0ns," *Sparsity and its Applications*, vol. 257, 1985.

[52] C. Wen, "An accelerated two-level multigrid method for markov chains," *Journal of Applied & Computational Mathematics*, vol. 4, no. 5, pp. 1–6, 2015.

[53] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[54] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[55] X. I. Yang and R. Mittal, "Acceleration of the jacobi iterative method by factors exceeding 100 using scheduled relaxation," *Journal of Computational Physics*, vol. 274, pp. 695–708, 2014.

[56] P. P. Pratapa, P. Suryanarayana, and J. E. Pask, "Anderson acceleration of the jacobi iterative method: An efficient alternative to krylov methods for large, sparse linear systems," *Journal of Computational Physics*, vol. 306, pp. 43–54, 2016.

[57] D. G. Anderson, "Iterative procedures for nonlinear integral equations," *Journal of the ACM (JACM)*, vol. 12, no. 4, pp. 547–560, 1965.

[58] T. Rohwedder and R. Schneider, "An analysis for the diis acceleration method used in quantum chemistry calculations," *Journal of Mathematical Chemistry*, vol. 49, no. 9, pp. 1889–1914, 2011. cited By 16.

[59] F. Potra and H. Engler, "A characterization of the behavior of the anderson acceleration on linear problems," *Linear Algebra and Its Applications*, vol. 438, no. 3, pp. 1002–1011, 2013. cited By 5.

[60] H. F. Walker and P. Ni, "Anderson acceleration for fixed-point iterations," *SIAM Journal on Numerical Analysis*, vol. 49, no. 4, pp. 1715–1735, 2011.

[61] A. Touzene, "A new parallel block aggregated algorithm for solving markov chains," *The Journal of Supercomputing*, vol. 62, no. 1, pp. 573–587, 2012.

[62] V. Migallon, J. Penades, and D. B. Szyld, "Experimental study of parallel iterative solutions of markov chains with block partitions," 1999.

[63] A. Touzene, “A new parallel algorithm for solving large-scale markov chains,” *The Journal of Supercomputing*, vol. 67, no. 1, pp. 239–253, 2014.

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

[65] Y. Saad, “Sparskit: a basic tool kit for sparse matrix computations-version 2,” 1994.

[66] R. G. Grimes, D. R. Kincaid, and D. M. Young, *ITPACK 2.0 user's guide.* Center for Numerical Analysis, The University of Texas at Austin, 1979.

[67] N. Bell and M. Garland, “Efficient sparse matrix-vector multiplication on cuda,” tech. rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[68] T. Mitchell, *Machine Learning.* McGraw-Hill Education - Europe, 1997.

[69] Y. Bengio, “Learning deep architectures for ai,” *Found. Trends Mach. Learn.*, vol. 2, pp. 1–127, Jan. 2009.

[70] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, May 2015. Insight.

[71] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton, “On rectified linear units for speech processing,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 3517–3521, May 2013.

[72] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks.,” in *Aistats*, vol. 15, p. 275, 2011.

[73] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *Proc. ICML*, vol. 30, 2013.

[74] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[75] B. Xu, N. Wang, T. Chen, and M. Li, “Empirical Evaluation of Rectified Activations in Convolutional Network,” *ArXiv e-prints*, May 2015.

[76] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. C. Courville, and Y. Bengio, “Maxout networks.,” *ICML (3)*, vol. 28, pp. 1319–1327, 2013.

[77] S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv preprint arXiv:1609.04747*, Sept. 2016.

[78] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[79] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, “Parallelized stochastic gradient descent,” in *Advances in Neural Information Processing Systems 23* (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds.), pp. 2595–2603, Curran Associates, Inc., 2010.

[80] J. J. E. Dennis and J. J. MorÃ©, “Quasi-newton methods, motivation and theory,” *SIAM Review*, vol. 19, no. 1, pp. 46–89, 1977.

[81] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng, “On optimization methods for deep learning,” in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 265–272, 2011.

[82] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” Dec. 2014.

[83] M. F. Moller, “A scaled conjugate gradient algorithm for fast supervised learning,” *Neural Networks*, vol. 6, no. 4, pp. 525 – 533, 1993.

