Efficient Parallel Algorithms for Sparse Matrix Operations on a GPU with Applications

Thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

by

Dharma Teja Vooturi 201207571 dharmateja.vooturi@research.iiit.ac.in



International Institute of Information Technology, Hyderabad (Deemed to be University) Hyderabad - 500 032, INDIA June 2024 Copyright © Dharma Teja Vooturi, 2024 All Rights Reserved

International Institute of Information Technology Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Efficient Parallel Algorithms for Sparse Matrix Operations on a GPU with Applications" by Dharma Teja Vooturi, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Prof. Kishore Kothapalli

To My Parents Venkanna and Sujatha

Acknowledgments

I would like to express my sincere gratitude to my advisor Prof. Kishore Kothapalli for his excellent guidance and unwavering support through out my PhD journey. I would also like to thank Dr. Girish Varma for guiding me on a key problem in my thesis.

I would like to thank the members of my thesis committee comprising of Dr. Yogesh Simmhan from the Indian Institute of Science, Dr. Rupesh Narse from the Indian Institute of Technology (Madras), and Prof. Kamalakar Karlapalem for their valuable feedback on the thesis.

I would like to thank all IIIT faculty and staff who strive to create the right environment for students to blossom. I would like to thank my lab mates at Centre for Security, Theory, and Algorithmic Research (CSTAR), who made my time at IIIT interesting and fun. I would like to specially thank Jatin Agarwal for being there from the beginning through all the ups and downs.

I am grateful to TCS for supporting my PhD through TCS Research fellowship. I am also grateful to IBM Research and Intel Labs for providing me the necessary industry exposure through internship opportunities.

I would like to thank my parents who have strived hard to provide me better education and supported my choice of doing PhD. I would like to thank my wife Dr. Mamatha and my son Nirvaan for providing the necessary family support.

Abstract

Computational approaches use computing machines to perform functional tasks like image classification, speech recognition, fluid flow simulation, etc. A computational approach is composed of many computational tasks, and the time taken to perform a functional task using a computational approach depends on the time taken to process the underlying computational tasks of that computational approach. Furthermore, the time taken to perform a computational task depends on three factors: the computing machine, algorithm and data structures, and the input to the computational task. Improved run time performance for a computational task can be obtained by co-designing algorithm and data structures while being aware of the target computing machine and the input. A computational approach, when applied to a functional task, achieves a certain level of accuracy and takes a certain amount of time to perform. While accuracy is independent of the computing machine, time is dependent on the choice of the computing machine. Hence, better computational approaches can be designed by being aware of the computing machine. The proposed research work is categorized into two themes: 1) Design efficient algorithms and data structures for a computational task on a given instance of (computing machine, input class). 2) Design efficient computational approaches for a functional task on a given computing machine. In these themes, there are choices to be made on the computing machine, computational tasks, and computational approaches. We chose GPU for the computing machine due to its high throughput, high memory bandwidth, and its applicability in accelerating a wide variety of computational tasks from different application domains. We chose sparse matrix operations for the computational tasks, as they play a crucial role in computational approaches that are used to solve problems in application domains like scientific computing, artificial intelligence(AI), and graph analytics. We chose Artificial Neural Networks(ANN) for the computational approach, as it is able to achieve human-level accuracy for many functional tasks in the AI domain. The GPU computing machine and ANN computational approach have kickstarted the AI revolution, and we made these choices because of the opportunity it offers and the potential impact that can be created using our proposed co-design approach.

In the first part (Chapters 2-3) of the thesis, we focus on the co-design approach for designing better algorithms and data structures for computational tasks on the GPU. We work on two computational tasks, namely QBMM (Matrix Multiplication computational task ($C = A \times B$), where A and B are Quasi Band sparse matrices), and HSOLVE (Matrix solver computational task (Ax = b), where A is a Hines sparse matrix), and achieve an average speedup of $2 \times$ for QBMM and $2.5 \times$ for HSOLVE by using the co-design approach.

In the second part (Chapters 4-8) of the thesis, we focus on GPU-aware computation approaches based on Artificial Neural Networks (ANN) for efficiently performing functional tasks in the area of Artificial Intelligence (AI). The efficiency of an ANN can be measured using four metrics: accuracy, memory, compute, and runtime. Sparse ANNs, a subclass of ANNs, are shown to have better memory and compute metrics with minimal/no loss in the accuracy metric compared to dense ANNs. However, the only problem is that despite having less compute, sparse ANNs take more time to process on GPU than dense ANNs. This is because sparse ANNs have irregular compute and memory access, which is unsuitable for highly regular hardware like GPU. This problem can be alleviated by making the sparsity structured. However, introducing structure negatively affects the accuracy metric. The challenge, then is to design structured sparsity patterns that lead to better runtime metric on the GPU while maintaining the accuracy metric.

Block sparsity pattern leads to better runtime performance on the GPU when compared to the unstructured sparsity pattern, due to its regularity in compute and memory access patterns. So in Chapter 5, we propose a novel method to train block sparse ANNs from scratch. Sparse ANNs can be generated either by training from scratch or by finetuning. While training from scratch provides maximum flexibility in generating sparse ANNs, finetuning is widely used to generate sparse ANNs because it is faster and uses an already available pre-trained dense ANN as the starting point. However, the main disadvantage of the finetuning method from a runtime perspective on GPU is that finetuning is very sensitive to the imposed sparsity pattern, with accuracy inversely proportional to the rigidity in the sparsity pattern. While block sparsity pattern leads to better runtime performance on the GPU, its rigidity (a block must be either completely zero or not) leads to less accurate sparse ANNs with finetuning method. So, to fix the rigidity in block sparsity and make it more flexible for maintaining accuracy during finetuning, we introduce the idea of having multiple blocking levels in the sparsity pattern. The main advantage of a sparsity pattern with multiple blocking levels is its ability to retain essential parameters that play a crucial role in achieving good accuracy for the finetuned sparse model while maintaining the structure required for improving runtime performance.

We propose two sparsity patterns with multiple blocking levels: HB (Hierarchical Block) in Chapter 6 and RMB (Regularized Multi Block) in Chapter 7. With HB, we showed that the accuracy gap present with plain block sparsity pattern can be significantly bridged while retaining the potential to accelerate on the GPU. With a deep understanding of the GPU architecture and finetuning dynamics, we build up on the HB sparsity pattern and design a more generic, uniform, and accuracy-friendly RMB sparsity pattern by introducing regularizing constraints on multiple blocking levels while retaining the flexibility required for maintaining accuracy. With the RMB sparsity pattern, we reach an important milestone of achieving the same accuracy level as that of the unstructured sparsity pattern (most suitable for accuracy and least suited for runtime performance on GPU) while being 4-5x faster on the GPU for the industry standard image classification task on Imagenet dataset using the Resnet50 model. We further apply our RMB sparsity pattern on an image segmentation task and observe the same accuracy levels as the unstructured sparsity pattern while being 4-5x faster on the GPU. It is also important to note that on the latest A100 and H100 NVIDIA GPUs, hardware acceleration is provided for the 2:4 sparsity pattern, which is a simple configuration of our generic RMB sparsity pattern.

Edge devices are widely deployed to perform AI tasks for latency, reliability, and privacy reasons. These devices operate on a low power budget and thus have limited memory capacity and compute capabilities. Because of the importance of edge devices and their unique characteristics, there is a need to develop ANNs that are efficient on edge devices. In Chapter 8, we address this need using our proposed RBGP (Ramanujan Bipartite Graph Product) framework. The RBGP framework uses Ramanujan graphs to improve the accuracy metric by ensuring good connectivity between neurons in the ANN. The RBGP framework uses the idea of graph products for improving memory and runtime metrics by generating structured sparse ANNs with cloning and uniformity properties. On the NVIDIA Jeston Nano GPU, we are able to achieve the same accuracy levels as that of unstructured sparsity while reducing the memory requirement by half and being 7-10x faster on the GPU. Even though the RBGP sparse ANNs are designed for edge devices, they achieve similar gains on the NVIDIA V100 server GPU.

In summary, our work demonstrates that significant runtime benefit can be obtained for computational tasks and functional tasks by using the co-design approach and hardware aware computational approaches respectively. Our choices of GPU for the computing machine, sparse matrix operations for the computational tasks, and ANNs for the computational approach are made only to demonstrate the potential benefit. Nevertheless, the same ideas can be applied to different choices of computing machines, computational tasks that arise, and computational approaches that are used to solve a wide variety of functional tasks from different application domains.

Contents

Ch	apter			Page
1	Intro	duction		. 1
	1.1	Graphi	cs Processing Unit (GPU)	. 5
		1.1.1	Architecture	. 6
		1.1.2	Programming Model (CUDA)	. 6
	1.2	Sparse	Matrices	. 7
		1.2.1	Application Domains	. 8
		1.2.2	Sparse Matrix Operations on GPU	. 9
	1.3	Co-des	ign Approach for Computational Tasks	. 11
		1.3.1	Input-aware Algorithms	. 11
		1.3.2	Hardware-aware Algorithms	. 13
		1.3.3	Input-aware Data Structures	. 14
		1.3.4	Hardware-aware Data Structures	. 15
		1.3.5	Codesign of Algorithm and Data Structure	. 16
	1.4	Hardwa	are-aware Computational Approach	. 17
	1.5	Docum	ent Flow	. 18
2	01195	i-Band 9	Sparse Matrix-Matrix Multiplication	20
2	2 1	Prelimi	inaries	. 20
	2.1		hanes	. 21
	2.2	221	Matrix Partition	· 22 24
		2.2.1	Multiplying A_1 , with B_1 ,	· 24
		2.2.2	Multiplying America with Bu	. 24
		2.2.5 2 2 4	Merging Asparse with D bands	. 25
	23	Experie	mental Results and Analysis	. 20
	2.5	2 3 1	Datasets	. 20
		2.3.1	Experimental Platform	. 20
		2.3.2	Results on Real-world Datasets	. 20
		2.3.3 234	Synthetic Datasets and Results	. 20
	24	Conclu		. 30
	∠.⊤	Conclu		. 51

CONTENTS

3	Hine	es Matrix Solver	32
	3.1	Hines Matrix	35
		3.1.1 The Hodgkin-Huxley Model	35
		3.1.2 A General Form	37
		3.1.3 Necessary Conditions	39
	3.2	Our Approach	10
		3.2.1 EDD on an Undirected Graph	10
		3.2.2 Domain Matrix of Hines Matrix	11
		3.2.3 An O(N) Linear Algorithm for HinesSolver	13
		3.2.4 EDD for HinesSolver on GPU	15
		3.2.5 Implementation Details	18
	3.3	Results and Analysis	19
		3.3.1 Platform	19
		3.3.2 Dataset	19
		3.3.3 Results	19
		3.3.4 Further Experiments	50
	3.4	Conclusion	54
4	Spar	se Neural Networks on GPU	56
•	4.1	Artificial Neural Network Primer	56
	4.2	Snarse Neural Networks	57
	4.3	Runtime performance on GPU	59
	4.4	Sparsity Pattern Co-design	50
~	D		~
С	Dyna 5_1	amic Block Sparse Reparameterization of Convolutional Neural Networks)3 ~~
	5.1	Approach	5
	5.2)9 ()
		5.2.1 Effectiveness of DBSR on classification networks)9 70
		5.2.2 Comparison with block pruning	12
	5.2		13
	5.3		/4
6	Hier	archical Block Sparse Neural Networks	75
	6.1	Approach	76
		6.1.1 Pruning Methodology	17
		6.1.2 Performance Model	78
	6.2	Results	79
		6.2.1 ResNet-v2-50/Imagenet	79
	6.3	Conclusion	30

7	Regi	ularized Multi Block Sparse Neural Networks
		7.0.1 Our Contributions
	7.1	Preliminaries
		7.1.1 Sparse Matrix Structures
		7.1.2 Sparse Matrix Storage Formats
	7.2	Our Algorithm for RMBMM
		7.2.1 BLMM
		7.2.2 MBLMM
		7.2.3 RMBMM
	7.3	RMBMM on a GPU 90
		7.3.1 RMBMM-GPU
		7.3.2 Experimental Platform
		7.3.3 Exploring the RMB Sparsity Pattern Space
		7.3.3.1 Varying Block Size
		7.3.3.2 Varying Grouping
		7.3.3.3 Varying Blocklet Matrix
		7.3.3.4 Varying Multi-Blocklet Matrix
		7.3.4 Comparing RMBMM with CSRMM and BSRMM
	7.4	Regularized Multi Block (RMB) Sparse Neural Networks
		7.4.1 Generation of Sparse Neural Networks
		7.4.1.1 Unstructured Pruning
		7.4.1.2 Block Pruning
		7.4.1.3 RMB Pruning
		7.4.2 Image Classification
		7.4.2.1 VGG11/CIFAR-100
		7.4.2.2 Resnet18/CIFAR100
		7.4.2.3 Resnet50/ILSVRC12
		7.4.3 Image Segmentation
		7.4.3.1 FCN-Resnset{50,101}/VOC12:
		7.4.4 Exploring the RMB Sparsity Pattern Space
		7.4.4.1 Varying Block Size
		7.4.4.2 Varying Multi-Blocklet Matrix
	7.5	Conclusion
8	Ram	anuian Binartite Graph Products for Efficient Block Sparse Neural Networks 108
0	8 1	Preliminaries 110
	8.2	Ramanujan Bipartite Graph Product Framework 111
	0.2	8.2.1 Ramanujan Bipartite Graph Generation 115
	8.3	The RBGP Framework for GPU
	0.0	8.3.1 The RBGP4 Sparsity Pattern
	8.4	Results
		

		8.4.1	Image Classification	12	2
		8.4.2	Machine Translation	12	25
	8.5	Conclu	usion	12	27
9	Conc	clusions	and Future Directions	12	29
	9.1	Conclu	usions	12	29
	9.2	Future	Directions	13	60
		9.2.1	Optimal Structured Pruning Algorithms	13	60
		9.2.2	GPU Aware Multi Block Sparse Training	13	60
Bi	bliogr	aphy .		13	52

List of Figures

Figure		Page
1.1	NVIDIA's Volta V100 GPU (https://devblogs.nvidia.com/inside-volta/)	7
1.2	Sample of sparse matrices from scientific computing [22]	8
1.3	Sample of sparse matrices from deep learning	9
1.4	Sample of sparse matrices from graph analytics [22]	10
1.5	The influence graph of a computational task. Algorithm and data structures are used to perform a computational task. The choice of the computing machine and	
	the input to the computational task influences the design of the algorithm and data	10
1.0	structures.	12
1.6	row and column indices.DIA format only stores non zero diagonals with diagonal	
	indices	14
1.7	Multi threaded matrix vector multiplication($c = A \times b$), where each thread computes an element in c. Memory is divided into four banks, where consecutive loca-	
	tions in memory have consecutive bank id's in a cyclical manner.	15
1.8	Influence graph for the design of computational approach for performing a func- tional task. Accuracy time, and computing machine influences the design of a com-	
	nutational approach	18
1.9	Sparse matrices in QBMM and HSOLVE sparse matrix operations	19
2.1	DIA COOSE and COOSC matrix storage formats	22
2.1 2.2	Demonstration of key observations in Quasi Band Matrix Multiplication	22
2.2	Ouasi Band Matrix Multiplication	23
2.4	Results on real world datasets from [22].	29
2.5	Percentage of time taken across the various steps of our algorithm on quasi-band	20
26	Studies on quesi hand metrices with verying handDereentege and	50
2.0	spatial distribution of non-band Elements.	31
3.1	Multi compartment neuron modelling	37

LIST OF FIGURES

3.2	The general form of Hines matrix corresponding to compartmentalized neuron in	
	Figure 3.1b with $J = [10, 14]$.	39
3.3	Decomposition strategies.	46
3.4	Recursive application of fine decomposition with K=3	46
3.5	Mapping Stage-1 computation to tridiagonal solver(TRSV) with three right hand	
	sides in MIN-TRSV.	47
3.6	Results on the input dataset. Our approach (R-FINE-TPT) achieves 2.5x speedup	
	on average over MIN-TRSV approach.	50
3.7	Impact of varying K on R-FINE-TPT approach.	51
3.8	Impact of varying recursion depth D on speedup.	51
3.9	Impact of varying resolution on R-FINE-TPT and MIN-TRSV approaches.	52
3.10	Impact of varying right hand sides on speedup.	52
3.11	Threshold function for matrix with multiple right hand sides	53
3.12	Best value of K in Fine Decomposition	53
4.1	An example Sparse Feed Forward Neural Network(SNN)	58
4.2	Sparsity patterns	60
4.3	The effect of sparsity pattern on the performance of the SDMM operation on a V100 GPU. The matrix size is set to 4096 and the block size in the block sparsity	
4.4	Dual effect of sparsity pattern on the accuracy and the run time of a sparse neu- ral network for image classification task on the CIFAR-100 dataset using VGG11	60
	model. The error/accuracy for the dense neural network is 31.5/68.5.	62
4.5	Impact of sparsity pattern on accuracy and runtime metric	62
5.1	W and S are parameters of a neural network layer with block size (2x2). Before training, S is initialized to 1. During training, S is regularized to induce sparsity. At the end of the training, S becomes sparse due to regularization, and thus result	
	in a block sparse neural network layer	65
5.2	Cache friendly blocking scheme for efficiently processing BSMM (block sparse matrix multiplication) operation	68
53	Comparison of block sparse models from DBSR approach with structured sparse	00
0.0	models from (SSS) [40] for the task of image classification on Imagenet dataset	70
54	Effect of varying block size on model efficiency	71
5.7 5.5	Comparison of DBSR approach with block pruning approach on image classifica-	/ 1
5.5	tion task over CIFAR datasets for VGG network with block size 32x32	72
56	Comparison of DBSR approach with block pruning approach on image classifica-	14
5.0	tion task over CIFAR datasets for Resnet20 network with block size 8x8	73

xiv

LIST OF FIGURES

6.1	Hierarchical block sparse(HBS) neural network layer L with three parameter groups L_1, L_2 , and L_3 . The corresponding parameter group matrices M_1, M_2 , and M_3 are block sparse matrices with block sizes 4x4,2x2, and 1x1 respectively
6.2	2-Level block sparse generation : Block sizes= $[(2x2),(1x1)]$ sparsities= $[50,75]$ 78
7.1	A multi-block matrix with two block matrices of block sizes (2,3) and (2,2) 84 Negative and positive cases of a block matrix being a blocklet matrix. All the block
1.2	matrices have block size 2x2.
7.3	A regularized multi block matrix (RMB) with block size $(bh, bw) = (8, 8)$. Each block is either a multi-block at matrix or a zero matrix
7.4	CRMB storage format for storing a RMB sparse matrix with block size (6,6). Each non zero block in the matrix is stored in CMBL storage format, which inturn uses
	CBL storage format to store blocklet matrices
7.5	Varying block size in RMB sparsity pattern
7.6	Varying grouping in RMBMM-GPU Algorithm
7.7	Varying blocklet matrix for a given blocklet type
7.8	Varying multi blocklet matrix configuration
7.9	Speedup of RMBMM wrt CSR-B and BSR-B
7.10	Separation of a blocklet matrix Ab_{bl} of type (2,2) from an input matrix Ab_{res} of size (6,6). Block strengths are first calculated for each 2x2 block in A_{res} and a
	block with maximum strength is chosen for each of the three row blocks 99
7.11	Resnet50 network over ILSVRC12 dataset. Accuracy of the dense model is 76.13%. 102
8.1	Types of block sparse matrices with block size (4,4). In a BS matrix, a block is
	either completely zero or not. In a CBS matrix, all non zero blocks have the same
	non zero pattern. In a UBS matrix, all row/column blocks have equal number of
	non zero blocks. A CUBS matrix is both a UBS matrix and a CBS matrix 111
8.2	Bipartite graph product operation (\otimes_b) along with matrix view. Biadjacency matrix
0.0	of the product graph has CBS (Cloned Block Sparse) pattern with block size (2,2). 112
8.3	Biadjacency matrix BA of a bipartite graph generated using RBGP framework. BA
	has RCUBS (Recursive Cloned Uniform Block Sparse) sparsity pattern with three
81	blocking levels $(10, 10)$, $(0, 0)$ and $(2, 2)$
0.4	2-int operation on graph G. Clone graph G is first cleated and edges (u_1, v_1) and (u_2, v_2) are randomly chosen to cross over with the corresponding edges (u^c, u^c)
	(u_2, v_2) are randomly chosen to cross over with the corresponding edges (u_1, v_1) and (u_2^c, u_2^c) respectively in the clone graph 115
8.5	Sparse matrix multiplication using register blocking technique with a compatible
5.0	sparsity pattern. For a sparsity pattern to be compatible. RM row blocks in AT
	corresponding to an element group in CT should have the same sparsity pattern 120