[84] S. K. Abdali and D. S. Wise, “Experiments with quadtree representation of matrices,” in *Symbolic and Algebraic Computation International Symposium ISSAC ’88 Rome, Italy, July 4–8, 1988 Proceedings*, (Berlin, Heidelberg), pp. 96–108, Springer Berlin Heidelberg, 1989.

[85] A. C. Hearn *et al.*, *REDUCE 2 user’s manual.* Department of Computer Science, Stanford University, 1970.

[86] J. Dvorský and M. Krátký, “Multi-dimensional sparse matrix storage.,” in *DATESO*, pp. 152–161, 2004.

[87] R. Fenk, “The bub-tree,” in *IN VLDB 02, PROCEEDINGS OF 28TH INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, HONG KONG*, Morgan Kaufman Publishers, 2002.

[88] I. Balk, I. Pavlovsky, A. Ushakov, and I. Landman, “Balanced binary search trees based approach for sparse matrix representation,” in *Computational Science-ICCS 2004*, pp. 1045–1048, Springer, 2004.

[89] M. A. H. Shaikh and K. M. A. Hasan, “Efficient storage scheme for n-dimensional sparse array: Gcrs/gccs,” in *2015 International Conference on High Performance Computing Simulation (HPCS)*, pp. 137–142, July 2015.

[90] I. Simecek, D. Langr, and P. Tvrdik, “Minimal quadtree format for compression of sparse matrices storage,” in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pp. 359–364, IEEE, 2012.

[91] I. Šimecek, “Sparse matrix computations using the quadtree storage format,” in *Proceedings of 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2009)*, pp. 168–173, 2009.

[92] L. Yuan, Y. Zhang, X. Sun, and T. Wang, “Optimizing sparse matrix vector multiplication using diagonal storage matrix format,” in *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pp. 585–590, Sept 2010.

[93] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, and J. P. Draayer, “Adaptive-blocking hierarchical storage format for sparse matrices,” in *Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on*, pp. 545–551, IEEE, 2012.

[94] P. T. Stathis, *Sparse matrix vector processing formats.* TU Delft, Delft University of Technology, 2004.

[95] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, “Performance optimizations and bounds for sparse matrix-vector multiply,” in *Supercomputing, ACM/IEEE 2002 Conference*, pp. 26–26, IEEE, 2002.

[96] P. Stathis, S. Vassiliadis, and S. Cotofana, “A hierarchical sparse matrix storage format for vector processors,” in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pp. 8–pp, IEEE, 2003.

[97] I. Simecek, D. Langr, and P. Tvrdík, “Space-efficient sparse matrix storage formats for massively parallel systems,” in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 54–60, IEEE, 2012.

[98] R. Shahnaz and A. Usman, “Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers.,” *Int. Arab J. Inf. Technol.*, vol. 8, no. 2, pp. 130–136, 2011.

[99] F. Smailbegovic, G. N. Gaydadjiev, and S. Vassiliadis, “Sparse matrix storage format,” in *Proceedings of the 16th annual workshop on circuits, systems and signal processing*, pp. 445–448, 2005.

[100] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks,” in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pp. 233–244, ACM, 2009.

[101] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels.* PhD thesis, Citeseer, 2003.

[102] J. Willcock and A. Lumsdaine, “Accelerating sparse matrix computations via data compression,” in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 307–316, ACM, 2006.

[103] R. W. Vuduc and H.-J. Moon, “Fast sparse matrix-vector multiplication by exploiting variable block structure,” in *High Performance Computing and Communications*, pp. 807–816, Springer, 2005.

[104] G. D. Corporation, “GAMS: Package SPBLASC,” 2017.

[105] Y. Saad, “SPARSEKIT - Sparse Matrix Utility Package,” 1994.

[106] J. Zhang, J. Wan, F. Li, J. Mao, L. Zhuang, J. Yuan, E. Liu, and Z. Yu, "Efficient sparse matrix–vector multiplication using cache oblivious extension quadtree storage format," *Future Generation Computer Systems*, vol. 54, pp. 490–500, 2016.

[107] D. Guo and W. Gropp, "Optimizing sparse data structures for matrix-vector multiply," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 115–131, 2011.

[108] D. Guo and W. Gropp, "Applications of the streamed storage format for sparse matrix operations," *International Journal of High Performance Computing Applications*, vol. 28, no. 1, pp. 3–12, 2014.

[109] S. Vassiliadis, S. Cotofana, and P. Stathis, "Block based compression storage expected performance," in *High Performance Computing Systems and Applications*, pp. 389–406, Springer, 2002.

[110] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 339–350, ACM, 2015.

[111] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 781–792, IEEE Press, 2014.

[112] G. E. Blelloch, M. A. Heroux, and M. Zagha, "Segmented operations for sparse matrix computation on vector multiprocessors," tech. rep., DTIC Document, 1993.

[113] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[114] W. Xu, H. Zhang, S. Jiao, D. Wang, F. Song, and Z. Liu, "Optimizing sparse matrix vector multiplication using cache blocking method on fermi gpu," in *Software*

Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), 2012 13th ACIS International Conference on, pp. 231–235, Aug 2012.

[115] F. Vázquez, J.-J. Fernández, and E. M. Garzón, "A new approach for sparse matrix vector product on nvidia gpus," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 8, pp. 815–826, 2011.

[116] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[117] A. Cevahir, A. Nukada, and S. Matsuoka, "Fast conjugate gradients with multiple gpus," in *Computational Science–ICCS 2009*, pp. 893–903, Springer, 2009.

[118] F. VáZquez, J. J. Fernández, and E. M. Garzón, "Automatic tuning of the sparse matrix vector product on gpus based on the ellr-t approach," *Parallel Computing*, vol. 38, no. 8, pp. 408–420, 2012.

[119] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *High Performance Embedded Architectures and Compilers*, pp. 111–125, Springer, 2010.

[120] J. Bylina, B. Bylina, and M. Karwacki, "An efficient representation on gpu for transition rate matrices for markov chains," in *Parallel Processing and Applied Mathematics*, Springer, 2013.

[121] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK.* New York, NY, USA: Springer-Verlag New York, Inc., 1984.

[122] B. Neelima and P. S. Raghavendra, "Cspr: Column only sparse matrix representation for performance improvement on gpu architecture," in *Advances in Parallel Distributed Computing*, pp. 581–595, Springer, 2011.

[123] A. J. Wijs and D. Bošnački, "Improving gpu sparse matrix-vector multiplication for probabilistic model checking," in *Model Checking Software*, pp. 98–116, Springer, 2012.

[124] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: Probabilistic symbolic model checker," in *Computer performance evaluation: modelling techniques and tools*, pp. 200–204, Springer, 2002.

[125] W. T. Tang, W. J. Tan, R. S. M. Goh, S. J. Turner, and W.-F. Wong, "A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the gpu," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 9, pp. 2373–2385, 2015.

[126] Z. Koza, M. Matyka, Ł. Mirosław, and J. Poła, "Sparse matrix-vector product," in *Numerical Computations with GPUs*, pp. 103–121, Springer, 2014.

[127] A. Dziekonski, A. Lamecki, and M. Mrozowski, "A memory efficient and fast sparse matrix vector product on a gpu," *Progress In Electromagnetics Research*, vol. 116, pp. 49–63, 2011.

[128] C. C. Yan, H. Yu, W. Xu, Y. Zhang, B. Chen, Z. Tian, Y. Wang, and J. Yin, "Memory bandwidth optimization of spmv on gpgpus," *Frontiers of Computer Science*, vol. 9, no. 3, pp. 431–441, 2015.

[129] H.-V. Dang and B. Schmidt, "The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus," *Procedia Computer Science*, vol. 9, pp. 57–66, 2012.

[130] M. Maggioni and T. Berger-Wolf, "An architecture-aware technique for optimizing sparse matrix-vector multiplication on gpus," *Procedia Computer Science*, vol. 18, pp. 329–338, 2013.

[131] M. J. Yin, X. B. Xu, H. Chen, S. B. He, and J. Hu, *Parallel Optimization for Sparse Matrix–Vector on GPU*, pp. 559–568. London: Springer London, 2013.

[132] C. Zheng, S. Gu, T.-X. Gu, B. Yang, and X.-P. Liu, "Biell: A bisection ellpack-based storage format for optimizing spmv on gpus," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2639–2647, 2014.

[133] P. Zardoshti, F. Khunjush, and H. Sarbazi-Azad, "Adaptive sparse matrix representation for efficient matrix–vector multiplication," *The Journal of Supercomputing*, vol. 72, no. 9, pp. 3366–3386, 2016.