LIST OF FIGURES

8.6	Efficiency of RBGP4MM operation on GPU. In RBGP4 sparse matrix, sparsity is	
	due to base graphs G_o and G_i . The sparsity of G_o is set based on <i>isp</i> (Sparsity	
	of G_i) and total sparsity. For denser tiles, speedups are close to ideal and as <i>isp</i>	
	increases, efficiency decreases. Size of matrices are set to 4096x4096	21
8.7	Matrices A_s , B , and C are divided into 4,2, and 2 tiles respectively. Both dense, and	
	sparse tile cases have the same 50% sparsity. But to process a tile of C in dense tile	
	case using tiling approach, only half the B matrix needs to be accessed. Whereas	
	in the sparse tile case, entire B matrix needs to be accessed.	21
8.8	Throughput for VGG19 and WRN-40-4 networks on V100 GPU	.24
8.9	Throughput for VGG19 and WideResnet-40-4 (WRN) networks on Jetson Nano2GB	
	edge GPU	25
8.10	Throughput of the key kernel (512-512-N) in Transformer network on V100 and	
	Jetson Nano GPUs.	28

List of Tables

Table	I	Page
1.1	Key sparse matrix operations	10
2.1	Glossary of Terms	21
2.2	Properties of Band Matrices. Letter K stands for a thousand, and letter M stands	77
2.3	Properties of Quasi-Band Matrices. Letter K stands for a thousand, and letter M	21
	stands for a million	27
3.1	Notation used in general form, algorithms and proof.	38
3.2	Details of neuron morphologies	51
5.1	Block sparse models generated using DBSR approach for the task of image clas- sification on Imagenet ILSVRC2012 dataset. Centre crop is used for calculating error.	70
5.2	Comparison of block sparse models from DBSR approach with models from other structured pruning methods for the task of image classification over Imagenet dataset. We can see that DBSR models are more efficient in parameters and FLOPS when	
	compared to SSS models.	71
5.3	Block sparse ERFNet models with 16x16 block size for the task of semantic seg- mentation over cityscapes dataset.	73
6.1	HBS configuration vs Retained percentage on ResNet-v2-50 model	76
6.2	Varying sparsity with block size set to 1x1.	79
6.3	Varying block size with sparsity set to 50%.	79
6.4	Varying block size with two levels. (Cumulative-Sparsity=50)	80
6.5	Hierarchical block sparsity with varying distribution. (Cumulative-Sparsity=50)	80
6.6	Quasi block sparsity with varying distribution. (Cumulative-Sparsity=50)	81

LIST OF TABLES

7.1	Storage cost (in MB) of the CRMB and the CSR data formats for storing a sparse matrix of size 4096×4096 with an RMB sparsity pattern. CRMB-(x,y) corresponds
	to an RMB sparsity pattern with blocklets of type (x,y)
7.2	VGG11 network over CIFAR100 dataset. Accuracy of the pretrained dense model
	is 68.58%
7.3	Resnet18 network over CIFAR100 dataset. Accuracy of the pretrained dense model
	in 72.9%
7.4	FCN-Resnet50 network over VOC2012 dataset. Accuracy of the pretrained dense
	model is 56.12%
7.5	FCN-Resnet101 network over VOC2012 dataset. Accuracy of the pretrained dense
	model is 60.09%
7.6	Effect of varying block size in RMB on accuracy and runtime of VGG11 sparse
	model
7.7	Effect of varying block size in RMB on accuracy and runtime of Resnet18 sparse
	model
7.8	Effect of varying block size in RMB on accuracy and runtime of FCN-Resnet50
	sparse model
7.9	Effect of varying block size in RMB on accuracy and runtime of FCN-Resnet101
	sparse model
7.10	Effect of varying blocklet types in RMB on accuracy and runtime of VGG11 sparse
	model
7.11	Effect of varying blocklet types in RMB on accuracy and runtime of Resnet18
	sparse model
7.12	Effect of varying blocklet types in RMB on accuracy and runtime of FCN-Resnet50
	sparse model
7.13	Effect of varying blocklet types in RMB on accuracy and runtime of FCN-Resnet101
	sparse model
8.1	Memory and runtime of sparsity patterns for image classification on CIFAR10 and
	CIFAR100 datasets using VGG19 networks running on V100 and Jetson Nano
	GPUs. For block pattern, we set block size to be $(4, 4)$. Memory is given in MB,
	and time is given in milliseconds for one forward pass in training with 256 batch size.122
8.2	Memory and runtime of sparsity patterns for image classification on CIFAR10 and
	CIFAR100 datasets using WideResnet-40-4 networks running on V100 and Jetson
	Nano GPUs. For block pattern, we set block size to be $(4, 4)$. Memory is given
	in MB, and time is given in milliseconds for one forward pass in training with 128
0.0	batch size
8.3	Machine translation on IWSLT (de to en) dataset using Transformer network. For
	block pattern, we set the size to be $4x4$. Dense transformer model has a BLEU
	score of 34.51 takes up 150.56MB of memory

LIST OF TABLES

Chapter 1

Introduction

The ability to process computation by machines plays an essential role in advancing science, technology, and engineering. From mechanical calculators to supercomputers, computing machines have grown tremendously in their capabilities and speed. For instance, the first commercial CPU (Central Processing Unit), Intel 4004, launched in 1971, had a clock speed of 740 kHz. Modern-day CPUs have clock speeds that go upwards of 5,000,000 kHz, a 6800x fold increment! Despite this growth, the demand for faster and more capable computing machines is like never before.

Modern day computing devices are based on the von Neumann Architecture (VNA), where a computing machine consists of a CPU (processing and control units) and a memory unit. In VNA, a computational task is expressed as a series of instructions and these instructions are processed by the CPU. Synchronization on the CPU is done using a clock, and a given instruction takes a fixed number of clock cycles to process. One of the earliest and cleanest way to improve CPU performance is to increase the clock speed. By increasing the clock speed, the time taken for a clock cycle decreases and thus the time for processing an instruction. But unfortunately, clock rates have reached their saturation, and increasing clock rates any further will result in impractical power requirements and heat dissipation. This is commonly known as the **power wall** [33].

Stagnated clock speed is not a dead end for the CPU performance. The performance of a CPU can be increased by improving the instruction throughput. An instruction is processed in multiple stages like fetch, decode and so on. One way to increase the instruction throughput is to have multiple execution units for each stage of instruction and pipeline the instructions. This way, at any given point of time multiple instructions are processed. Other techniques like out-of-order execution, branch prediction, etc., are being used to increase the instruction throughput. But all of

these techniques are limited by the amount of parallelism available at the instruction level. This is commonly known as the **ILP** (**Instruction Level Parallelism**) wall [33].

Memory unit is an integral part for CPU performance. The CPU needs to access the main memory (RAM) for reading and writing the data. Over the years, the rate at which the CPU can perform operations have grown much faster than the rate at which memory can be accessed from the RAM. This scenario leads to CPU idling, where the CPU is waiting for memory access to complete before processing the next operation. Computer architects have partially addressed this problem by the use of intermediary fast memories (cache memory) between the CPU and the RAM. The idea is to store frequently used data from the RAM in to a cache memory and reduce access time by retrieving data from a faster cache memory than from the slow RAM. But the caches are limited in size and work well only if the memory access patterns have spatial and temporal locality. The higher rate at which operations can be performed on a CPU when compared to the lower rate at which memory can be accessed from the RAM is commonly known as the **memory wall** [33].

The performance of a single core (computing unit) CPU is saturated due to the power, ILP, and memory walls. So in order to provide more performance, computer architects have shifted to multi-core designs, where a CPU has more than one core and these cores can process independent computation in parallel to increase the CPU performance. Furthermore, technologies like hyper-threading allowed for more efficient use of these multiple cores in the CPU. However, the performance gains obtained on multi-core CPUs are not ideal i.e., doubling the number of cores does not necessarily double the performance. The realized performance depends on various factors like the amount of parallelism, the frequency of synchronization, memory access patterns, arithmetic intensity, etc. Even though, a multi-core computing machine offers more performance than a single core machine, they are still subjected to power, ILP, and memory walls.

Computational tasks are diverse in nature. For example, the computational task of adding two arrays is very different from that of sorting an array. The former is embarrassingly parallel and can be completed in a single step if there are enough computing cores. In contrast, the latter cannot be performed in a single step irrespective of the number of computing cores available. This diversity in computational tasks influences the design of computing machines. By and large, computing machines can be classified into three categories, namely general-purpose, accelerator, and ASIC (Application Specific Integrated Circuit). A CPU is an example of a general-purpose machine that can process any computational task, and because of this generality, the performance is normalized across the board. A GPU (Graphics Processing Unit) is an example of an accelerator that results in more performance than a CPU for computational tasks that have a high degree of parallelism. On the contrary, the GPU has less performance than a CPU for computational tasks that are sequential.

A TPU (Tensor Processing Unit) is an example of ASIC explicitly designed to accelerate machine learning and artificial intelligence applications. Because of the diversity in computational tasks, there is a need to develop different computing machines to maximize performance.

Orthogonal to the choice of a computing machine, the performance of a computational task depends on the choice of data structures and algorithms. For example, suppose we consider the computational task of searching a number in a sorted array of size N. In that case, the linear search algorithm has O(N) time complexity, whereas the binary search algorithm has only $O(\log N)$ time complexity. In another example, suppose you consider the computational task of counting the number of edges in a sparse graph G(V, E). In that case, the adjacency matrix data structure leads to $O(N^2)$ time complexity, whereas the adjacency list data structure leads to O(N + M) time complexity, where N = |V| and M = |E|. Thus, using the right algorithms and data structures can significantly improve the performance of a computational task.

The design of data structures and algorithms is influenced by two factors: the computing machine and the input to the algorithm. An algorithm/data structure that gives good performance on one computing machine may not give similar performance on another computing machine. Similarly, an algorithm/data structure that is good for one input class may not be the right choice for another input class. Hence, the optimal performance for a computational task on a given instance of (computing machine, input class) can be obtained by designing data structures and algorithms that exploit the characteristics of the computing machine and the input class.

Computational approaches are used to perform functional tasks in the real world using computing machines. A functional task can be as trivial as solving a differential equation to as complex as simulating the brain. A computational approach is primarily evaluated using two metrics: namely accuracy and time. The goal is to design computational approaches that maximize accuracy on the functional task and minimize time on the computing machine. On a given computing machine, the runtime performance of a computational approach depends on the efficiency with which the computing machine can execute the underlying computational tasks. For example, suppose the computing machine supports parallelism, and the majority of the computational tasks in the chosen computational approach are serial. In that case, the computing machine cannot efficiently execute the underlying computational tasks.

The computing machine plays a critical role in the runtime performance of the low level computational tasks and the high level computational approaches. Apart from the computing machine, the input to a computational task also plays an important role in the runtime performance of the computational task. Given these relationships, following is the core theme of our research: • Design efficient algorithms and data structures for a computational task on a given instance of (computing machine, input class). Design efficient computational approaches for a functional task on a given computing machine.

Given the above theme, there are choices to be made on the computing machine, computational tasks, and the computational approaches. We choose GPU for the computing machine, sparse matrix operations for the computational tasks, and artificial neural networks for the computational approach. The choices are made based on the available opportunities and the potential impact. Below, we expound more on why these choices are made.

Why GPU? : GPU (Graphics Processing Unit) is a specialized computing machine that is designed to accelerate a graphics rendering pipeline. The key characteristic of a graphics rendering pipeline is that the computation is simple and parallel. The GPU is designed to exploit this by having thousands of simple cores, and a high bandwidth memory. Earlier versions of the GPU had fixed functionality with no support for programmability. Over time, the GPU has become programmable, which allowed for flexibility in the graphics rendering pipeline and led to richer graphics content. As an offshoot, the ability to program the GPU allowed for accelerating a wide range of workloads that share similar computational characteristics as that of graphics. This phenomenon is commonly referred to as GPGPU (General Purpose compute on GPU). Due to the capability of the GPU to achieve significant performance gains for a wide variety of applications while being power and cost efficient when compared to CPU, we choose GPU as the computing machine.

Why sparse matrix operations? : BLAS (Basic Linear Algebra Subprograms) is a collection of basic vector and matrix operations that are widely used in many computational approaches. GPU offers significant performance gains for many dense BLAS operations like GEMV (Multiplication of a matrix with a vector) and GEMM (Multiplication of a matrix with a matrix). Even though dense BLAS operations are ubiquitous, many computational approaches in important domains like scientific computing, deep learning, and graph analytics have sparse matrix operations, where one or more operands are sparse, i.e., a significant number of elements are zeros. Sparse matrix operations are more challenging to accelerate on GPU due to variability in the locations of non-zero elements in the operands (vector, matrix). Not only is it challenging, but it is impossible to come up with a single algorithm that provides optimal performance for all classes of the sparse operands. One aspect of our core theme of the research proposal is to design efficient algorithms and data structures that exploit the characteristics of the input to the computational task. Because of multiple sparse matrix classes (Diagonal, Block, etc.) that occur in different application domains, we choose sparse matrix operations as the computational tasks.

Why Artificial Neural Networks? : Data and compute are growing at an exponential rate. Computational approaches based on Artificial Neural Networks (ANN) that rely heavily on data and compute are now capable of achieving human-level performance on many tasks such as image classification, speech recognition, language translation, etc. Because of its importance, it is crucial to design ANNs that are efficient in compute, memory, runtime, and accuracy on the task at hand. An essential property of ANNs is that they are flexible. For the same task, one can design multiple ANNs to achieve the same level of accuracy. On a given hardware, this flexibility offered by ANN-based computational approaches allows us to design runtime-efficient ANNs without compromising on accuracy. Furthermore, training ANNs can sometimes take up to days, and improving the runtime, even by a small factor, will have a significant impact. The majority of ANN compute consists of matrix multiplication operations, which can be significantly accelerated by the GPU(our choice of hardware). Hence, we choose ANN as the computational approach for our work.

1.1 Graphics Processing Unit (GPU)

A GPU is a specialized hardware designed to accelerate computation in graphics rendering workloads. Due to strong thrust from gaming community to render more and more realistic graphics, the processing capabilities of GPUs have dramatically increased. Computation in graphics rendering workloads have following three main characteristics: 1) Large amount of computation, 2) High degree of parallelism and 3) Throughput over latency. At 4K resolution (3840×2160) and 60 frames per second (*fps*), real time rendering requires to process over half a billion pixels every second resulting in large amount of computation. As computation across pixels is independent, this results in high degree of parallelism. As the scale at which human vision operates in-terms of *fps* is limited, graphics processing workloads can be efficiently processed by focusing on throughput rather than on latency.

The availability of high computational throughput and memory bandwidth on GPUs allowed for the acceleration of workloads that share similar characteristics with graphics rendering workloads. Realizing that GPUs can be used not just for accelerating graphics processing workloads, NVIDIA developed CUDA [1], a proprietary framework for programming NVIDIA GPUs. This ability to program GPUs opened up the possibility of accelerating many workloads from different domains like scientific computing, machine learning, data analytics, etc. [66]. OpenCL [4] and SYCL [44] frameworks emerged as open alternatives to program GPUs from all vendors, including NVIDIA. GPUs from all vendors (NVIDIA, AMD, Intel, etc.) share the same key high-level architectural design of multiple simple cores and high-bandwidth memory. In this section, we provide details on the NVIDIA GPU architecture and the CUDA programming model.

1.1.1 Architecture

A GPU architecture is designed to be a throughput oriented processor aimed at workloads with high degree of fine grained parallelism and vast amount of computeA. A GPU has large number of simple cores grouped into units called streaming multiprocessors (SMs). A group of SMs are packed into a TPC (Texture Processing Clusters) and a group of TPCs are further grouped into GPC (Graphics Processing Clusters). Each SM has a set of registers, shared memory, and an L1 cache. On some of the modern GPUs, the shared memory and L1 cache are pooled into one memory sub-system and can be configured depending on the needs of a workload. Apart from having an L1 cache for each SM, a GPU has an L2 cache which is shared across all the SMs. A GPU has multiple memory controllers which are shared by all SMs to fetch data from DRAM. In order to perform computation, data has to go through the memory hierarchy of GPU all the way from DRAM to registers. Figure 1.1 shows the architecture layout of NVIDIA Volta V100 GPU model.

Execution models on parallel architectures can be broadly classified into two classes, namely SIMD (Single Instruction, Multiple data) and MIMD (Multiple Instruction Multiple Data). An example of MIMD is a multi core CPU, where each core has an independent instruction stream and can operate on different data elements. In the SIMD execution model, a single instruction stream is shared across multiple cores and accesses different data elements. A GPU follows a variant of SIMD model called the SIMT (Single Instruction Multiple Thread) execution model. In the SIMT model, each core is assigned a thread and they share a single instruction stream with the exception that some cores can be idle while others execute a single instruction. At the core of an SM, computation is processed using a warp of threads using SIMT model, where the size of a warp is fixed for a given GPU. Each SM has warp schedulers which schedules warps on to cores. When a warp requires memory access, it is the job of warp scheduler to schedule an available warp onto cores in the SM. Apart for this, some GPUs have support for accelerating matrix operations, using systolic arrays and has native support for different data types like bfloat16, tf32, fp8, int8, fp4 etc.

1.1.2 Programming Model (CUDA)

CUDA (Compute Unified Device Architecture) is a parallel computing framework developed for harnessing the computational power of GPU for general purpose compute. CUDA provides extensions for popular programming languages like C, C++ and Fortran. These extensions are used by programmer to express parallelism in a simple and easy manner. A compute task is broken down into computational tasks called GPU kernels. Each kernel is launched with a specified number of threads. Threads are grouped into thread blocks and these thread blocks are then grouped into



Figure 1.1: NVIDIA's Volta V100 GPU (https://devblogs.nvidia.com/inside-volta/)

grid blocks. Grouping of thread and grid blocks is upto the programmer and is configured for each kernel. Once the kernel is launched, thread blocks in a grid are enumerated and launched on SMs based on the availability of resources. On an SM, threads of a thread block are executed concurrently and thread blocks themselves can execute concurrently on an SM.A thread block is active until all threads within the thread block are complete. Once a thread block is completed, a new thread block is scheduled in its place. A kernel finishes when all the thread blocks in the kernel are processed.

1.2 Sparse Matrices

A sparse matrix is a matrix where a significant number of elements in the matrix have zero value. A sparse matrix operation is a matrix operation, where one or more operands are sparse. When compared to a dense matrix, the main advantage of a sparse matrix is that it can be stored and processed efficiently by skipping the storage and compute corresponding to zero elements. Sparse matrices come from many different domains of computing and have multiple operations. In this section, we first go through some key application domains and show how they are generated and the importance of sparse matrix computations in those domains. We also discuss some of the key sparse matrix operations in context of GPU hardware.