[134] Z. Koza, M. Matyka, S. Szkoda, and L. Miroslaw, “Compressed multirow storage format for sparse matrices on graphics processing units,” *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C219–C239, 2014.

[135] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao, “Optimization of sparse matrix-vector multiplication with variant csr on gpus,” in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pp. 165–172, IEEE, 2011.

[136] P. Guo and L. Wang, “Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus,” in *Computational and Information Sciences (ICCIS), 2010 International Conference on*, pp. 1154–1157, IEEE, 2010.

[137] S. Yan, C. Li, Y. Zhang, and H. Zhou, “yaspmv: Yet another spmv framework on gpus,” in *ACM SIGPLAN Notices*, vol. 49, pp. 107–118, ACM, 2014.

[138] A. Monakov and A. Avetisyan, “Implementing blocked sparse matrix-vector multiplication on nvidia GPUs,” *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Jan. 2009.

[139] J. L. Greathouse and M. Daga, “Efficient sparse matrix-vector multiplication on gpus using the csr storage format,” in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pp. 769–780, IEEE, 2014.

[140] B. Wieczorek, M. Połomski, P. Pecka, and S. Deorowicz, “An effective way of storing and accessing very large transition matrices using multi-core cpu and gpu architectures,” in *Beyond Databases, Architectures, and Structures*, pp. 323–334, Springer, 2014.

[141] X. Feng, H. Jin, R. Zheng, Z. Shao, and L. Zhu, “A segment-based sparse matrix–vector multiplication on cuda,” *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 271–286, 2014.

[142] W. Liu and B. Vinter, “Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors,” *Parallel Computing*, vol. 49, pp. 179–193, 2015.

[143] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Graphics hardware*, vol. 2007, pp. 97–106, 2007.

[144] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 205–213, ACM, 2008.

[145] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, (New York, NY, USA), pp. 273–282, ACM, 2014.

[146] M. Maggioni and T. Berger-Wolf, "Adell: An adaptive warp-balancing ell format for efficient sparse matrix-vector multiplication on gpus," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, pp. 11–20, IEEE, 2013.

[147] M. Maggioni and T. Berger-Wolf, "Coadell: Adaptivity and compression for improving sparse matrix-vector multiplication on gpus," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pp. 933–940, IEEE, 2014.

[148] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *ACM sigplan notices*, vol. 45, pp. 115–126, ACM, 2010.

[149] A. Ekambaram and E. Montagne, "An alternative compressed storage format for sparse matrices," in *Computer and Information Sciences-ISCIS 2003*, pp. 196–203, Springer, 2003.

[150] T. Oberhuber, A. Suzuki, and J. Vacata, "New row-grouped csr format for storing the sparse matrices on gpu with implementation in cuda," *arXiv preprint arXiv:1012.2270*, 2010.

[151] M. Yang, C. Sun, Z. Li, and D. Cao, "An improved sparse matrix-vector multiplication kernel for solving modified equation in large scale power flow calculation on cuda," in *Proceedings of The 7th International Power Electronics and Motion Control Conference*, vol. 3, pp. 2028–2031, June 2012.

[152] W. Cao, L. Yao, Z. Li, Y. Wang, and Z. Wang, "Implementing sparse matrix-vector multiplication using cuda based on a hybrid sparse matrix format," in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 11, pp. V11–161, IEEE, 2010.

[153] M. Heller and T. Oberhuber, "Adaptive row-grouped csr format for storing of sparse matrices on gpu," *arXiv preprint arXiv:1203.5737*, 2012.

[154] W. Abu-Sufah and A. A. Karim, "An effective approach for implementing sparse matrix-vector multiplication on graphics processing units," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, HPCC '12, (Washington, DC, USA), pp. 453–460, IEEE Computer Society, 2012.

[155] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop, "Sparse matrix-vector multiplication on gpgpu clusters: A new storage format and a scalable implementation," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 1696–1702, IEEE, 2012.

[156] F. Stock and A. Koch, "A fast gpu implementation for solving sparse ill-posed linear equation systems," in *Parallel Processing and Applied Mathematics*, pp. 457–466, Springer, 2009.

[157] L. Ziane Khodja, R. Couturier, A. Giersch, and J. M. Bahi, "Parallel sparse linear solver with gmres method using minimization techniques of communications for gpu clusters," *The Journal of Supercomputing*, vol. 69, p. 200, July 2014.