1.2.1 Application Domains

Scientific computing: Mathematical models which model natural phenomenon in science are typically represented in the form of PDE's (Partial Differential Equations). These PDE's are then solved by discretizing spatial dimension using finite difference/element methods. This discretization in space leads to sparse matrices and to compute values at those discrete locations, one needs to solve a sparse matrix system Ax = b, where A is a sparse matrix. Examples of sparse matrices that arise in scientific computing can be seen in Figure 1.2. A sparse matrix system can be solved either by using a direct method or an iterative method. But regardless of the choice one makes, both the methods involve operations on sparse matrices which need to be performed at every time step of the simulation. So parallelizing sparse computations in scientific computing greatly decreases the simulation time and will hasten scientific discoveries.



Figure 1.2: Sample of sparse matrices from scientific computing [22].

Deep learning: Modern day intelligent systems like speech recognition, face detection, language translation, etc. are based on deep neural networks. A typical deep neural network is composed of many layers. The Fully Connected (FC) layer, a key layer in deep neural networks takes in an input neuron tensor x and produces an output neuron vector y using a parameter matrix W. In the FC layer, all input neurons are connected to all output neurons. The value of neuron y_i is obtained by performing a weighted summation of x with row vector w_i . Thus, to compute all of y, we need to perform a matrix multiplication y = Wx, where W is the weight matrix with w_i as the i^{th} row. The weights of a neural network are initialized at random and are adjusted based on the gradients calculated from backpropagation technique at every time step. Once the neural network/model is sufficiently trained, pruning techniques [28–30, 49] can be employed to remove significant number of elements in weight matrix W with minimal loss to model accuracy. Pruning elements in weight matrix W, converts a dense matrix into a sparse matrix and this leads to a sparse model which has less computational complexity and storage than that of dense model. In Figure 1.3, we can see some of the sparse matrices obtained by pruning dense neural networks. Once a sparse model is generated, it is deployed in cloud/mobile/IOT setting and is processed again and again for the lifetime of the application. Hence, it becomes extremely important that sparse matrix operations that occur in deep learning are processed efficiently.



Figure 1.3: Sample of sparse matrices from deep learning.

Graph analytics: A graph models pairwise relationships between different entities and is used to capture relationships in social networks, biological networks, etc. These relationships/edges among entities are typically captured using an adjacency matrix A, where each entry A[i, j] corresponds to relationship between entities i and j. Many of the adjacency matrices that come from real world graphs are highly sparse and some of them are shown in Figure 1.4. Many graph computations like multi-source breadth first search, triangle counting, betweenness centrality etc require operations involving sparse adjacency matrix [42]. Thus by accelerating sparse matrix operations in graphs, graph analytics can be accelerated.

1.2.2 Sparse Matrix Operations on GPU

Table 1.1 lists some of the key sparse matrix operations that occur in many applications.

SPMV: Computation in SPMV ($y = A_s \times x$) involves multiplying a sparse matrix A_s with a dense vector x to produce a dense output vector y. In SPMV, each element in y can be processed independently and this leads to high degree of parallelism. As A is a sparse matrix, the number of nonzeros in each row varies and thus the amount of compute for processing elements in y. This non uniformity in compute poses challenges for GPU architectures. For example, if we naïvely



(a) as-735

(b) email-Eu-core

(c) p2p-Gnutella08

Figure 1.4: Sample of sparse matrices from graph analytics [22].

SPMV	Multiplication of a sparse matrix with a dense vector.
CSRMM	Multiplication of a sparse matrix with a dense matrix
SPMM	Multiplication of two sparse matrices
SPSOLVE	Solving a system of sparse linear equations

Table 1.1: Key sparse matrix operations.

assign a thread to compute an element y, the runtime is limited by the thread which has maximum compute. Another challenge with SpMV is that it is a bandwidth bound kernel and its performance is limited by the amount of bandwidth available on GPU.

CSRMM: In CSRMM ($C = A_s \times B$), a sparse matrix A_s is multiplied with a dense matrix B to produce a dense matrix C. As each element in C can be processed independently, CSRMM has high degree of parallelism. Similar to SPMV, computation of elements across rows in C is non uniform due to varied number of non zeros in each row. CSRMM requires multiple accesses to elements in input matrices A and B. To compute elements in i^{th} row of C, each element in C[i, :] requires loading the i^{th} row of A into memory. Similarly to compute j^{th} column in C, each element in C[i, j] requires loading the j^{th} column of B into memory. Efficient GPU algorithms exploit this by loading portions of A and B into fast memories like registers, shared memory and then reusing them as much as possible, while they are in fast memory. CSRMM is a compute bound kernel and its performance is limited by amount of compute on GPU.

SPMM: In SPMM ($C_s = A_s \times B_s$), two sparse matrices A_s and B_s are multiplied together to generate a sparse matrix C_s . Similar to CSRMM, there is abundant parallelism as each non zero element in C can be processed in parallel. Computation in SPMM is processed in following two

steps. In step 1, the non zeros in matrix C_s are identified and in step 2, computation corresponding to non zeros in C_s is performed. The reason for the division is to have less memory footprint, as we do not want to store C_s matrix in a dense format and do unnecessary compute for zeros in C_s .

SPSOLVE: Any matrix solver takes the form (Ax = b), where matrix A and vector b are known and vector x is the unknown. Solution x can be computed by using techniques like direct methods like gauss elimination, LU factorization or by iterative methods like conjugate gradient, GMRES etc. Majority of the compute in iterative methods comprises of SPMV operations. Parallel algorithms for direct methods are more challenging than iterative methods, as majority of computation in iterative methods is comprised of SPMV operations, which are reasonably optimized on GPU. For direct methods, techniques from graph theory are used to expose more parallelism and regularity.

1.3 Co-design Approach for Computational Tasks

The design of algorithms and data structures for performing a computational task is influenced by two factors, namely, the computing machine(hardware) and the input class. Figure 1.5 shows the influence graph, where the (computing machine, algorithm, data structures, input) are nodes and edges capture the influences between them. Designing an algorithm/data structure in isolation may not always lead to optimal performance. Optimal performance can be obtained by co-designing both algorithm and data structures in tandem. This co-design becomes even more critical when the computing machine and the input have strong influences. In this section, we understand these influences in detail and show that efficient algorithms and datastructures can be designed by being aware of them.

1.3.1 Input-aware Algorithms

An algorithm takes an input, processes it, and produces an output. Given two inputs, an algorithm may take different amount of time for them. For example, let us take the computational task of sorting with two inputs where the first input is already sorted and the second input is unsorted. If we use the quick sort algorithm [20] with first element as the pivot, the sorted input takes more time than the unsorted input. This behaviour is not desirable as the algorithm is taking the worst possible time for the best possible input. Now, if we know that the inputs to the algorithm are either already sorted or almost sorted, we will not get good performance by using the quick sort



Figure 1.5: The influence graph of a computational task. Algorithm and data structures are used to perform a computational task. The choice of the computing machine and the input to the computational task influences the design of the algorithm and data structures.

algorithm. This leads to the following important question : "For a given class of input, can we design better algorithms by being aware of the properties of the input class ?" The answer is yes and such algorithms are called input-aware algorithms. Let's take a simple example of MV (Multiplication of a matrix A with a vector b) operation ($c = A \times b$), where we expect input matrix A to be sparse (significant number of elements are zeros). Both Algorithms 1 and 2 perform the MV operation, but Algorithm 1 is invariant to the input and takes same amount of time for any A. In contrast, Algorithm 2 is aware of the input A being sparse and exploits it for performance by skipping computation corresponding to zero elements in A.

Algorithm 1 Multiplication of a sparse ma-	Algorithm 2 Multiplication of a sparse ma-trix A with a vector b (Input-aware)1: for i=1:N do	
$\frac{\text{trix } A \text{ with a vector } b \text{ (Input agnostic)}}{1: \text{ for } i=1:N \text{ do}}$		
2: for j=1:N do	2: for j=1:N do	
3:	3: if A[i][j] != 0 then	
4: $c[i] += A[i][j] \times b[j]$	4: $c[i] += A[i][j] \times b[j]$	
5:	5: end if	
6: end for	6: end for	
7: end for	7: end for	

1.3.2 Hardware-aware Algorithms

Computing machines(hardware) have varied features and capabilities. Due to this variability, any single algorithm does not give optimal performance on all types of hardware. For example, running a serial algorithm on a parallel hardware does not exploit the parallel computing capability of the hardware and leads to sub-optimal performance. For a given hardware, efficient algorithms can only be designed by being aware of the hardware and exploiting its capabilities. Let's take a simple example of MV (Multiplication of a matrix A with vector b) operation ($c = A \times b$) on an Intel CPU with hardware support for AVX (Advanced Vector eXtensions) instructions. AVX is a set of instructions where simple operations like addition, subtraction etc, are performed on vectors of a fixed size (4,8 etc). These vector instructions are accelerated by the underlying SIMD units in the hardware and leads to increased performance. Both Algorithms 3 and 4 perform the MV operation, where an entry in c is calculated in N/L steps (N is the size of the matrix and L is the vector size of the AVX instruction). In each step, corresponding vectors of size L are multiplied and the result is accumulated. The only difference is in Lines 6 to 8. In Algorithm 3, vector multiplication of size L is serial, whereas in Algorithm 4, it is parallelized using an AVX instruction. This results in superior performance for Algorithm 4 as it exploits the AVX capability of the hardware.

Algorithm 3 Multiplication of a matrix A	A Algorithm 4 Multiplication of a matrix A
with a vector b (Hardware agnostic)	with a vector b (Hardware-aware)
1: for i=1:N do	1: for i=1:N do
2: Vacc = $[0,,0]$ \triangleright Array of size 2	L 2: Vacc = $[0,,0]$ \triangleright Array of size L
3: for j=1:N/L do	3: for j=1:N/L do
4: $V1 = A[i][j*L:(j+1)*L]$	4: $V1 = A[i][j*L:(j+1)*L]$
5: $V2 = b[j*L:(j+1)*L]$	5: $V2 = b[j*L:(j+1)*L]$
6: for k=1:L do	6:
7: $Vacc[k] += V1[k] * V2[k]$	7: Vacc $+=$ AVX-MUL(V1,V2)
8: end for	8:
9: end for	9: end for
10: $c[i] = \sum_{l=0}^{l=L-1} \text{Vacc}[1]$	10: $c[i] = \sum_{l=0}^{l=L-1} Vacc[l]$
11: end for	11: end for



Figure 1.6: Input-aware data structures. COO format only stores non zero values along with row and column indices.DIA format only stores non zero diagonals with diagonal indices.

1.3.3 Input-aware Data Structures

Data structures play a critical role in the run time performance of a computational task. In Section 1.3.1, we have seen that efficient algorithms can be designed by being aware of the input. Similarly, efficient data structures can be designed by being aware of the input. For a given input class, the design of the data structures is driven by the properties of the input class. Let us understand this idea using data structures for storing a matrix. A matrix is typically stored in either row major format (rows are stored contiguously in memory) or in column major format (columns are stored contiguously in memory). Sparse matrices (significant number of elements are zeros) are a sub class of matrices, and if our input class is sparse matrices, then you can exploit the sparsity property and store only the non zero elements of the matrix in a generic sparse matrix format like COO (Coordinate) format, where non zeros along with its corresponding row and column indices are stored. You can take this idea even further. Within sparse matrices, diagonal sparse matrices are an input class where non zeros are aligned in the form of diagonals. Now, if the input class is diagonal sparse matrices, then you can store the matrix in DIA(diagonal) format, where non zero diagonals are stored along with diagonal indices. From Figure 1.6, we can see that, shifting to COO format for sparse matrix input class decreases memory requirement from 16 to 12 dwords, and shifting from COO format to DIA format for diagonal sparse matrix input class decreases the memory requirement from 12 to 5 dwords.



Figure 1.7: Multi threaded matrix vector multiplication $(c = A \times b)$, where each thread computes an element in c. Memory is divided into four banks, where consecutive locations in memory have consecutive bank id's in a cyclical manner.

1.3.4 Hardware-aware Data Structures

The capabilities and properties of two different hardware can be significantly different, and a single choice of data structure may not result in optimal performance. This provides an opportunity for designing efficient data structures by being aware of the hardware. Using an example of matrixvector (MV) multiplication ($c = A \times b$) for the computational task and a GPU for the hardware, let's understand the idea of hardware-aware data structures. To perform MV on a GPU, we follow a simple parallel algorithm, where each element in c is mapped to a thread, and each thread takes N time steps to complete, where N is the number of columns in the matrix A. Shared memory on a GPU is divided into banks, where consecutive memory locations have straight bank id's cyclically. For example, when the bank size is 2, memory locations 1, 2, 3, 4 have bank id's 1, 2, 1, 2. The property of the shared memory is that the memory access time is inversely proportional to the number of unique banks accessed by a warp of threads in a given time step. In Figure 1.7, we show MV operation with two data structures, namely row-major and column-padded-row-major. In the row-major case, all memory accesses into A matrix fall under a single bank in a given time step, resulting in increased memory access times and decreased run-time performance. By shifting to column-padded-row-major format, all memory accesses into A matrix fall into different banks in a given time step, resulting in reduced memory access times and improved run-time performance.

1.3.5 Codesign of Algorithm and Data Structure

Sections 1.3.1-1.3.4 showed how efficient algorithms and data structures can be designed by being aware of the hardware and input in isolation. But designing the algorithm or data structures in isolation may not always lead to better performance for a given computational task. The better option is to follow a co-design approach, where algorithms and data structures are designed in tandem. Additionally, the effects of hardware and input make it even more important to follow a co-design approach for better performance on the computational task.

In this section, let us understand the idea of co-design using a simple example. Let us take an example of the computational task of finding the L1 norm (sum of absolute values of the matrix) of a dense matrix on a CPU with a single level of cache memory. For the data structure, a dense matrix can be stored either in row-major or column-major formats (See Figure 1.6). In row/column format, the rows/columns are stored contiguously in memory. For the algorithm, the L1 norm of a matrix can be computed either using row-by-row or column-by-column algorithms as described in Algorithms 5 and 6 respectively. When looked at in isolation, the choice of data structure among row-major and col-major and the choice of algorithm among row-by-row and column-by-column are on equal footing. This choice in data structure and algorithm leads to four approaches, namely RR (Row major for data structure and Row-by-row for algorithm), RC, CR, CC. Despite making good choices for data structure and algorithms, all four approaches do not lead to equal performance due to inconsistent memory access patterns and their effect on cache reuse.

In a CPU, the cache memory sits between the processor and the RAM and is an order of magnitude faster for accessing memory. The role of cache memory is to store frequently used memory portions in DRAM and feed them to the processor at a faster rate than DRAM. The cache size is limited, and if needed, it has to make space for the new memory block by evicting the old memory block, also known as a cache miss. This cache miss is costly as it requires accessing memory from DRAM. In two of our approaches, i.e., RC and CR, every memory access will be a cache miss as the memory accesses are strided by the number of rows and columns, respectively. On the other hand, the memory access pattern for RR and RC has a stride of only 1, reducing the number of cache-misses by a factor of L, where L is the number of elements in the cache line. Hence, it is important to follow a co-design approach for achieving optimal performance on a given computational task.

Algorithm 5 L1 norm of a matrix A.	Algorithm 6 L1 norm of a matrix A.
(Row by Row)	(Column by column)
acc = 0	acc = 0
for row=1:A.rows do	for col=1:A.columns do
for col=1:A.columns do	for row=1:A.rows do
acc += A[row][col]	acc += A[row][col]
end for	end for
end for	end for

1.4 Hardware-aware Computational Approach

Computational approaches are used to perform many functional tasks like speech recognition, fluid flow simulation, protein folding, etc. Unlike a computational task, where time is the only characteristic, a functional task has an additional accuracy characteristic along with time. This dual characteristics lead to two types of computational approaches, namely accuracy-centric and time-centric. In accuracy-centric/time-centric approach, accuracy/time is optimized at the cost of time/accuracy. In reality, it is not always possible to design a computational approach that is optimal in both accuracy and time. While accuracy depends solely on the computational approach, time depends on both the computational approach and the computing machine (hardware). On a given hardware, two different computational approaches that have same accuracy and compute (number of operations) may have different time due to the difference in the efficiencies with which the underlying compute can be performed on that hardware. This effect of hardware on the time of a computational approach implies that efficient computational approaches can be designed by being aware of the hardware than by designing the computational approach in isolation. Figure 1.8 summarizes the influences on the design of the computational approach in context of the computing machine (hardware).


Figure 1.8: Influence graph for the design of computational approach for performing a functional task. Accuracy, time, and computing machine influences the design of a computational approach.

1.5 Document Flow

The thesis is divided into two parts. Part 1 comprises of Chapters 2-3, and Part 2 comprises of Chapters 4-8. Part-1 focuses on co-design approach for computational tasks, and Part-2 focuses on co-design approach for functional tasks. We conclude the thesis in Chapter 9.

PART-I

Matrix operations are the fundamental building blocks in many applications. Depending on the number of non-zeros in the matrix operands, a matrix operation can be classified into two types, namely dense matrix operation and sparse matrix operation. Unlike a dense matrix operation, where all the matrix operands are dense matrices, a sparse matrix operation has one or more matrix operands that are sparse matrices (A matrix where significant number of elements are zero values). Two sparse matrices can have same number of non-zero elements, but can vastly differ in the arrangement of non-zero elements and thus can lead to various classes of sparse matrices. This variability makes it more challenging to design a single algorithm that results in good performance across all the sparse matrix classes. Furthermore, on a hardware like GPU, different sparse matrix classes lead to different levels of irregularity in compute and memory access patterns. This provides a rich set of challenging problems to work up on using the co-design approach (Section 1.3).

Co-design approach helps in speeding up computational tasks by exploiting the properties of the input and the hardware in the design of the algorithm and data structures. In Part-1 of the thesis, we demonstrate the effectiveness of the co-design approach on the GPU hardware for two (computational tasks)/(sparse matrix operations), namely **QBMM** (Matrix Multiplication computational task (C = A * B), where A and B are **Q**uasi Band sparse matrices), and **HSOLVE** (Matrix **solver** computational task (Ax=b), where A is a **H**ines sparse matrix).



(a) Quasi Band sparse Matrix : A multi-band matrix along with non-zero elements in the non-band region.



(b) Hines sparse matrix : A symmetric matrix with only one non-zero element after each main diagonal non-zero element.

Figure 1.9: Sparse matrices in QBMM and HSOLVE sparse matrix operations

Chapter 2

Quasi-Band Sparse Matrix-Matrix Multiplication

Multiplying two sparse matrices, denoted SPMM, is an important and challenging problem in parallel computing with applications to a wide variety of disciplines including climate modeling, computational fluid dynamics, and molecular dynamics [22]. Due to the importance of SPMM, a number of works aimed at efficient algorithms and their implementations on a variety of architectures are reported in the literature [13, 52, 72].

A current trend in parallel algorithm engineering is to focus on customizing algorithms based on input characteristics. Such a customization allows the algorithms to benefit from the properties of the input. Recent examples include finding the strongly connected components of real-world graphs by Hong et al. [39], mapping graph traversals to a CPU+GPU heterogeneous platform by Gharibieh et al. [26], sparse matrix-vector multiplication and matrix-matrix multiplication of scale free matrices by Indarapu et al. [41] and Ramamoorthy et al. [72].

In this context, we note that applications such as aerodynamics and computational fluid dynamics [22] produce sparse matrices called as *quasi-band* matrices that exhibit a near-diagonal nature of sparsity. As noted by Yang et al. [85], many sparse matrices also can be reordered or divided into a near-diagonal form.

In this chapter, we focus on quasi-band matrices and design a GPU algorithm to multiply two such matrices. Our work extends the work of Yang et al. [85] who design a GPU algorithm for multiplying a quasi-band matrix with a dense vector. Our algorithm starts by separating the input quasi-band matrices into a diagonal part and the rest as a sparse part. Once such a separation is achieved, we introduce specific optimizations to perform the four multiplications: the diagonal/sparse part with the diagonal/sparse part.

Our main technical contributions can be summarized as follows.

Term	Description
nnz	Number of nonzero elements in a matrix.
nnzPercentage	Percentage of NNZ to all elements in matrix.
bandPercentage	Percentage of NNZ in band part to NNZ in matrix.
bandOccupancy	Percentage of NNZ in band part to all elements in band part.
bandCount	Number of bands present in the matrix.
diagonalCount	Total number of diagonals across all the bands in the matrix.

- We propose an algorithm (see Section 2.2) for multiplying two sparse quasi-band matrices. Our algorithm identifies the indices of the output matrix that will have nonzero entries and uses this information to manage the space required and an estimate of the work required.
- An implementation of our algorithm on an Nvidia K40 GPU achieves a speedup of 5x and 2x on average over a collection of band and quasi-band matrices, respectively, taken from the University of Florida dataset [22]. (See Section 2.3).
- We also perform experiments on synthetic quasi-band matrices to understand the effect of the nature of the matrix on the speedup. (See Section 2.3).

2.1 Preliminaries

We start with a few definitions. If all the nonzero elements in a matrix are in a single diagonal then that matrix is called a *uni-diagonal* matrix. If there exists a diagonal index pair (i, j) such that $i \leq j$ and all the non-zero elements of a matrix are present between diagonals d_i (leftOffset), d_j (rightOffset) then such a matrix is said to be a *uni-band* matrix with a bandwidth of (j - i + 1). If multiple such disjoint pairs of indices exist then it is called a *multi-band matrix*. Uni-band and multi-band matrices are commonly referred to as *band* matrices. A band matrix along with some non-zero elements in the non-band diagonals is called a *quasi-band* matrix. We also use the terms defined in Table 2.1 that indicate some of the properties of quasi-band matrices. We make use of the following data structures for storing the quasi band sparse matrix.

DIA Format [9] : Each diagonal in the matrix with at least one nonzero is stored as a column array of length equal to the number of rows in the matrix. Diagonals are numbered starting with zero for the principal diagonal and -1 and +1 for the diagonals to the left and right of the principal

	offsetArray -2	-1	0	1	2
1 2 0 0 0	data 0	0	1	2	0
3 4 5 1 0	0	3	4	5	1
4 6 7 8 0	4	6	7	8	0
0 0 9 8 7	0	9	8	7	0
00065	0	6	5	0	0
5X5 Matrix	Ď	IÅ	Fo	rn	nat

 values
 1
 2
 3
 4
 5
 1
 4
 6
 7
 8
 9
 8
 7
 6
 5

 rowIndex
 0
 0
 1
 1
 1
 2
 2
 2
 2
 3
 3
 3
 4
 4

 colIndex
 0
 1
 0
 1
 2
 3
 0
 1
 2
 3
 4
 3
 4

 rowPtr
 0
 2
 6
 9
 13
 15
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5

COOSR Format

values	1	3	4	2	4	6	5	7	9	1	8	8	6	7	5
colIndex	0	0	0	1	1	1	2	2	2	3	3	3	3	4	4
rowIndex	0	1	2	0	1	2	1	2	3	1	2	3	4	3	4
colPtr	0	3	6	9	13	3 1	۱5								

COOSC Format

Figure 2.1: DIA, COOSR, and COOSC matrix storage formats.

diagonal, and so on. These numbers, called as *diagonal offsets*, are stored in a separate array called offsetArray. Figure 2.1 shows an example of storing a sparse matrix in DIA format.

COOSR (**COOSC**) Format: These can be thought of as a mix of the COO and the CSR (CSC) formats reported in [9]. Each non-zero element in a matrix can be mapped to a triplet (value, row, column). In this format we store three arrays *data, rowIndex, and colIndex* each of length nnz(number of non-zero elements in the matrix). Elements are ordered row (*resp.* column) wise in the arrays. An array rowPtr (colPtr) of length [rows + 1]([columns + 1]) is also stored. An index *i* in rowPtr (colPtr) array maps to the index value of first element of row (column) *i* in *value* array. Figure 2.1 shows an example of storing a sparse matrix in COOSR and COOSC formats.

2.2 Algorithm

One of the prime difficulties of sparse matrix multiplication is the lack of any relation between the nature and degree of sparsity of the input matrices and their product matrix. On multi- and many-core architectures, other difficulties such as load imbalance imply that efficient sparse matrix multiplication is often challenging. One way of addressing this difficulty is to look for particular properties of the input matrices and their impact on the product matrix. To this end, focusing on quasi-band matrices, we start with the following observations.



(a) Observation 1 example : Diagonal with index -2 in A_{band} when multiplied with diagonal with index 1 in B_{band} leads to diagonal with index -1 (-2+1) in C_{bb}



(b) Observation 2 example : Non zero element at index (0,1) when multiplied with band matrix with diagonals (-1,0,1) results in contributing to output in row 0 and columns (0,1,2) (-1+1, 0+1, 1+1)

Figure 2.2: Demonstration of key observations in Quasi Band Matrix Multiplication



Figure 2.3: Quasi Band Matrix Multiplication

Observation 1. Multiplying two uni-diagonal matrices A and B with diagonal offsets a_o and b_o respectively results in another uni-diagonal matrix with diagonal offset $(a_o + b_o)$.

Observation 2. When a matrix A having a single non-zero element a_{ij} is multiplied with a uniband matrix B defined by diagonal offsets (left_o, right_o), in the product matrix $C = A \times B$, only elements in row i with column indices between $[max(0, j + left_o) : min(B_{cols} - 1, j + right_o)]$ are effected.

To make use of the above observations, in the matrix product $C = A \times B$, we start by partitioning quasi-band matrices A and B into their band and non-band components denoted A_{bands} and A_{sparse} , B_{bands} and B_{sparse} respectively. As Figure 2.3 the matrices A and B can be written as the sum of A_{bands} , A_{sparse} , and B_{bands} , B_{sparse} , respectively. Appealing to the distributive nature of matrix multiplication over addition, we notice that the product matrix $C := A \times B$ is the result of adding the matrices C_{bb} , C_{sb} , C_{bs} , and C_{ss} =, where $C_{\text{bb}} = A_{\text{bands}} \times B_{\text{bands}}$, $C_{\text{sb}} = A_{\text{sparse}} \times B_{\text{bands}}$, $C_{\text{bs}} = A_{\text{bands}} \times B_{\text{sparse}}$ and $C_{\text{ss}} = A_{\text{sparse}} \times B_{\text{sparse}}$. An outline of the above approach is shown in Algorithm 7.

The computation of each of the four matrix products along with how to achieve the required partitioning of A and B is described in subsections 2.2.1–2.2.3. For computing C_{SS} we use the SPMM kernel from NVIDIA's cusparse library [2].

Algorithm 7 Algorithm for multiplying quasi-band m	natrices A and B into C .
--	---------------------------------

1: $A = A_{bands} + A_{sparse} (A_{bands} \text{ is } A_{band}^1 + A_{band}^2 + \dots + A_{band}^m)$ 2: $B = B_{bands} + B_{sparse} (B_{bands} \text{ is } B_{band}^1 + B_{band}^2 + \dots + B_{band}^n)$ 3: $C_{bb} = A_{bands} \times B_{bands}$ 4: $C_{sb} = A_{sparse} \times B_{bands}$ 5: $C_{bs} = A_{bands} \times B_{sparse}$ 6: $C_{ss} = A_{sparse} \times B_{sparse}$ 7: $C = MERGE(C_{bb}, C_{sb}, C_{bs}, C_{ss})$

2.2.1 Matrix Partition

We first calculate diaOccupancy for each diagonal in the matrix. DiaOccupancy for a diagonal is defined as the percentage of non-zero elements in that diagonal to the rows of the matrix. We filter all the diagonals which have diaOccupancy greater than a threshold diaOcc to a set S. Diagonals in set S are potential pivots of bands. We start by identifying the band surrounding the diagonal with the largest diaOccupancy in S. This is done by including this diagonal and the diagonals to its left and right so long as their bandOccupancy is more than a threshold bandOcc. These diagonals are then recognized as one band and the diagonals from S which intersect with the band are removed from S. The process is repeated to locate other bands of diagonals until S is empty. The diagonals that are chosen as part of some band are arranged in the DIA format. The left over elements are arranged in the COOSR format for A_{sparse} and COOSC format for B_{sparse} . As threshold diaOcc is used for finding pivot diagonals of bands, we keep the value at 50%. We also set bandOcc to be 40%, to ensure that bands are nearly half dense.

2.2.2 Multiplying *A*_{bands} with *B*_{bands}

In this section we present optimizations that can be applied when we multiply two band matrices. We perform the multiplication in two steps: a preprocessing step and a computation step.

In the preprocessing step, we use Observation 1 to allocate the correct space for the matrix C_{bb} which we store in the DIA format. Each diagonal from A_{bands} having offset a_o can then be multiplied with each diagonal in B_{bands} having offset b_o in parallel. Furthermore each row element

r in the resultant diagonal can be updated atomically in parallel using equation $C_{bb}(a_o + b_o, r) +=$ $A_{bands}(a_o, r) \times B_{bands}(b_o, r + a_o)$ provided diagonal offset $a_o + b_o$ and row index $r + a_o$ are valid. (For a matrix *M* represented in the DIA format, M(o, r) refers to the element with diagonal offset *o* and row number *r*.). An outline of computation step is shown in Algorithm 8.

Algorithm 8 Multiplying A_{bands} with B_{bands} .

1:	for a_o in A_{bands} . of fset Array do
2:	for b_o in B_{bands} .offsetArray do
3:	for r in $[0:A_{rows})$ do
4:	if $(-A_{rows} < a_o + b_o < B_{cols})$ and $(0 \le r + a_o < A_{rows})$ then
5:	$C_{bb}(a_o + b_o, r) = A_{bands}(a_o, r) * B_{bands}(b_o, r + a_o)$
6:	end if
7:	end for
8:	end for
9:	end for

2.2.3 Multiplying Asparse with B_{bands}

We now use Observation 2 in this computation. We start with a preprocessing step where we find all the column segments effected by $A_{\text{sparse}} \times B_{bands}^i$ for any *i*. These segments may overlap with each other. We proceed by merging these segments so that we can then allocate necessary storage for the matrix C_{sb} in the COOSR format.

In the actual computation step, we note from Observation 2 that an element e from A_{sparse} at row r and column c can be multiplied with each diagonal in B_{bands} having offset b_o in parallel. Such a computation effects at most one element in C_{sb} with row r and column $(c + b_o)$. As C_{sb} is stored in the COOSR format, this element needs to be introduced in to row r of C_{sb} by doing binary search on column indices of row r. An outline of computation step is shown in Algo 9. Using the COOSR format instead of CSR format increases the efficiency of our algorithm as we have simultaneous access to the row and the column indices of elements. This can be observed in steps 3 and 4 in Algorithm 9.

The multiplication of A_{bands} with B_{sparse} is similar to that of $A_{\text{sparse}} \times B_{\text{bands}}$. In this case row segments will be effected, instead of column segments. Another change to note is that we store the matrix C_{bs} in the COOSC format.

	Al	gorithm	9	Multiplying	Asparse	with Bsparse.
--	----	---------	---	-------------	---------	---------------

1: for i in $[0: A_{sparse}.NNZ)$ do for b_o in B_{bands} .offsetArray do 2: $r = A_{sparse}.rowIndex[i]$ 3: $c = A_{sparse}.colIndex[i] + b_o$ 4: 5: if $0 \le c < B.cols$ then index = $SEARCH(c, C_{sb}.colIndex[C_{sb}.rowPtr[r]:C_{sb}.rowPtr[r+1]-1])$ 6: 7: $C_{sb}.data[index] + = A_{sparse}.data[i] * B_{bands}(b_o, c)$ end if 8: 9: end for 10: end for

2.2.4 Merging

In the merging step, an element e at row r and column c in the matrix C has to be accumulated from corresponding elements in the four sub-products C_{bb} , C_{sb} , C_{bs} and C_{ss} . This is done in four steps. In the first step, for every element with row index r and column index c in the matrix C_{bs} we check if there is a corresponding element in any of the matrices C_{sb} , C_{ss} , and C_{bb} . If it exists, we consolidate the contribution of the element (r, c) in C_{bs} with one of the other three matrices. In the second step, we consolidate overlapping elements in C_{ss} with elements in matrices C_{sb} and C_{bb} . In the third step, we consolidate overlapping elements in C_{sb} with elements in matrix C_{bb} . Having easy access to both the row and column indices via the COOSR/COOSC formats makes the merge process efficient compared to using CSR/CSC formats.

We now have all the four subproducts that do not have any overlapping elements, but they all are not in the same format. In the fourth step, we convert matrices C_{bs} and C_{bb} into the COOSR format leaving us with four non-overlapping matrices in the COOSR format. It is now easy to combine these four matrices to a single matrix C in the COOSR format. (See also [72].)

2.3 Experimental Results and Analysis

2.3.1 Datasets

We experiment with real-world band and quasi-band matrices from the University of Florida sparse matrix collection [22]. The matrices we use and some of the properties are shown in Tables 2.2 and 2.3. We also experiment with a variety of synthetic datasets that help us understand the impact of the nature of the matrix on our algorithm. These are described in Section 2.3.4.

Matrix	Rows	NNZ	Band	Diagonal	Band	Band
			Count	Count	Occupancy	Spread
Bai/af23560	24K	484K	5	33	62.50	12.90
Boeing/crystm03	25K	583K	27	39	88.80	689.21
Castrillon/denormal	89K	1156K	5	13	99.75	6.67
Averous/epb1	15K	95K	3	11	58.96	3.77
Norris/fv1	10K	87K	3	9	99.32	4.08
Nasa/nasa2146	2K	72K	3	45	76.63	13.79
Boeing/pcrystk03	25K	1751K	9	99	72.66	76.53
Oberwolfach/windscreen	22K	1482K	9	99	79.36	909.68
Nemeth/nemeth21	10K	1173K	1	169	73.39	0

Table 2.2: Properties of Band Matrices. Letter K stands for a thousand, and letter M stands for a million.

Table 2.3: Properties of Quasi-Band Matrices. Letter K stands for a thousand, and letter M stands for a million.

Matrix	Rows	NNZ	Nonband	Band	Diagonal	Band	Band
			Elements	Count	Count	Percentage	Occupancy
Schenk_IBMSDS/	103K	2M	9.5K	5	31	99.55	66.73
matrix_9							
Fluorem/PR02R	161K	8.2M	61K	5	108	99.25	47
Boeing/pwtk	218K	11M	2.1M	3	71	81.46	61.31
Simon/raefsky3	21K	1.5M	34K	3	93	97.71	75.48
Schenk_AFE/af_shell9	505K	18M	10M	1	25	42.93	59.83
Norris/heart1	3557	1.4M	0.95M	1	239	31.51	52.32
DNVS/trdheim	22K	1.9M	1.4M	1	35	25.92	64.88
HB/cegb2802	2802	277K	156K	1	67	43.73	65
MathWorks/Sieber	2290	15K	8K	1	5	46.18	60.02
Muite/Chebyshev3	4101	37K	16K	1	9	55.59	44.43

2.3.2 Experimental Platform

We use the K40 GPU from the NVidia Tesla series in our experiments. The host on which K40 is mounted is an Intel i7-4790K CPU with 32GB of global memory. To program the GPU we used CUDA API Version 6.5.

2.3.3 Results on Real-world Datasets

We show results as speedup when compared to SPMM kernel (cusparseScsrgemm) in NVIDIA's cusparse library [2] and SPMM kernel (mkl_scsrmultcsr) from the Intel MKL library [3]. Both cusparseScsrgemm and mkl_scsrmultcsr expect the input matrices to be in the CSR format. The computation is done in single precision.

Band Matrices: Figure 2.4a shows the speedup achieved by our algorithm on band matrices from Table 2.2. In our algorithm, only steps partitioning and computing $A_{bands} \times B_{bands}$ need to be executed as the input matrices are band matrices. Speedup variations can be partly explained as follows.

Firstly, notice that for matrices with more bandOccupancy, the number of unproductive computations in our algorithm reduce. Hence, for our algorithm, a high bandOccupancy is helpful. On the other hand, given that the input and the output matrices are band matrices, the number of nonzeros in the output matrix depends on how bands are spread in the input matrices. We capture the above via parameter bandSpread which is calculated as follows. Take the middle diagonal in each band to be a pivot diagonal. Calculate the distance (absolute difference of the diagonal offset values) between all pairs of pivot diagonals. Divide sum by dimension and multiply it by 100 to arrive at bandSpread of the matrix. Table 2.2 shows the bandSpread value for all matrices considered. It can now be noticed that the performance of the SPMM routine from cusparse degrades as bandSpread increases. Hence, we expect that a high bandSpread usually results in a bigger speedup keeping other parameters fixed.

Following the above, consider matrices af23560, crystm03, pcrystk03, and windscreen that have near equal size. Matrix af23560 has less bandOccupancy and less bandSpread when compared to matrix crystm03. Hence the former has less speedup. Matrix windscreen has high bandOccupancy and high bandSpread compared to that of matrix pcrystk03 resulting in a better speedup. Matrices epb1, fv1, and nasa2146 have very small size and NNZ compared to others. To offset the cost of partitioning and pre-processing, a reasonable size and NNZ are desirable. Hence these matrices show a lesser speedup. Among matrices epb1 and fv1, epb1 has low bandOccupancy and low bandSpread resulting in low speedup.



(a) Band \times Band. Average speedup of 5x and 6x is of 2x(Excluding Chebyshev3) and 1.3x is achieved achieved when compared to cuSPARSE and MKL when compared to cuSPARSE and MKL respectively.

Figure 2.4: Results on real world datasets from [22].

Quasi-band Matrices: Figure 2.4b shows the speedup achieved by our algorithm on quasiband matrices from Table 2.3. For quasi-band matrices, the performance of our algorithm and also that of the SPMM library routine from cusparse depends on various factors such as nnzPercentage, bandPercentage, bandOccupancy, bandCount, diagonalCount, bandSpread, and the spatial distribution of non-band elements. Because of various dependencies among these factors, it is in general not possible to explain the speedup achieved.

On matrix_9, the high speedup achieved is due to its high bandPercentage. On the other hand, though the matrix PR02R has a bandPercentage comparable to that of matrix_9, the speedup is not as high due to poor bandOccupancy. For matrix trdheim, the low speedup can be attributed to the fact that all the nonzero elements are present in a small band of diagonal indices (-531 to 531). This ensures that also the cusparse library routine performs well. The matrix Chebyshev3 has its nonzero elements outside of the bands cohesively within the first four rows. Such a structure is likely to create load imbalance in the cusparse library routine for SPMM which results in a high speedup for our algorithm. (See also Experiment 3 in Section 2.3.4).

Profile: In Figure 2.5, we show the time taken by various steps of our algorithm on matrices from Table 2.3 as a percentage of the overall time. As we use cusparse library for computing C_{ss} , this suggests that indeed multiplying sparse matrices is always difficult and focusing on the nature of sparsity is usually beneficial.



Figure 2.5: Percentage of time taken across the various steps of our algorithm on quasi-band matrices from Table 2.3.

2.3.4 Synthetic Datasets and Results

We conduct three experiments with synthetic quasi-band matrices to understand the factors that influence the speedup and runtime of our algorithm. We consider synthetic matrices with 16,000 rows, an nnzPercentage of 0.5%, a bandOccupancy of 90% and a bandPercentage of 95%. The number of bands and the position of the bands are chosen uniformly at random.

In Experiment 1, we study the impact of bandPercentage and bandOccupancy. The results of this study, shown in Figure 2.6a, indicate that a high bandPercentage is favorable to our algorithm compared to bandOccupancy. In Experiment 2 we study the impact of nnzPercentage. The results of this study, shown in Figure 2.6b, indicate that a high nnzPercentage improves the performance of our algorithm. This can be understood from the highly parallel structure of the algorithm.