[158] Y.-C. Chou and W.-C. Liao, "A hybrid iterative method based on mpi and cuda for steady-state solutions of markov chains," in *Proceedings of the 2nd International Conference on Intelligent Technologies and Engineering Systems (ICITES2013)*, pp. 1055–1062, Springer, 2014.

[159] C. Yan, G. Zhao, T. Yue, C. Chen, J. Liu, H. Li, and N. Su, "Speeding up the high-accuracy surface modelling method with gpu," *Environmental Earth Sciences*, vol. 74, p. 6511, Oct. 2015.

[160] N. Zhao and X. Wang, “A parallel preconditioned bi-conjugate gradient stabilized solver for the poisson problem,” *Journal of Computers*, vol. 7, no. 12, pp. 3088–3095, 2012.

[161] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, “Sparse matrix solvers on the gpu: conjugate gradients and multigrid,” in *ACM Transactions on Graphics (TOG)*, vol. 22, pp. 917–924, ACM, 2003.

[162] A. Gaikwad and I. M. Toke, “Parallel iterative linear solvers on gpu: a financial engineering case,” in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pp. 607–614, IEEE, 2010.

[163] D. Weber, J. Bender, M. Schnoes, A. Stork, and D. Fellner, “Efficient gpu data structures and methods to solve sparse linear systems in dynamics applications,” in *Computer Graphics Forum*, vol. 32, pp. 16–26, Wiley Online Library, 2013.

[164] R. Couturier and S. Domas, “Sparse systems solving on gpus with gmres,” *The journal of Supercomputing*, vol. 59, no. 3, pp. 1504–1516, 2012.

[165] J. M. Bahi, R. Couturier, and L. Z. Khodja, “Parallel gmres implementation for solving sparse linear systems on gpu clusters,” in *Proceedings of the 19th High Performance Computing Symposia*, pp. 12–19, Society for Computer Simulation International, 2011.

[166] N. Ghaemian, A. Abdollahzadeh, Z. Heinemann, A. Harrer, M. Sharifi, and G. Heinemann, “Accelerating the gmres iterative linear solver of an oil reservoir simulator using the multi-processing power of compute unified device architecture of graphics cards,” in *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA 2008), May*, pp. 13–16, 2008.

[167] M. Wang, H. Klie, M. Parashar, and H. Sudan, “Solving sparse linear systems on nvidia tesla gpus,” in *Computational Science–ICCS 2009*, pp. 864–873, Springer, 2009.

[168] T. Jost, S. Contassot-Vivier, and S. Vialle, “An efficient multi-algorithms sparse linear solver for gpus.,” in *ParCo*, pp. 546–553, 2009.

[169] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. Buijssen, M. Grajewski, and S. Turek, “Exploring weak scalability for fem calculations on a gpu-enhanced cluster,” *Parallel Computing*, vol. 33, no. 10, pp. 685–699, 2007.

[170] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek, “Using gpus to improve multigrid solver performance on a cluster,” *International Journal of Computational Science and Engineering*, vol. 4, no. 1, pp. 36–55, 2008.

[171] D. Bošnački, S. Edelkamp, D. Sulewski, and A. Wijs, “Parallel probabilistic model checking on general purpose graphics processors,” *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 1, pp. 21–35, 2011.

[172] “GPGPU Â» CUDPP.”

[173] A.-K. Cheik Ahamed and F. Magoules, “Iterative methods for sparse linear systems on graphics processing unit,” in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 836–842, IEEE, 2012.

[174] “cuSPARSE | NVIDIA Developer.”

[175] E. Cormie-Bowins, “A comparison of sequential and gpu implementations of iterative methods to compute reachability probabilities,” *arXiv preprint arXiv:1210.6412*, 2012.

[176] H. A. Van der Vorst, “Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems,” *SIAM Journal on scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[177] M. Garland, “Sparse matrix computations on manycore gpu’s,” in *Proceedings of the 45th annual Design Automation Conference*, pp. 2–6, ACM, 2008.

[178] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014.

[179] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[180] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, (New York, NY, USA), pp. 99–108, ACM, 2015.

[181] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass svm for spmv on gpu," in *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 496–505, Aug 2016.