In Experiment 3, we consider three data layouts i.e, RANDOM,ROW-BLOCK and COLUMN-BLOCK for the non-band elements in matrix. In case of RANDOM, the non-band elements are spread uniformly at random in non-band part. In case of ROW-BLOCK and COLUMN-BLOCK, the non-band elements are filled in contiguous rows and columns respectively. Figure 2.6c shows the speedup as we vary the bandPercentage for a given data layout. As can be observed, our algorithm too suffers in the RANDOM distribution model but is faster compared to the library routine from cusparse once the bandPercentage crosses 85%. Further, the speedup on COLUMN-

BLOCK distribution is lower compared to that of ROW-BLOCK. The reason for this can be attributed to the fact that the library routine from cusparse does 5X times better on COLUMN-BLOCK compared to ROW-BLOCK as there is lesser chance of load imbalance in the COLUMN-BLOCK.



Figure 2.6: Studies on quasi-band matrices with varying bandPercentage, nnzPercentage and spatial distribution of non-band Elements.

2.4 Conclusion

We have demonstrated that significant performance improvement can be achieved on the GPU by designing input aware algorithms and data structures (Refer Sections 1.3.1 and 1.3.3 for more details), as opposed to designing input agnostic methods. For the sparse matrix multiplication operation with band and quasi-band sparsity patterns for the input, our input-aware approach achieved a speedup of 5x and 2x on average over a collection of band and quasi-band matrices respectively.

Chapter 3

Hines Matrix Solver

Sparse matrices and computations on sparse matrices arise in many areas of science and engineering such as computational fluid dynamics, computational neuroscience and molecular dynamics [22]. Prominent among the computations on sparse matrices include matrix vector multiplication, matrix multiplication, and solving a system of linear equations where the underlying matrix of coefficients is sparse. The importance of these computations can be gauged by the fact that these computations are included as dwarfs in the Berkeley report [7]. It is therefore not surprising that most modern libraries in the parallel setting include optimized routines for the above computations on sparse matrices [2, 3].

Several researchers have focused on improving the performance of sparse matrix computations on a variety of modern many- and multi-core architectures. Prominent examples include [84], [61] and [5]. Wangdong et al. [84] and Matam et al. [61] provide efficient algorithms for sparse matrix vector multiplication and sparse matrix matrix multiplication respectively on hybrid (CPU+GPU) architectures. Agullo et al. [5] optimize direct matrix solvers for Intel KNL architectures.

In recent years, one approach that is being used to improve the efficiency of sparse matrix computations on modern parallel architectures is to understand the strucure of sparsity of the matrix and its implications to parallel algorithm design and implementation. Examples of such instances are seen in the work of Ramamoorthy et al. [72] for multiplying two scale-free sparse matrices, Vooturi et al. [80] for multiplying two quasi-band sparse matrices, Buluc et al. [13] for multiplying two hyper-sparse matrices, and Wangdong et al. [85] for multiplying a quasi-band sparse matrix with a dense vector.

In this chapter, we investigate GPU algorithms for sparse matrix solver(Ax = B) operation that solves for x, where A is a Hines sparse matrix [35]. A Hines matrix is a symmetric matrix where the main diagonal is non zero and after each main diagonal entry, there is only one nonzero element. Solving a system of linear equations with the underlying matrix being a Hines matrix is denoted as HinesSolver in the rest of the chapter.

The rest of the chapter is organized as follows: In this section, we first motivate the problem, discuss related work, and list our key contributions. In Section 3.1, we describe how a Hines matrix is generated from the mathematical model, its structural properties and a general form. In Section 3.2, we describe a linear O(N) algorithm for HinesSolver using an Exact Domain Decomposition method (EDD) and also discuss how to tailor our algorithm on a GPU. Results and experiments are discussed in Section 3.3. Finally, we conclude and outline future work in Section 3.4.

Motivation: Neuron simulations [73] happen in a time-step manner and a HinesSolver is invoked in each time-step. Generally, researchers have to run these simulations for many time-steps to understand a particular behaviour. For example, a single neuron simulation running for a neuron time of one minute with 1 milli second time-step involves 60,000 HinesSolver instances. In a given time-step, the computation apart from a HinesSolver is parallel and is suitable for GPU. By having HinesSolver on a GPU, simulations can be made faster by making use of GPU hardware and avoiding costly memory transfers from GPU to CPU at each time-step.

In case of network simulations, which involve the study of the behavior of interconnected neurons, multiple Hines matrix systems have to be solved at each time-step. It is possible to map the computation of solving multiple Hines matrix systems into a single big Hines matrix system, which when solved gives solutions to the individual systems.

Further, researchers frequently run experiments which involves changing only a single parameter while keeping other parameters fixed. These types of experiments also result in solving multiple Hines matrices in each time-step. We discuss one such case in Section 3.3.4, where for a set of experiments the matrix remains the same, but the right hand side vectors vary in a given time-step.

Hence, parallelizing HinesSolver on a GPU can speedup neuron simulations and also enable researchers to perform rapid experimentation.

Related work: HinesSolver is studied in the sequential setting by Hines [34]. Hines [34] proposed a modified Gauss elimination algorithm whose runtime is O(N), where N is the number of rows in the Hines matrix. Hines also proposed parallel Gauss elimination algorithms for HinesSolver in [35] and [36]. To understand these algorithms, it is helpful to visualize a Hines matrix as a rooted tree with self-loops. These algorithms are based on the fact that suitably selected subtrees can be processed in parallel. However, this approach suffers from the following drawbacks. Firstly, there is a constraint on subtree division which limits the amount of parallelism available. Secondly, they are designed for optimizing network simulations on multi-core architec-

tures, where multiple Hines matrices of different sizes have to be solved and the ability to break a tree allows for efficient load balancing.

In HinesSolver, triangularizing the segments at the same level of a tree can be done in parallel. This idea was exploited by Roy et al. in [10] which is also based on Gauss elimination. One of the drawbacks of this approach is that the parallel time of the algorithm is bounded by the depth of the tree. Another drawback is that the amount of parallelism and computation at each level is dependent on the input and hence can introduce significant load imbalance.

Some of the above problems are solved by Mascagni [60], Larriba Pey [48] who introduced the Exact Domain Decomposition (EDD) technique to solve matrices corresponding to undirected graphs. EDD involves solving a domain matrix of size equal to the number of nodes with degree greater than two. In case of matrices corresponding to undirected graphs, the domain matrix does not exhibit any special properties and it was suggested to solve using any direct solver, such as Gauss elimination.

Contributions: In this work, we first start by proving that the domain matrix obtained via the exact domain decomposition method on a Hines matrix has the same structural properties as that of a Hines matrix. This result has three immediate benefits: (i) it allows us to apply the exact domain decomposition technique recursively, (ii) As the recursion bottoms out, the small size of the resulting domain matrix allows us to invoke a sequential HinesSolver [34] in much less time, and (iii) It allows us to introduce a decomposition strategy called fine decomposition which can be efficiently mapped onto a GPU. Using the above observations, we design an efficient parallel algorithm and its GPU implementation to solve a system of linear equations where the underlying matrix is a Hines matrix.

Our experimental results on an Nvidia Tesla K40c GPU over a variety of inputs indicate that our algorithmic approach R-FINE-TPT based on fine decomposition is 2.5x faster than the previously known approach. We also conduct experiments to study the effect of parameters such as amount of fineness in R-FINE-TPT, depth of recursion, compartment resolution and number of right hand sides in the matrix system to show the robustness of our approach.

Finally, we employ a machine learning technique called linear regression to find a threshold function which helps in deciding when to stop the recursion in our algorithm.

3.1 Hines Matrix

3.1.1 The Hodgkin-Huxley Model

The Hodgkin-Huxley model [38], revolutionized the understanding of how nerve cells generate electrical signals, known as action potentials. This mathematical model describes the behavior of the cell membrane in response to electrical stimuli in neurons. It is based on the concept of ion channels, specialized protein structures embedded in the cell membrane that control the flow of ions, such as sodium and potassium, across the membrane. The model incorporates equations that represent the dynamics of ion movement through these channels, accounting for factors like voltage and time. Through meticulous experimental observations and mathematical analysis, Hodgkin and Huxley elucidated the mechanisms underlying action potential generation and propagation, laying the foundation for modern neuroscience and computational neuroscience.

A non-linear differential equation [38] models how potential difference (V_m) changes with respect to ion-channels, current and other properties of a neuron. To simulate the model, a neuron is discretized spatially into multiple compartments as shown in Figure 3.1a. The relationships between various compartments in a compartmentalized neuron can then be represented as a rooted tree as shown in Figure 3.1b where each node in the tree corresponds to a compartment. A node V_i in the tree has a unique parent compartment V_p and child compartments as shown in Figure 3.1c. The tree is then numbered using a DFS numbering scheme from leaves to root. DFS numbering ensures two things.

- 1. The number of a node is larger than all its children and smaller than its parent.
- 2. The compartments in each branch of the neuron have consecutive numbers.

The current balance equation of the i^{th} compartment at the j^{th} timestep is then described according to Equation 3.1.

$$(V_i^j - V_i^{j-1}) \times \frac{C_i}{\Delta t} = (E_i - V_i^j) / Rm_i + (V_p^j - V_i^j) \times Ga_{i,p} + \sum_{k=IonChannels(i)} G_{i,k}^j \times (E_{i,k} - V_i^j) + Iext_i + \sum_{k=IonChannels(i)} (V_c^j - V_i^j) \times Ga_{i,c} \quad (3.1)$$

$$\sum_{c=children(V_i)} (V_c^j - V_i^j) \times Ga_{i,c}$$

where V_i^j and $G_{i,k}^j$ represent voltage and conductance of ion channel k respectively for compartment i at time step j. The constants $Iext_i$, C_i , E_i , Rm_i , $E_{i,k}$ correspond to external current, membrane capacitance, membrane resting potential, membrane resistance, and reverse potential of ion channel k respectively for compartment i. Ga_{t_1,t_2} represents the radial conductance between compartments t_1 and t_2 . Δt is the time interval between two time steps. The only unknowns in Equation 3.1 are V_i^j, V_p^j and $\{V_c^j | c \in children(V_i)\}$. By isolating them, Equation 3.1 can be written in a concise manner as shown in Equation 3.2. For more details refer [12].

$$A_{i,p}^{j}V_{p}^{j} + A_{i,i}^{j}V_{i}^{j} + \sum_{c=children(V_{i})} A_{i,c}^{j}V_{c}^{j} = b_{i}^{j}$$
(3.2)

where

$$A_{i,i}^{j} = \left(\frac{C_{i}}{\Delta t} + \frac{1}{Rm_{i}} + \sum_{r=neigh(i)} Ga_{i,r} + \sum_{k=IonChannels(i)} G_{i,k}^{j}\right)$$
$$A_{i,p}^{j} = Ga_{i,p}, \forall_{c \in children(V_{i})} A_{i,c}^{j} = Ga_{i,c}$$

$$b_i^j = \left(Iext_i + \frac{E_i}{Rm_i} + V_i^{j-1} \times \frac{C_i}{\Delta t} + \sum_{k=IonChannels(i)} G_{i,k}^j \times E_k\right)$$

 $V_i^j V_p^j$ and $\{V_c^j | c \in children(V_i)\}$ are the voltages of i^{th} compartment, parent compartment of V_i and child compartments of V_i respectively at j^{th} time step and $A_{i,i}^j, A_{i,p}^j$ and $\{A_{i,c}^j | c \in children(V_i)\}$ are the corresponding coefficients. The current balance equations of all compartments can then be represented in a matrix form $A\vec{x} = \vec{b}$, where $\vec{x} = [V_1^j \cdots V_N^j]^T$ and $\vec{b} = [b_1 \cdots b_N]^T$. Solving this linear system gives the voltage values for compartments in each time-step. Figure 3.2 corresponds to the structure of the matrix formed by the current balance equations of compartmentalized neuron in Figure 3.1b. The matrices obtained from the voltage PDE simulations fall under a class of matrices called Hines matrices. A Hines matrix has the following structural properties:

- 1. The matrix is symmetric. $[A_{ij} = A_{ji}]$
- 2. In each row *i*, there exists only one nonzero element with column index *j* such that j > i. $[\exists!j|(A_{i,j} \neq 0 \text{ and } j > i)]$

From Equation 3.2, we can see that the off-diagonal elements of A have contributions only from the radial conductance $Ga_{t1,t2}$. As the radial conductance between any two compartments t_1 and



Figure 3.1: Multi compartment neuron modelling.

 t_2 is the same irrespective of order i.e, $Ga_{t_1,t_2} = Ga_{t_2,t_1}$, the matrix A is symmetric. The only non-zero element in row *i* after $A_{i,i}$ corresponds to the coefficient of parent compartment of V_i .

3.1.2 A General Form

A Hines matrix A can be represented in a general form along with some conditions. Let R be the set of compartments with more than one children and junction set J be a superset of R. Dividing the matrix at rows J and columns J results in a grid G of dimensions $(S + 1) \times (S + 1)$, where S = |J|. Each main diagonal entry of G is a block diagonal matrix Tr_i with k_i blocks, with each block being a symmetric tridiagonal matrix. A non main-diagonal entry of G is a zero matrix O. A Hines matrix A can then be represented in the form of Equation 3.3. We however note that not all matrices which can be represented in the form of Equation 3.3 are Hines matrices. In Section 3.1.3, we describe the conditions that the general form should satisfy for it to represent only Hines matrices. The notation used for describing general form is described in Table 3.1.

$A\vec{x} = \vec{b}$	Matrix system				
$Tr_i(Tr_i^1Tr_i^{k_i})$	Block diagonal matrix with k_i blocks				
Tr_i^j	Symmetric Tridiagonal matrix				
$\overrightarrow{C_{ij}}(\overrightarrow{C_{ij}^{1}}, \overrightarrow{C_{ij}^{k_{i}}})$	Column vector $\overrightarrow{C_{ij}}$ split into k_i vectors.				
J, J_i	Junction array J and i^{th} junction.				
$\vec{x_i}$	Vector $\vec{x}[J_{i-1}+1:J_i-1]$				
$\vec{b_i}$	Vector $\vec{b}[J_{i-1}+1:J_i-1]$				
x_i, b_i	$\vec{x}[i], \vec{b}[i]$				
$A_{i,j}$	A[i][j]				
Parent[i]	Column index of the only non-zero entry				
	after $A_{i,i}$				
JunctionIndex[k]	Index of junction k in junction array J				

Table 3.1: Notation used in general form, algorithms and proof.

$$A = \begin{bmatrix} Tr_{1} & \vec{C}_{1,1} & 0 & \cdots & \vec{C}_{1,S} & 0 \\ \vec{C}_{1,1}^{T} & A_{J_{1},J_{1}} & \vec{C}_{2,1}^{T} & \cdots & A_{J_{1},J_{S}} & \vec{C}_{(S+1),1}^{T} \\ 0 & \vec{C}_{2,1} & Tr_{2} & \cdots & \vec{C}_{2,S} & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vec{C}_{1,S}^{T} & A_{J_{S},J_{1}} & \vec{C}_{2,S}^{T} & \cdots & A_{J_{S},J_{S}} & \vec{C}_{S+1,S}^{T} \\ 0 & \vec{C}_{S+1,1} & 0 & \cdots & \vec{C}_{S+1,S} & Tr_{S+1} \end{bmatrix}$$

$$Tr_{i} = \begin{bmatrix} Tr_{i}^{1} & & & \\ & Tr_{i}^{2} & & \\ & & & Tr_{i}^{k} \end{bmatrix} \quad \vec{C}_{i,j} = \begin{bmatrix} \vec{C}_{i,j}^{1} \\ \vec{C}_{i,j}^{2} \\ \vdots \\ \vec{C}_{k,i}^{k} \end{bmatrix}$$

$$\vec{x} = \begin{bmatrix} \vec{x}_{1} \\ x_{J_{1}} \\ \vec{x}_{2} \\ \vdots \\ x_{J_{S}} \\ \vec{x}_{S+1} \end{bmatrix} \quad \vec{b} = \begin{bmatrix} \vec{b}_{1} \\ b_{J_{1}} \\ \vec{b}_{2} \\ \vdots \\ b_{J_{S}} \\ \vec{b}_{S+1} \end{bmatrix} \quad [\vec{x}_{i}, \vec{b}_{i}] = \begin{bmatrix} \vec{x}_{i}^{1}, \vec{b}_{i}^{1} \\ \vec{x}_{i}^{2}, \vec{b}_{i}^{2} \\ \vdots \\ \vec{x}_{k}^{k}, \vec{b}_{k}^{k} \end{bmatrix}$$

$$(3.5)$$



Figure 3.2: The general form of Hines matrix corresponding to compartmentalized neuron in Figure 3.1b with J = [10,14].

3.1.3 Necessary Conditions

Condition 1: For a given Tr_i^r , where $1 \le i \le S + 1$ and $1 \le r \le k_i$, only one column vector among $\{\overrightarrow{C_{ij}^r} | i \le j \le S\}$ is non-zero with only one non-zero element at the end.

Each Tr_i^r is bounded by two rows *startRow* and *endRow* in matrix A. As Tr_i^r is a tridiagonal matrix, the matrix element to the right of each main diagonal element in Tr_i^r is in itself except for endRow. From general form, we know that Parent[endRow] has to be in junction set J. As there is only one non-zero after each main diagonal element in a Hines matrix, the matrix element to the right of $A_{endRow,endRow}$ has to be in one of the column vectors $\{\overrightarrow{C_{ij}^r}|i \leq j \leq S\}$. As the non-zero column vector is also bounded by startRow and endRow, only the last element of that vector is non-zero. For the matrix in Figure 3.2, we can see that each tridiagonal matrix $(Tr_1^1, Tr_1^2, Tr_1^3, Tr_2^1)$ has only one nonzero column vector $(\overrightarrow{C_{1,2}^1}, \overrightarrow{C_{1,1}^2}, \overrightarrow{C_{1,1}^3}, \overrightarrow{C_{2,2}^1})$ to its right.

Condition 2: For a given junction J_i , $\overrightarrow{C_{i,i}^{k_i}} \neq \vec{0}$.

Each tridiaonal matrix Tr_i^r corresponds to an unbranched segment in the tree. For a junction node J_i , $Tr_i^{k_i}$ corresponds to the segment which is numbered just before numbering junction node J_i . From a DFS numbering scheme, we can say that $J_i=Parent[endRow(\overrightarrow{Tr_i^{k_i}})]$. So, $\overrightarrow{C_{i,i}^{k_i}}$ becomes a non-zero vector. For the matrix in Figure 3.2, both the vectors corresponding to junctions i.e, $\overrightarrow{C_{1,1}^3}$ and $\overrightarrow{C_{2,2}^1}$ are non-zero.

Condition 3: If $Parent[J_i] \in J$, then $(\overrightarrow{C_{i+1,i}})^T = \overrightarrow{0}$, else $(\overrightarrow{C_{i+1,i}})^T \neq \overrightarrow{0}$ and has exactly one non-zero at the first index.

If $Parent[J_i] \notin J$, then according to DFS numbering system $Parent[J_i] = J_i + 1$. This means $A_{J_i,(J_i+1)} \neq 0$. As $A_{J_i,(J_i+1)}$ is the first element of $(\overrightarrow{C_{i+1,i}})^T$, $(\overrightarrow{C_{i+1,i}})^T[1] = A_{J_i,(J_i+1)}$. As there can be only one non-zero element to the right of A_{J_i,J_i} , all the elements of $(\overrightarrow{C_{i+1,i}})^T$ are zero except for the first one.

If $Parent[J_i] \in J$, then the row and column of the only non-zero element to the right of A_{J_i,J_i} belong to J. This means all the row vectors to the right of A_{J_i,J_i} are zero vectors which includes $(\overrightarrow{C_{i+1,i}})^T$.

For matrix in Figure 3.2, both $Parent[J_1]$ and $Parent[J_2]$ are not in J. Hence both $\overrightarrow{C_{2,1}^1}^T$ and $\overrightarrow{C_{3,2}^1}^T$ are non-zero.

Condition 4: For a given junction row J_i , all row vectors after $(\overrightarrow{C_{i+1,i}^1})^T$ are zero vectors.

Only one non-zero element exists after each main diagonal element in Hines matrix. From the previous condition, we know that for a junction row J_i , it may only be part of row vector $(\overrightarrow{C_{i+1,i}^1})^T$. So the rest of the row vectors after $(\overrightarrow{C_{i+1,i}^1})^T$ are zero vectors.

3.2 Our Approach

3.2.1 EDD on an Undirected Graph

The Exact Domain Decomposition (EDD) method was first employed by Mascagni [60] to solve matrices corresponding to undirected graphs. The main idea is to create subdomains by breaking the graph at nodes with degree greater than two. In such a decomposition, each subdomain corresponds to a chain graph and the matrix of the subdomain corresponds to a tridiagonal matrix. These subdomains are solved independently and the solutions are fused together based on subdomain relationships to construct the final solution. Any undirected graph can be represented in general form described in Equation 3.3. Thus the EDD algorithm for a matrix that can be represented in general form can be described in Algorithm 10.

Algorithm 10 Exact Domain decomposition method for a matrix in general form corresponding to an undirected graph.

1: $\forall_{i=1:S+1} \overrightarrow{R_i} = Tr_i \overrightarrow{b}_i$	Solve independent tridiagonal systems
2: $\forall_{i=1:S+1} \forall_{j=1:S} P_{i,j} = Tr_i C_{i,j}$	▷ Solve independent tridiagonal systems
3: $\forall_{i=1:S} \forall_{j=1:S}$	
4: $M[i][j] = (\sum_{l=1}^{l=S+1} \overrightarrow{C_{l,i}}^{I} \times \overrightarrow{P_{l,j}}) - A_{J_i,J_j}$	\triangleright Generate domain matrix M
5: $\forall_{i=1:S} M_{rhs}[i] = (\sum_{l=1}^{l=S+1} \overrightarrow{C_{l,i}}^{I} \times \overrightarrow{R_{l}}) - b_{J_{i}}$	
6: $M_x = M M_{rhs}$	▷ Solved domain matrix
7: $\forall_{i=1:S+1} \overrightarrow{x_i} = \overrightarrow{R_i} - \sum_{k=1}^{k=S} M_x[k] \times \overrightarrow{P_{i,k}}$	▷ Construct final solution

Algorithm 10 has following four stages:

- 1) Solve tridiagonal systems.
- 2) Construct the the domain matrix M.
- 3) Find solutions at junctions $(M|M_{rhs})$.
- 4) Construct the final solution (\vec{x}) .

In stage-1, each tridiagonal matrix Tr_i is solved with multiple right hand sides $\forall_{1 \leq j \leq S} \overrightarrow{C_{i,j}}$ and $\overrightarrow{b_i}$. A tridiagonal system can be solved in linear time in the size of the matrix, and the sum of all tridiagonal matrices Tr_1 to Tr_{S+1} is (N-S), so the time complexity for stage-1 is $O(N*S-S^2)$. In stage-2, the domain matrix M and its right hand side M_{rhs} are constructed using rows at junction indices of matrix A and the tridiagonal solutions computed in stage-1. Each entry in M and M_{rhs} takes O(N) time and as there are S^2 elements in M, the time complexity for stage-2 is $O(N*S^2)$. In stage-3, the domain matrix M is solved with M_{rhs} to compute M_x (junction node solutions) using Gauss elimination. Thus the time complexity for stage-3 is $O(S^3)$. In stage-4, the solution for non junction nodes are computed using M_x and tridiagonal solutions in stage-1. The computational pattern is similar to stage-1 and thus has the time complexity of $O(N*S-S^2)$.

3.2.2 Domain Matrix of Hines Matrix

In this section, we prove that when EDD is applied on a Hines matrix, the domain matrix M is again a Hines matrix.

Theorem 3. The domain matrix M when EDD is applied on a Hines matrix is a Hines matrix.

Proof. We prove it by showing that the domain matrix M satisfies the structural properties of a Hines matrix as described in Section 3.1.

(i) The matrix is symmetric, $M_{i,j} = M_{j,i}$.

From Algorithm 10, we know that any element $M_{i,j}$ of the domain matrix M can be constructed as follows:

$$M_{i,j} = \left(\sum_{l=1}^{S+1} \overrightarrow{C_{l,i}}^T \times (Tr_l)^{-1} \times \overrightarrow{C_{l,j}}\right) - A_{J_i,J_j}$$

and

$$M_{j,i} = \left(\sum_{l=1}^{S+1} \overrightarrow{C_{l,j}}^T \times (Tr_l)^{-1} \times \overrightarrow{C_{l,i}}\right) - A_{J_j,J_j}$$

If *R* is a symmetric matrix of size $N \times N$ and *p*, *q* are column vectors of size *N*, then $p^T \times R \times q$ = $q^T \times R \times p$. So for a valid *l*, *i*, and *j*, $(\overrightarrow{C_{l,i}}^T \times (Tr_l)^{-1} \times \overrightarrow{C_{l,j}}) = (\overrightarrow{C_{l,j}}^T \times (Tr_l)^{-1} \times \overrightarrow{C_{l,i}})$. As a Hines matrix is a symmetric matrix, $A_{J_i,J_j} = A_{J_j,J_i}$. Hence the matrix *M* is symmetric.

(ii) In each row *i*, there exists only one nonzero element with column index *j* such that j > i. $[\exists! j | (M_{i,j} \neq 0 \text{ and } j > i)]$

For a given junction row J_i , the non-zero row vectors before and after $A_{J_iJ_i}$ can be divided into two sets S_{left} and S_{right} respectively. Because a Hines matrix is symmetric, S_{left} can contribute only to the main diagonal element $M_{i,i}$ of the domain matrix M. From the Conditions 3 and 4, we know that $S_{right} = \{\overrightarrow{C_{i+1,i}^1}\}$ or \emptyset . Each element in row i of M after main diagonal element $M_{i,i}$, j > i, can then be represented in the following cases.

Case 1:
$$S_{right} = \{ \overrightarrow{C_{i+1,i}^1}^T \}$$

In this case, we can see that:

$$M_{i,j} = \overrightarrow{C_{i+1,i}^{1}}^{T} \times (Tr_{i+1}^{1})^{-1} \times \overrightarrow{C_{i+1,j}^{1}}$$

From condition 1, we know that for Tr_{i+1}^1 , there exists only one non-zero column vector to its right i.e, $\exists!j|(\overrightarrow{C_{i+1,j}^1} \neq \overrightarrow{0} \& j > i)$.

Case 2: $S_{right} = \emptyset$

We have that $M_{i,j} = -A_{J_i,J_j}$. From the conditions 3 and 4, we know that this can happen only if $Parent[J_i] \in J$. As there is only one non-zero element after A_{J_i,J_i} , only one of the elements from set $\{A_{J_i,J_{i+1}}, A_{J_i,J_{i+2}}, \dots, A_{J_i,J_S}\}$ is non-zero.

In both cases, for a given row i in the domain matrix M, there exists only one j such that j > iand $M_{i,j} \neq 0$.

3.2.3 An O(N) Linear Algorithm for HinesSolver

A Matrix A that can be expressed using the general form (Section 3.1.2) can be solved using Algorithm 10. Hines matrices are only a subset of matrices that can be expressed in general form and using Algorithm 10 for HinesSolverleads to $O(N * S + S^3)$ time complexity. Because Hines matrix has extra conditions (Section 3.1.3) on top of the general form, there is lot of zero compute is Algorithm 10. For example, because domain matrix of a Hines matrix is also a Hines matrix 3.2.2, there is no need to compute all S^2 elements of S in stage 3, as only 3 * S - 2 elements are non-zero in a Hines matrix. Similarly, there is zero compute in other stages due to zero vectors in the Hines matrix. We remove such zero compute from Algorithm 10 and express in the form of Algorithm 11, which is a linear time O(N) algorithm.

Analysis: In Steps 3,6,7, and 10 of Algorithm 11, independent tridiagonal systems are being solved. The cumulative size of the tridiagonal systems from Step-3 is bounded by N and that of Steps 6,7, and 10 is bounded by 2N. So Stage-1 involves solving tridiagonal systems with cumulative size bounded by 3N. As a tridiagonal system can be solved in linear time, the complexity of Stage-1 is O(N). The complexity for computing both the main diagonal of M and right hand side M_{rhs} is $O(\sum_{i=1}^{i=S} n_i)$, where n_i is the number of neighbours of compartment J_i . It can be seen from Step-16 that the complexity for computing non-zero off diagonal element of M is O(1). As $(\sum_{i=1}^{i=S} n_i < 2N)$ and the domain matrix M has only 2(S - 1) non-zero off diagonal elements,

Algorithm 11 O(N) algorithm for HinesSolver using EDD.

1: #Stage-1 Solve tridiagonal systems. 2: for i=1:S do $Q_i = Tr_{i+1}^1 | \overrightarrow{C_{i+1,i}^1}|$ 3: for $r=1:k_i$ do 4: 5: $col = JunctionIndex[Parent[endRow(Tr_i^r)]]$ $P_i^r = Tr_i^r |\overrightarrow{C_{i,col}^r}]$ 6: $R_i^r = Tr_i^r |\vec{b_i^r}|$ 7: end for 8: 9: end for 10: $R_{S+1}^1 = Tr_{S+1}^1 | \overrightarrow{b_{S+1}^1} /$ End case 11: 12: #Stage-2 Constructing the domain matrix. 13: for i=1:S do $M[i][i] += (C_{i+1,i}^{1}[1] \cdot Q_{i}[1] - A_{J_{i},J_{i}})$ 14: $col = JunctionIndex[Parent[endRow(Tr_{i+1}^1)]]$ 15: $M[i][col] = (C_{i+1,i}^{1}[1] \cdot P_{i+1}^{1}[1] - A_{J_{i},J_{col}})$ $M_{rhs}[i] += (C_{i+1,i}^{1}[1] \cdot R_{i+1}^{1}[1] - b_{J_{i}})$ 16: 17: for $r=1:k_i$ do 18: $col = JunctionIndex[Parent[endRow(Tr_i^r)]]$ 19:
$$\begin{split} M[col][col] + = \overrightarrow{C_{i,col}^{r}} [|\overrightarrow{C_{i,col}^{r}}|] \cdot P_{i}^{r}[|\overrightarrow{C_{i,col}^{r}}|] \\ M_{rhs}[col] + = \overrightarrow{C_{i,col}^{r}} [|\overrightarrow{C_{i,col}^{r}}|] \cdot R_{i}^{r}[|\overrightarrow{C_{i,col}^{r}}|] \end{split}$$
20: 21: end for 22: 23: end for 24: 25: #Stage-3 Find solutions at junctions 26: $M_x = M | M_{rhs}$ 27: 28: #Stage-4 Construct \vec{x} . 29: for i=1:S do $x_{i+1}^1 \rightarrow M_x[i] \times Q_i$ 30: for $r=1:k_i$ do 31: $col = JunctionIndex[Parent[endRow(Tr_i^r)]]$ 32: $\vec{x_i^r} \stackrel{-=}{\to} M_x[col] \times P_i^r$ $\vec{x_i^r} \stackrel{+=}{\to} R_i^r$ 33: 34: 35: end for 36: end for

the complexity of Stage-2 is O(N + S). As a Hines matrix can be solved in linear time [34], the complexity for solving domain matrix in Stage-3 is O(S). Stage-4 has the same loop structure as that of Stage-1. In places where a tri-diagonal system is solved, vector scaling and addition is performed. Hence the complexity of Stage-4 is O(N). The complexity of Stages 1-4 are O(N), O(N + S), O(S) and O(N) respectively. As the number of junctions S cannot exceed N, the complexity of our algorithm is O(N).

3.2.4 EDD for HinesSolver on GPU

In this section, we show how HinesSolver can be efficiently mapped onto a GPU architecture using EDD. In Section 3.2.3, we showed that the complexity of our algorithm is O(N), irrespective of the number of junction compartments. We use this fact and propose a decomposition strategy called fine decomposition.

Minimal Decomposition (J_{md}) : Compartments with more than one children are chosen as junctions.

Fine Decomposition (J_{fd}) : The goal of this decomposition is to have equal size tridiagonal systems to solve in Stage-1. In order to achieve that, we break each branch of the tree into K sized chains and include last compartment of each chain in junction set J_{fd} . Apart from these compartments, J_{fd} includes all compartments which have more than one children i.e, junction set J_{md} .

An example of minimal decomposition and fine decomposition with K = 4 for a given tree are shown in Figures 3.3a and 3.3b respectively. Hollow nodes in Figures 3.3a and 3.3b correspond to junction compartments.

Algorithm 12 describes the recursive algorithm for HinesSolver using the Exact Domain Decomposition method.

Stage-1: In this stage, the computation involves solving many tridiagonal systems. This computation can be performed using two approaches **TRSV** and **TPT**.

TRSV: Making all junction rows and junction columns of matrix A zero, except for main diagonal elements results in a tridiagonal matrix T. Figure 3.5 shows the tridiagonal matrix T obtained from the Hines matrix shown in Figure 3.2. From Stage-1 of Algorithm 11, we know that each tridiagonal Tr_i^r has to be solved with at least two and at most three right hand sides. We position them accordingly as shown in Figure 3.2 and use an optimized tridiagonal solver from NVIDIA's CUDA library CuSparse [2] to solve tridiagonal matrix T with three right hand sides.

TPT: In this approach, we map each thread to solve an independent tridiagonal system. In minimal decomposition, independent tridiagonal systems that have to be solved in Stage-1 are big and have



Figure 3.3: Decomposition strategies.



Figure 3.4: Recursive application of fine decomposition with K=3



Figure 3.5: Mapping Stage-1 computation to tridiagonal solver(TRSV) with three right hand sides in MIN-TRSV.

lot of variance in size. So using TPT approach for solving tridiagonal systems suffers from less parallelism, higher load imbalance and more amount of work per thread. TRSV approach hides these to some extent and takes advantage of optimized library function for tridiagonal solver. This compatibility leads to MIN-TRSV approach.

In fine decomposition, each independent tridiagonal system in Stage-1 is almost of same size and has equal compute. So using the TPT approach, one can take advantage of the SIMD architecture of a GPU. This compatibility leads to R-FINE-TPT approach.

Stage-2: In this stage, we construct the domain matrix system (M, M_{rhs}) . As there is no dependency among the non-zero elements of the domain matrix system, they can be constructed in parallel.

Stage-3: In this stage, we have a choice to run the algorithm recursively to solve domain matrix system $(M|M_{rhs})$. In the R-FINE-TPT approach, we run the algorithm recursively and stop when using a GPU is no longer efficient. In case of MIN-TRSV, we do not run the algorithm recursively. The reason is that the tree corresponding to domain matrix when minimal decomposition is applied has no vertices of degree two. Applying any decomposition recursively on that tree will not reduce

the size of domain matrix size significantly. So in the MIN-TRSV approach, the runtime is less when we do not recurse.

Stage-4: Stage-4 involves constructing final solution \vec{x} . As there is no dependency among elements of \vec{x} , each element of \vec{x} can be constructed in parallel.

Algorithm 12 Recursive algorithm for HinesSolver using Exact Domain Decomposition method.

```
R-EDD(A, \vec{b}, \text{decomposition, num-rhs})
if decomposition == MIN then
    Use minimal decomposition with TRSV approach to solve tridiagonal systems in Stage-1.
else if decomposition == FINE then
    Use fine decomposition with TPT approach to solve tridiagonal systems in Stage-1.
end if
Stage2: Construct domain matrix system (M, M_{rhs})
if decomposition == FINE and
Threshold(rows(M),num-rhs) == False then
    M_x = \mathbf{R} - \mathbf{EDD}(M, M_{rhs}, \text{decomposition, num-rhs})
else
    Transfer (M, M_{rhs}) to CPU.
    Solve Domain system M_x = M/M_{rhs} on CPU.
    Transfer M_x to GPU.
end if
Stage-4: Construct \vec{x}
return \vec{x}
}
```

3.2.5 Implementation Details

A Hines matrix A of size N is stored using two arrays, parent array P of size N and data array D of size 2N. P[i] stores the column index of the only nonzero after A[i][i]. As A is symmetric, we store only the nonzero values of upper triangular matrix in a row major fashion in D. By storing in the row major fashion, we get better memory coalescing while accessing a tridiagonal matrix Tr_i^j in Stage-1. We used buffer arrays to avoid replication of tridiagonal matrices in the computations of Stage-1. We employed NVIDIA's CUDA Occupancy Calculator tool to configure thread block sizes in CUDA kernels.

3.3 Results and Analysis

3.3.1 Platform

We use an Nvidia Tesla K40c GPU for all our experiments. It is mounted on an Intel i7-4790K CPU with 32GB RAM. The K40c has a total of 2880 cores organized in 15 SMx, with each core clocked at 745 MHz. It provides a peak double precision floating point performance of 1.43 Tflops and single precision floating point performance of 4.29 TFlops. Each SM also has a 64KB configurable cache to exploit data locality.

3.3.2 Dataset

All input Hines matrices come from neuron morphologies taken from www.neuromorpho.org [8]. We choose our dataset in such a way that they come from different parts of brain and has variation in size and the number of junctions. Some details of the chosen morphologies are shown in Table 3.2. We group the dataset into three categories: small (7K-11K), medium (29K-35K), and large (80K-120K) neurons based on size of the matrix.

3.3.3 Results

We compare our R-FINE-TPT approach with the MIN-TRSV approach which is based on minimal decomposition strategy suggested by Mascagni in [60]. These approaches differ in the decomposition strategy used for finding junctions and the computation strategy used to solve tridiagonal systems in Stage-1. All operations are carried out in double precision. From the results in Figure 3.6, it can be observed that R-FINE-TPT is faster than MIN-TRSV for all classes of input. It has to be noted that we achieve a speedup of 2x on EC5 neuron, which is the neuron with highest number of compartments than any other neuron in the repository [8].

The reason for the good performance of R-FINE-TPT over MIN-TRSV is that in R-FINE-TPT there is more parallelism, less work per thread, and negligible load imbalance. Whereas in case of MIN-TRSV, threads have to coordinate among themselves to solve one big tridiagonal system with three right hand sides. This requirement for coordination results in poor performance when compared to R-FINE-TPT.



Figure 3.6: Results on the input dataset. Our approach (R-FINE-TPT) achieves 2.5x speedup on average over MIN-TRSV approach.

3.3.4 Further Experiments

In this section, we perform two sets of experiments. One set of experiments study the impact parameters such as K in the fine decomposition, depth D in the recursion, and varying the number of right hand sides on the runtime of the algorithm. The second set of experiments are aimed at coming up with guidelines to choose appropriate values for K and D automatically based on empirical data. For some experiments we use linear neuron as our model. In linear neuron, there is only one branch and all the compartments have only one child.

Varying K in Fine Decomposition: In this experiment, we study how varying K in fine decomposition affects overall runtime and runtime of individual stages in R-FINE-TPT. To understand the behaviour, we take a linear neuron of size 100K as the input. For a linear neuron with N compartments, there are roughly N/K junctions and 3N/K tridiagonal systems of size (K-1) to be solved in Stage-1. As we increase K, number of threads to be launched i.e, 3N/K decreases in Stage-1 and work per thread i.e., solving tridiagonal of size (K-1) increases. Having few threads with more work is not good for GPU and it can be observed in run time of Stage-1 in Figure 3.7. As K increases, running time of Stage-1 increases. Stage-3 of EDD involves time for recursive call T(N/K). It decreases with increase in K and it can be observed in Figure 3.7. Threads launched in Stage-2 and Stage-4 are very light and changing K has little impact on their runtime. The overall runtime decreases to a certain K and then increases. For our input linear neuron of size 100K the best performance is at K = 3.

Neuron	Compartments	Junctions	Branches
EC5-609202-2	123248	1321	2259
Rat-ngf-11-11-04	114193	468	941
Alvarez-Control-Cell-1	84254	185	370
skinA12-18-2r	82110	3287	6183
L395-LCN	35630	787	1554
MA349-dSAC	35190	1838	3483
HICAP3	29159	119	242
14-traced	11193	546	1042
alphaMN4	9231	69	151
cell-36-trace	7881	97	184

Table 3.2: Details of neuron morphologies



Figure 3.7: Impact of varying K on R-FINE-TPT approach.



Figure 3.8: Impact of varying recursion depth D on speedup.



Figure 3.9: Impact of varying resolution on R-FINE-TPT and MIN-TRSV approaches.



Figure 3.10: Impact of varying right hand sides on speedup.

Varying the Depth (D) of Recursion: In this experiment, we study the effect of recursion depth D on the R-FINE-TPT approach. From Figure 3.8, we can see that as we increase D, the speedup increases to a certain point and decreases from then on. This is due to the fact that at inflexion point it is better to solve the matrix on a CPU rather than running the algorithm recursively. For large neurons, the inflection point is at D = 3 and the average size of the domain matrix at D = 3 is 1800. For such small matrices, it is faster to solve it on a CPU despite the cost of memory transfers. As long as we have bigger matrices to solve at each level, it is beneficial to run the algorithm recursively.

Varying Resolution: In compartmental modelling, each branch of the neuron is divided into multiple compartments. More accurate simulations are possible by increasing the number of compartments into which a branch is divided. The morphology file contains a particular compartmentalization of a neuron. In this experiment, we obtain a *P*-resolution morphology by breaking an original compartment in the morphology into *P* compartments. If the input morphology has *N* compartments, *P*-resolution morphology contains $P \times N$ compartments. From Figure 3.9, we can see that R-FINE-TPT performs better than MIN-TRSV for all resolutions. The primary reason for this is that using fine decomposition enables us to have computation in Stage-1 divided in to many threads with very little work. This coupled with recursion is the reason for better performance compared to MIN-TRSV.

Varying Right Hand Sides: Voltage behaviour studies have a lot of parameters to tinker with. For example, in Equation 3.2, having different values of external compartment current $(Iext_i)$ effects only right hand side of the matrix system. Now, it is possible to do multiple simulations for different values of $Iext_i$ at once. This is advantageous because it suffices to factorize the



Figure 3.11: Threshold function for matrix with multiple right hand sides



Figure 3.12: Best value of K in Fine Decomposition

tridiagonal matrices in Stage-1 only once and use the factorizations for all right hand sides. In this experiment, we see how R-FINE-TPT behaves with change in the number of right hand sides. From Figure 3.10, it can be seen that speedup increases with respect to number of right hand sides for all classes of neurons.

Determining the Threshold Function: In this experiment, we find a boolean threshold function for deciding when to stop the recursion in Algorithm 12. The threshold function depends on two parameters: the size of the matrix, N, and the number of right hand sides, R. We have to break the recursion at a stage where using CPU is better than using GPU. So, we run an experiment to find out the largest matrix size at which CPU is better than GPU for each value of R. From Figure 3.11, we can see that the data is following 1/x behaviour. Hence we modeled the function as $(N = a_0/R + a_1)$ and used linear regression to find constants a_0 and a_1 at which the error is minimum. Threshold function thus obtained from the above technique is $(N - (5245/R) - 40 \le 0)$. The actual value in case of R = 1 is 3500 and it is recommended to use threshold function $(N \le 3500)$ when using R = 1.

Choosing K in Fine Decomposition: In this experiment, we provide insights for choosing best value for K in fine decomposition when R = 1. The choice of K depends on the size of the matrix N and the K values chosen for matrices less than size N. For linear neurons of size N > 3500, we ran our algorithm for different values of K and chose the K which gave the best runtime. We find best K values for all matrix sizes constructively. Hence, in the lower levels of recursion, we use the computed best K values. From Figure 3.12, we can see that the variance of K is high for small values of N. For larger values of N, where recursion depth is greater than one, the best value of K remains constant at three. One interesting thing to observe is that for neurons
around size 20000, it is possible to go down two levels in the recursion but the best K value is the one that recurses only once. To get good performance, maintain a look up table for smaller matrices and use K = 3 for larger matrices.

3.4 Conclusion

In scientific simulations based on ordinary and partial differential equations, matrix solvers are almost always a bottleneck and making them faster reduces simulation time considerably. In this work, we have demonstrated that embracing the semantics of matrices into parallel algorithm design helps in designing efficient parallel solvers. The general form given for Hines matrices in this work provides a framework for proving more results on Hines matrices.

PART-II

Artificial intelligence is on the rise with functional tasks (image captioning, speech recognition, language translation, playing the game of Go, etc.) in areas like computer vision, speech, and language processing, etc. reaching or surpassing human-level accuracy. This progress can be largely attributed to computational approaches based on artificial neural networks (ANN) and hardware like GPU (Graphics Processing Unit). Based on the degree of connectivity between the neurons, a neural network can be classified into two types: Dense Neural Network (DNN) and Sparse Neural Network (SNN). The effectiveness of a neural network can be characterized using four attributes: accuracy, memory, compute, and runtime. For any given trained dense neural network, a sparse neural network can be generated that achieves the same accuracy, but takes less memory and less compute. But there is no guarantee that the generated sparse neural network leads to lower runtime than the dense neural network on hardware like GPU. This seems counter intuitive. Because in theory, less compute should translate to less runtime. But in reality, it is not always possible because the efficiency with which a GPU can process the underlying computational tasks in a sparse neural network is much less than processing computational tasks in a dense neural network.

Runtime of a sparse neural network on a GPU depends on how the neurons are connected with each other or simply put, the sparsity pattern. If the sparsity pattern is unstructured, then it results in irregularity in compute and memory access patterns on the GPU, which leads to lower efficiency for the underlying computational tasks in a sparse neural network. On the other hand, if the sparsity pattern is highly structured, then there is less irregularity, which leads to increased efficiency and decreased runtime. But imposing the structure comes at the cost of decreased sparsity for a given accuracy, and decreased accuracy for a given sparsity.

In Part-2, we propose solutions for generating sparse neural networks that are efficient in both accuracy and runtime on the GPU. First, in Chapter 4, we provide the necessary background for sparse neural networks in the context of the GPU hardware. In Chapter 5, we propose a dense to sparse training approach that generates sparse neural networks with block sparsity pattern. In Chapter 6 and 7, we propose a fine-tuning approach to generate sparse neural networks with hierarchical block and regularized multi block sparsity patterns. In Chapter 8, we propose a static sparse to sparse training approach to generate sparse neural networks with Recursive Cloned Uniform Block sparsity pattern generated using graph products of Ramanujan graphs.

Chapter 4

Sparse Neural Networks on GPU

4.1 Artificial Neural Network Primer

Artificial neural networks (ANN) can be of many types and each type of ANN is used for certain class of functional tasks. Convolutional neural networks, for example are used for performing image related functional tasks like image recognition, image segmentation etc. Recurrent neural networks for example, are used for performing functional tasks like language translation etc. Let us now understand what an artificial neural network is and how it is trained using the help of a simple feed forward neural network (FFN). An FFN can be expressed as a series of layers $L_1 \dots L_N$. Layer L_1 takes the input I, and L_N produces the output O. Each layer L_l has N_l neurons on the left and N_{l+1} neurons on the right. In a dense FFN, all the neurons on the left are connected to all the neurons on the right leading to $N_{l+1} \times N_l$ number of connections. Each connection is associated with a parameter, and for a layer L_l , these parameters can be arranged in the form of a matrix W_l of size (N_{l+1}, N_l) , and these parameters are called weight parameters. In layer L_l , along with weight parameters, there is another set of parameters called bias parameters b_l of size N_{l+1} . Each layer takes in an input matrix X_l of size (N_l, B) and produces an output matrix X_{l+1} of size (N_{l+1}, B) , where an entry in X_{l+1} is calculated using Equation 4.1. In Equation 4.1, ActivationFunction is a non-linear function like sigmoid, ReLU etc. The computation required for computing X_{l+1} matrix can be expressed in the form of matrix multiplication using 4.2.

$$X_{l+1}[i,j] = ActivationFunction\left(\sum_{k=0}^{k=N_l} W_l[i,k] * X_l[k,j] + b_l[i]\right)$$
(4.1)

$$X_{l+1} = ActivationFunction\left(MatrixMultiplication(W_l, X_l) + b_l\right)$$
(4.2)

Processing an input I through the neural network involves setting $X_1 = I$ and calculating the output O or X_{N+1} by recursively using Equation 4.2. Once the output is generated, a loss function calculates the loss with the reference output, and the loss is propagated back into the network using back propagation to calculate the gradients for the parameters in the network. An optimizer then takes these gradients and updates the parameter values to minimize the loss value. This process is repeated for multiple inputs for multiple times until the loss converges. The performance of a neural network can be measured using the following four metrics:

- 1. Accuracy : Accuracy achieved by the neural network on the functional task.
- 2. Memory : Memory required for storing the parameters of the neural network.
- 3. Compute : Floating point operations (FLOPS) required to process the neural network.
- 4. **Runtime :** The wall clock time taken to run the neural network on the hardware.

An optimal neural network achieves maximum accuracy by using the minimum amount of memory and compute. However, there is no definite way of arriving at such a neural network, and in practice, there are trade-offs between memory, compute, and accuracy. Sparsification of neural networks [28, 29, 51, 69] is an effective way to reduce memory and compute requirements with a minimal trade-off in accuracy.

4.2 Sparse Neural Networks

A sparse neural network (SNN) is a neural network where only a subset of connections are present between the neurons in a neural network. For example, in a dense feed forward neural network with N neurons in each layer, N^2 connections are present in each layer, with every neuron on the left connecting to every neuron on the right. In comparison, a sparse feed forward neural network has less than N^2 connections because only a subset of connections are present in each neural network layer. Figure 4.1 shows an example of a sparse feed forward neural network with four layers, where only 5 out of 12, 4 out of 9, 4 out of 9, and 6 out of 12 connections are present in layers one, two, three, and four respectively.

A connection in a neural network contributes to memory and compute in a neural network. Because a sparse neural network has fewer connections than a dense neural network, the parameter matrix W_l in Equation 4.2 becomes sparse. A sparse matrix can be efficiently stored and processed by avoiding storage and compute corresponding to the zero elements. Because of this, a sparse



Figure 4.1: An example Sparse Feed Forward Neural Network(SNN)

neural network takes less memory to store and requires less compute when compared to a dense neural network. One of the primary use cases of sparse neural networks is edge devices, where the resources are limited. Less memory and compute requirements of sparse neural networks make it an ideal choice for edge devices.

The runtime metric for evaluating the efficiency of a neural network is subjective, whereas other metrics (accuracy, memory, and compute) are objective. This is because the runtime is subject to the choice of hardware. On an ideal hardware, a sparse neural network with s% reduction in compute when compared to the corresponding dense neural network should result in a runtime speedup of 100/(100 - s). For example, a 50% reduction in compute should result in 2x speedup in runtime. But on parallel hardware like GPU, sparse neural networks lead to poor or negative runtime performance gains due to the irregularity in computation of sparse neural networks. A tough way to realize the fruits of reduced number of FLOPS is to design specialized ASIC hardware like EIE, SCNN [27,68] for accelerating sparse neural networks. But this requires a lot of work in terms of developing hardware, middleware, and software stack. On top of that, the utility of such specialized hardware is limited to only sparse workloads. In contrast, a parallel hardware like GPU can accelerate many compute intensive workloads from different application domains. In order to address the runtime performance on GPU like hardware, the focus has shifted to structured sparse neural networks [32, 50, 58, 62, 87] that lead to improved runtime performance.

Sparse neural networks can be divided into types: unstructured and structured, depending on the sparsity pattern (how the neurons are connected) present in them. In an unstructured sparse neural network, the sparsity pattern does not follow any particular pattern. In a structured sparse neural network, the sparsity pattern has a structure. For example, when all neurons connect to an equal

number of neurons, we say it has a sparsity pattern with a uniform structure. Structured sparse neural networks are essential because they lead to improved runtime metric on widely used AI hardware like GPU when compared to an unstructured sparse neural network. The improvement is proportional to the degree of structure in the sparsity pattern.

Computational approaches for generating sparse neural networks can be divided into four categories depending on when and how the sparsity is incorporated in the neural network training process.

Dense to sparse training approach: The training starts with a dense neural network and ends with a sparse neural network. During the training, connections are removed using various techniques like pruning, regularization, and masking etc.

Finetuning approach : It is a dense to sparse training approach, with a key difference that the training starts with a pretrained dense neural network instead of a randomly initialized dense neural network.

Dynamic Sparse to Sparse training approach : The training starts and ends with a sparse neural network with connections updated during the training process. The important thing is that the number of connections remain same throughout the training period.

Static sparse to sparse training approach : The training starts and ends with a sparse neural network without change in connections during the training process.

4.3 Runtime performance on GPU

A sparse neural network has only a subset of connections that a dense network has. This reduced number of connections makes the weight parameter matrix, i.e., W_l in layer l, into a sparse matrix, and the *MatrixMutliplication* operation in Equation 4.2 becomes an *SDMM*(Matrix Multiplication of a Sparse matrix with a Dense matrix) operation ($C = A_s \times B$). The number of FLOPS in the SDMM operation depends only on the amount of sparsity in A_s and is independent of the sparsity pattern in which the non-zero elements of A_s are arranged. Hence on an ideal hardware, the runtime of SDMM should depend only on the amount of sparsity in A_s . However, on a GPU, the run time of the SDMM operation depends also on the sparsity pattern present in A_s . From Figure 4.3, we see that for a given sparsity or FLOPS, the run time of SDMM when A_s has a block sparsity pattern is close to 4x faster when compared to the case when A_s has an unstructured sparsity pattern. The reason for the performance gap is that efficient parallel algorithms on a GPU can be designed for SDMM when A_s has a block sparsity pattern. Figure 4.3 also considers the case of row/column sparsity where entire rows/columns are removed. From Figure 4.3, we also see that the row/column sparsity pattern is 4-5x faster than the block sparsity pattern. As entire rows/cols are removed in row/column sparsity, the resulting SDMM operation can be processed as a small dense matrix multiplication using fast dense GPU kernels. The choice of sparsity pattern plays an essential role in the run time performance of SDMM operation on a GPU and, thus, the run time performance of the sparse neural network on the GPU.



Figure 4.2: Sparsity patterns



Figure 4.3: The effect of sparsity pattern on the performance of the SDMM operation on a V100 GPU. The matrix size is set to 4096 and the block size in the block sparsity pattern is set to 32. The runtime for dense case is 10.5 ms.

4.4 Sparsity Pattern Co-design

Connectivity between neurons in a neural network layer can be captured using a mask matrix M with M[i][j] value set to one if i^{th} neuron on the left connects to j^{th} neuron on the right, and zero

otherwise. If there are N neurons on either side of a neural network layer, and if the sparsity is 50%, then there are $(N^{2}!)/((N^{2}/2)! \times (N^{2}/2)!)$ possible mask matrices. Even when N = 8, the count goes to a staggering $\sim 1.8 \times 10^{18}$. However, if a block-structured sparsity pattern with block size 2x2 is imposed, the number of possibilities comes down to 12870. Furthermore, if a row or column structured sparsity pattern is imposed, the number of possibilities comes down to 70. So clearly, we can see that the number of possible mask matrices reduces when the sparsity pattern becomes more structured. Fewer possible mask patterns translate to less variability in memory access and compute patterns on the GPU. From Figure 4.3, we can see how going from unstructured to block to row/column sparsity pattern helps in improving the runtime performance on the GPU. *Does this mean that choosing a highly structured sparsity pattern leads to efficient sparse neural networks*? The answer is No. Because the choice of sparsity pattern also affects the neural network's accuracy metric. While structure has positive effect on the runtime metric, it has an inverse effect on the accuracy metric, with more structure and as the sparsity pattern gets more structured, the freedom in the choice of which parameters to keep or discard decreases and thus leads to lesser accuracy.

Pruning is a widely used method for generating sparse neural networks, where the parameter's magnitude determines the parameter's importance. To achieve a sparsity of s%, pruning methods follow a greedy approach and remove the bottom s% of the parameters. But doing so results in an unstructured sparsity pattern, which we already know results in poor runtime on the GPU. So in order to improve the runtime performance of sparse neural networks on GPU, structured pruning methods are used. In structured pruning methods, parameters are pruned in a structured manner while maximizing the preservation of important parameters. However, imposing structure results in a suboptimal choice of parameters, leading to lower accuracies. To demonstrate the effect of the structure on accuracy, we picked the image classification task on CIFAR100 dataset. For the neural network, we chose VGG11 neural network [77] and generated block sparse neural networks with varying degrees of structure by varying the block size from 1x1 to 16x16. From Figure 4.4, we can see that as the block size increases, the error (100-accuracy) increases, or in other words, the accuracy decreases. From Figure 4.4, we can also see that the runtime decreases as the block size increases. This is because SDMM operation with a larger block size can be efficiently processed on the GPU. More structure decreases accuracy but improves runtime. Less structure improves accuracy but decreases runtime. The challenge now is to design sparsity patterns that lead to a better tradeoff between task accuracy and GPU runtime.

Part I of the thesis focused on designing better algorithms and data structures for sparse matrix operations on the GPU where the sparsity pattern of the input sparse matrices are fixed, i.e., Quasi-



Figure 4.4: Dual effect of sparsity pattern on the accuracy and the run time of a sparse neural network for image classification task on the CIFAR-100 dataset using VGG11 model. The error/accuracy for the dense neural network is 31.5/68.5.



Figure 4.5: Impact of sparsity pattern on accuracy and runtime metric

Band sparsity pattern in QBMM (Matrix Multiplication computational task ($C = A_s \times B_s$), where A_s and B_s are Quasi Band sparse matrices), and Hines sparsity pattern in HSOLVE (Matrix solver computational task ($A_s x=b$), where A_s is a Hines sparse matrix). In both cases, the sparsity pattern of the input sparse matrices is fixed, and the algorithm and data structures are co-designed by exploiting the properties of the input sparsity pattern. The core operation in sparse neural networks is the SDMM(Matrix Multiplication of a Sparse Matrix with a Dense Matrix) ($C = A_s \times B$) operation. Here the sparsity pattern of A_s is not fixed and can be chosen. Figure 4.5 shows how the choice of the sparsity pattern affects the accuracy, the data structure for A_s , and the algorithm for SDMM operation. Given these dependencies, we can generate efficient sparse neural networks on GPU only by co-designing the sparsity pattern, the data structure, and the algorithm on the GPU.

Chapter 5

Dynamic Block Sparse Reparameterization of Convolutional Neural Networks

Structured sparse neural networks help in improving runtime performance on GPU hardware. Several types of structured sparsity patterns like filter sparsity [57], channel sparsity [40] and block sparsity [81] were proposed. In filter/channel sparsity, the unit is a row/column which is either completely zero or non-zero. In block sparsity, the unit is a block of parameters with dimensions (bh, bw), where the parameters in a given block are either all zero or non-zero. Both filter and channel sparsity are special cases of block sparsity and it has been shown in [76], that the computation of block sparse operations can be efficiently processed on parallel hardware like GPU. This is because the algorithms for processing block sparse operations heavily piggyback on the algorithms used for dense operations. Block sparsity is important also because, newer processor architectures have components like systolic array (TPU) and tensor core (V100 GPU), which process matrices in blocks. Hence block sparsity could be better exploited by these processor architectures. In this work, due to the generic nature of block sparsity pattern and its good runtime performance benefits, we developed techniques to generate efficient block sparse networks. Following are our main contributions:

 We develop a simple, easy to use, and effective approach (DBSR) for generating structured sparse neural networks with most generic block sparsity pattern. When compared to dense training, our DBSR approach requires only one extra hyper parameter ζ, which controls the amount of sparsity.

- As part of DBSR approach, we propose a block scaling operation and a block scaled convolution layer which forms the basis for quickly exploring new techniques for generating structured sparse neural networks.
- We show the effectiveness of DBSR approach on important vision tasks like image classification and semantic segmentation over varied networks (VGG, Resnet20, Resnet50, ResneXt50, ERFNet) and datasets (CIFAR, Imagenet, Cityscapes).

In Section 5.1, we detail our approach and discuss the performance aspects of block sparsity in context of a GPU. In Section 5.2, we go through our results and experiments. We finally conclude the chapter in Section 5.3.

Related Work : Generation of structured sparse neural networks can be classified into two paradigms: *in vitro* and *in vivo*. In the *in vitro* paradigm, a sparse model is generated from a pre-trained dense model. In the *in vivo* paradigm, a sparse model is learned during the training process. The common methodology underlying approaches based on *in vitro* paradigm has two main steps. They are: 1) Prune structural units from a pre-trained model based on some criterion 2) Finetune the pruned model to recover accuracy. Approaches in *in vitro* paradigm differ only in the chosen pruning criterion, and sparsity pattern. The L_1 norm of the weights of a unit structure is chosen as the pruning criterion in [50, 58] and [81] for pruning filters and blocks respectively. In [57], the filters of a current layer are pruned based on the channel weights of next layer. In [32], channel pruning is performed based on LASSO regression based channel selection method. In [87], filters are pruned across layers in a joint fashion by back propagating scores corresponding to the filters. In [62], filters are pruned using Taylor expansion based pruning criterion. Even though *in vitro* approaches are less computationally intensive than a full training, there are many challenges like determining the pruning percentages for layers, the choice of hyper parameters for fine-tuning and so on. This makes it difficult to use *in vitro* approaches in practice.

Our approach falls under the *in vivo* paradigm, where sparsity generation is tightly coupled with the training process. In our approach, we associate a scaling parameter for each structure, thus allowing our method to generate varied type of structured sparsity patterns. We focus on the block sparsity pattern, which is the most generic structured sparsity pattern with filter and channel sparsity patterns being sub cases of block sparsity. Many of the previous approaches in *in vivo* paradigm, focus only on filter or channel sparsity. In [53], filter sparsity is generated by regularizing scaling parameters already present in batch norm layers. To also support filter sparsity generation for layers with out batch norm, [40] introduced additional scaling parameter for each output activation channel. Both [40, 53] differ from our approach as they scale output activation



Figure 5.1: W and S are parameters of a neural network layer with block size (2x2). Before training, S is initialized to 1. During training, S is regularized to induce sparsity. At the end of the training, S becomes sparse due to regularization, and thus result in a block sparse neural network layer.

channels rather than weights thus limiting to generation of only filter sparsity pattern. In [82], weights of a chosen structure are regularized by using group LASSO regularization. This differs from our approach, as we regularize scaling factors associated with the block of weights rather than weights themselves. The idea of using group LASSO regularization was adapted for recurrent neural networks to generate structured sparse RNNs with block sparsity pattern [64]. In [55], a binary variable is associated with each structure, and the binary value is stochastically learned during training. Our method is a non stochastic approach, where scaling variable is a real valued number that is trained along with weight parameters.

5.1 Approach

Our approach DBSR (Dynamic Block Sparse Reparameterization) generates block sparsity through the training process in a dynamic fashion. The main idea in the DBSR approach is to use trainable scaling parameters for the blocks and generate block sparsity by pushing the values of scaling parameters to zero using L_1 regularization. DBSR approach makes use of two building blocks, namely block scaling operation and block scaled convolution layer. We first describe them in detail and later use them to formulate our approach. We also discuss about the performance aspects of block sparse networks.

Block Scaling Operation: Block scaling operation denoted by $*_b$, takes two input tensors S and W, and produces a single output tensor W_S . Input tensor S is a 2D tensor with dimensions (s_1, s_2) and W is at least a 2D tensor with dimensions $(w_1, w_2, ..., w_n)$. Blocking is performed on outer two dimensions of W, with dimensions (bh, bw), where $bh = w_1/s_1$ and $bw = w_2/s_2$. In the

forward (resp backward) pass, each element in $W(dW_S)$ is scaled by the scaling factor associated with it's block to generate $W_S(dW)$. In the backward pass each entry in dS is calculated by taking dot product of the associated blocks in W and dW_S . Algorithms 13 and 14, detail the computation in the forward and backward pass of block scaling operation.

Block scaled convolution layer : The convolution operation in a convolutional layer consists of producing an output activation tensor O, by applying convolution on an input activation tensor I. O = conv(W, I), where W is a 4D parameter/filter tensor. Block scaled convolution layer is built on top of regular convolution layer with additional parameters and computation. In block scaled convolution layer, we store an additional 2D parameter tensor S along with W. The computation in a block scaled convolution layer is processed in two steps: 1) A block scaling operation is performed on S and W to generate W_S ($W_S = S *_b W$). 2) Using W_S , the convolution operation is performed on input I to produce output O ($O = conv(W_S, I)$).

Algorithm 13 Forward pass for block scaling operation $W_S = S *_b W$.

1: $s_1, s_2 = S.shape$ 2: $w_1, w_2, ..., w_n = W.shape$ 3: $bh = w_1/s_1$ ▷ Block height 4: $bw = w_2/s_2$ ▷ Block width 5: 6: for $i = 0 : w_1 - 1$ do for $j = 0 : w_2 - 1$ do 7: $W_{S}[i,j] = S[i/bh, j/bw] * W[i,j]$ 8: end for 9: 10: end for 11: 12: return W_S

Algorithm 14 Backward pass for block scaling operation $W_S = S *_b W$.

1: for $i = 0 : s_1 - 1$ do for $j = 0 : s_2 - 1$ do 2: hr = i * bh : (i + 1) * bh - 13: wr = i * bw : (i + 1) * bw - 14: $W^{bij} = W[hr, wr]$ 5: $dW_S^{bij} = dW_S[hr, wr]$ 6: $dS[i,j] = Sum(W^{bij}.*dW^{bij}_S)$ 7: $dW[hr, wr] = S[i, j] * dW_S^{bij}$ 8: end for 9: 10: end for 11: 12: return dW, dS

Formulation : Learning block sparsity in a neural network is jointly modelled with primary learning task as an optimization problem according to Equation 5.1, where (x_i, y_i) is an instance in dataset, M is a model, and L is a loss function. W and S correspond to weight and block scaling parameters respectively. $R_w(.)$ and $R_S(.)$ are regularizers used for W and S respectively.

$$\min_{W,S} \sum_{i=1}^{N} L\left(M(S, W, x_i), y_i\right) / N + R_w(W) + R_s(S)$$
(5.1)

Given the choice of block size, all convolutional layers in model M are converted into block scaled convolution layers. Before training, all parameters in S are given equal importance by initializing them to a value of 1. As training progresses, the values of parameters in S change dynamically due to the error propagation from loss function and regularizers. In order to generate sparsity, we push the values of scaling parameters S to zero by using L_1 regularization ($\zeta * |S|$) for $R_s(.)$. The hyper parameter ζ controls the amount of sparsity in the model. Zero clipping is performed on S (where $S = \max(0, S)$) after every training iteration to ensure that the values in Sremains positive. Once the model is trained, S can be discarded after updating blocks in W using a block scaling operation ($W = S *_b W$).

A <u>11</u>		A ₁₃		B11	B ₁₂	B13	B14		C ₁₁	C ₁₂	C ₁₃	C ₁₄
	A ₂₂	A ₂₃		B21	B22	B23	B24		C ₂₁	C22	C ₂₃	C24
A31			A34	B31	B32	B33	B34	=	C31	C ₃₂	C33	C34
	A42	A43	A44	B41	B42	B43	B44		C41	C42	C43	C44

Figure 5.2: Cache friendly blocking scheme for efficiently processing BSMM (block sparse matrix multiplication) operation.

In our approach, there is a small memory and compute overhead when compared to dense model training. Memory overhead is due to the need for storage of block scaling parameters S along with weight parameters W. Compute overhead comes from additional block scaling operation performed before convolution operation. These memory and compute overheads are limited only to training, and are not present in inference.

Performance of block sparse CNNs : In popular deep learning libraries like cuDNN [17], convolution operation is performed by posing it as a matrix multiplication operation (GEMM) $(O = W * I_{mat})$, where each row in W matrix corresponds to a flattened 3D filter and I_{mat} is a lowered matrix of 3D input *I*. So in order to accelerate convolution operation, an efficient GEMM operation is required. In case of block sparse convolution, we need an efficient block sparse matrix matrix multiplication (BSMM) operation.

Efficient algorithms for the BSMM operation can be adapted from fast dense matrix multiplication (GEMM) algorithms that are based on cache friendly blocking schemes. In blocking scheme, the C matrix is divided into blocks, and a block in C is computed by iteratively loading corresponding blocks in A and B into a cache and then multiplying them. By keeping the blocks in the cache, memory accesses become more efficient and thus increases runtime performance. For BSMM, we can use the same blocking scheme based algorithms used for GEMM, except that the computation corresponding to zero blocks are skipped. For example in Figure 5.2, we can see that a block C_{11} is computed in only two steps as there are only two nonzero blocks corresponding to C_{11} in A. Using the blocking scheme, efficient block sparse kernels for GPU are successfully developed in [76] to deliver ideal speedups.

5.2 Experiments

We evaluate our approach on Imagenet [75] and cityscapes [19] datasets for the task of image classification and semantic segmentation respectively. For image classification, we choose two state of the art networks Resnet50 [31], and ResneXt50 [83]. And for semantic segmentation, we choose realtime semantic segmentation network ERFNet [74]. In all our experiments, we use the same experimental setting used for dense model training.

5.2.1 Effectiveness of DBSR on classification networks

Resnet50/Imagenet : In Resnet50, we chose 32x32 as the block size and trained multiple networks by varying ζ which controls the sparsity. From Table 5.1, we can see that with only an increase of 0.47 in Top-1 error, our model Resnet50-32x32-A decreases both parameters and FLOPS of Resnet50 by ~30%. More compact models can be generated by increasing the value of ζ . For instance, Renset50-32x32-C decreases parameters by a factor of 2.5x, and FLOPS by a factor of 2x, with an increase of 2.66 in Top-1 error when compared with Resnet50. In [31], two small dense networks Resnet18 and Renset34 are carefully designed to have less memory and computational footprint. When compared with them, our models have more efficiency. Our model Resnet50-32x32-D has 1.47 less Top-1 error, and takes 30% less parameters and 20% less FLOPS when compared to Resnet18. Similarly when compared with Resnet34, our model Resnet50-32x32-B has 0.34 less Top-1 error, and takes ~38% less parameters and FLOPS. From comparisons with small dense models such as Resnet18 and Resnet34, we can observe that big sparse models are more efficient in terms of both memory and FLOPS. This observation was also made by [63] in case of recurrent neural networks.

ResneXt50/Imagenet : In ResneXt50, grouped convolutions are introduced to improve accuracy and reduce memory and computational requirements. While applying DBSR, we do not replace grouped convolutional layers with block scaled convolution layers. The reason for that is a grouped convolution layer has inherent block sparsity pattern with blocks strictly lying on the main diagonal. From Table 5.1, we can see that with only an increase of 0.48 in Top1-error, our model ResneXt50-32x32-A decreases parameters and FLOPS of ResneXt50 by 49% and 45% respectively. While model ResneXt50-32x32-C with higher ζ values decreases parameters by a factor of ~3.1x and FLOPS by a factor of ~2.6x with only 2.01 increase in Top-1 error.

Comparison with SSS [40] : SSS (Sparse Structure Selection) is a recent work, where structured sparse neural networks are generated from dense networks (Resnet50, ResneXt50) through the training process. Similar to SSS, our DBSR approach generates structured sparse networks

Model	Top-1 Error	#Params	#FLOPS
Resnet18	30.24	11.67M	1.8B
Resnet34	26.70	21.77M	3.6B
Resnet-50	24.61	25.5M	3.85B
Resnet50-32x32-A	25.08	17.89M	2.74B
Resnet50-32x32-B	26.36	13.36M	2.19B
Resnet50-32x32-C	27.27	10.81M	1.85B
Resnet50-32x32-D	28.77	8.41M	1.42B
ResneXt-50	23.35	24.96M	4.23B
ResneX50-32x32-A	23.83	12.72M	2.32B
ResneXt50-32x32-B	24.69	9.39M	1.83B
ResneXt50-32x32-C	25.36	8.03M	1.65B
ResneXt50-32x32-D	26.20	6.60M	1.59B

Table 5.1: Block sparse models generated using DBSR approach for the task of image classification on Imagenet ILSVRC2012 dataset. Centre crop is used for calculating error.



Figure 5.3: Comparison of block sparse models from DBSR approach with structured sparse models from (SSS) [40] for the task of image classification on Imagenet dataset.

with block sparsity pattern through the training process. From Figure 5.3, we can see that when compared to models generated using SSS, DBSR models are more efficient in memory, compute, and error. For both Resnet50 and ResneXt50, we can see from Figures 5.3a & 5.3c, that DBSR approach becomes even more effective as the number of parameters in the model decreases. This is especially useful for effectively running low capacity models on resource limited embedded GPUs like Jetson TK2.

Comparison with structured pruning methods : We compare our DBSR approach with other structured pruning based methods like filter pruning [50], channel pruning [32] and ThinNet [57]. From Table 5.2, we can see that our model Resnet50-32x32-A has 0.75-1.6 less Top-1 error when

Model	Top-1 Error	Top-5 Error	Params	#FLOPs
ResNet-34-pruned [50]	27.44	-	19.9M	3.08B
Resnet-50-pruned [50] (From [40])	27.12	8.95	-	3.07B
Resnet-50-pruned(2x) [32]	27.70	9.20	-	2.73B
Resnet50-pruned (ThinNet-30) [57]	27.96	9.33	16.94M	2.44B
Resnet-50-32x32-B (Ours)	26.36	8.23	13.36M	2.19B
Resnet-101-pruned [86]	25.44	-	17.30M	3.69B
Resnet50-32x32-A (Ours)	25.08	7.73	17.89M	2.74B

Table 5.2: Comparison of block sparse models from DBSR approach with models from other structured pruning methods for the task of image classification over Imagenet dataset. We can see that DBSR models are more efficient in parameters and FLOPS when compared to SSS models.



Figure 5.4: Effect of varying block size on model efficiency.

compared to other pruning based approaches while still taking less number of parameters/FLOPS. When compared with pruned Resnet-101 from [86], our model has 0.36 less Top-1 error and has 26% less FLOPS for similar number of parameters. It should be noted that pruning approach is orthogonal to DBSR approach and can be applied on DBSR models to further decrease memory and computational requirements.

Varying block size : In this experiment, we would like to see the effect of block size on model efficiency. We train multiple block sparse models with block sizes 8,16 and 32. From Figure 5.4, we see that the models with smaller block sizes are more effective when the model capacity (number of parameters/FLOPS) is less. This might be because when number of parameters are less, flexibility offered by smaller block sizes helps in improving connectivity. But as the model capacity increases, the effectiveness gap among block sizes decreases. From Figure 5.4, we can see that the block size 16 has more efficiency than block size 8 for models with higher capacity.



Figure 5.5: Comparison of DBSR approach with block pruning approach on image classification task over CIFAR datasets for VGG network with block size 32x32

5.2.2 Comparison with block pruning

Our DBSR approach generates structured sparse neural networks with a block sparsity pattern. Block sparse neural networks can also be generated by adapting pruning approach [31] to prune blocks instead of individual elements. We compare our DBSR approach with block pruning approach on CIFAR datasets CIFAR10 & CIFAR100 [45] over VGG [77] and ResNet-20 [31] networks. Our VGG network for CIFAR is adapted from VGG16 network [77], with fc6 and fc7 replaced by a single fc layer of size 512, and batch normalization layer is added to all convolution and fc layers.

In DBSR, we train the model for 240 epochs with base learning rate set to 0.1. For block pruning, we finetune the pruned model for 24 epochs with base learning rate set to 0.01. For both DBSR and block pruning, the learning rate is decreased by a factor of 10 at 3/6, 4/6 and 5/6 th of the training/finetuning cycle. We use SGD with nestrov momentum optimization with momentum and weight decay values set to 0.9 and 1e-4 respectively.

VGG/CIFAR : For VGG, we set the block size to 32x32 and generate block sparse models with different model capacities. From Figure 5.5, we can see that models generated from our DBSR approach are more efficient than that of block pruning approach. Block pruning has an effect on error even for small amount of pruning. However, that is not the case for DBSR models. For CIFAR10, up to 90% parameter reduction and 50% FLOPS reduction can be achieved while maintaining accuracy comparable to that of dense. Similarly for CIFAR100, upto 80% parameter reduction and 40% FLOPS reduction can be achieved with comparable accuracies to that of dense.

Resnet20/CIFAR : In Resnet20, the number of channels in convolutional layers are less and range only from 16 to 64. Hence we chose 8x8 as the block size for Resnet20. From Figure 5.6, we can see that our approach performs better than block pruning. The effectiveness of DBSR for



Figure 5.6: Comparison of DBSR approach with block pruning approach on image classification task over CIFAR datasets for Resnet20 network with block size 8x8

Resnet20 is less when compared with that of VGG. This might be due to the residual connections in Resnet20, which make it more robust to block pruning approach. With little loss in accuracy, up to 50% parameters and 30% FLOPS can be reduced for CIFAR10, and upto 30% parameters and FLOPS can be reduced for CIFAR100.

5.2.3 Semantic Segmentation

We extend our DBSR approach to the task of semantic segmentation and evaluate on cityscapes dataset [19] using ERFNet [74] network. The choice of ERFNet is due to its low capacity and ability to do real time semantic segmentation, which is critical for many applications. ERFNet follows an encoder-decoder architecture with factorized convolutions. In our experiments, we choose 16x16 as the block size and convert all convolution layers into block scaled convolution layers. For training, we use the same setup used for dense training. Table 5.3, shows our results on ERFNet over cityscapes dataset. With only a loss of 1% in mIoU, our model ERFNet-16x16-A decreases parameters by 30% and FLOPS by 20% when compared with dense ERFNet. More compact models can be found in Table 5.3 by increasing the value of ζ .

Model	mIoU	#Params	#FLOPS
ERFNet	67.75	2.05M	29.83B
ERFNet-16x16-A	66.74	1.47M	23.82B
ERFNet-16x16-B	66.30	1.23M	21.92B
ERFNet-16x16-C	65.37	1.02M	19.47B
ERFNet-16x16-D	63.61	0.75M	15.87B

Table 5.3: Block sparse ERFNet models with 16x16 block size for the task of semantic segmentation over cityscapes dataset.

5.3 Conclusion

In this work, we developed a simple and effective technique called DBSR (Dynamic Block Sparse Reparameterization) for generating efficient block sparse neural networks. Unlike posttraining pruning approaches, our DBSR approach tightly integrates structured sparsity generation with the training process and results in efficient models on standard vision tasks like image classification and semantic segmentation when compared to state of the art structured sparsity approaches.

Chapter 6

Hierarchical Block Sparse Neural Networks

In neural networks, each parameter is equally important before the training begins. As the training progresses, the importance of these parameters varies. One can prune away the least important parameters during or after the training process with minimal/no loss to the model accuracy. Early studies by [21,30], have shown the efficacy of pruning technique in reducing the model complexity. More recently, pruning techniques were successfully applied on many classes of neural networks: On Convolutional Neural Networks(CNNs), Han et al. [29] show how to generate sparse CNNs from pretrained dense CNNs using finetuning approach. On recurrent neural networks(RNNs), Narang et al. [63] show how to generate sparse RNNs by pruning away parameters at regular intervals during the training process.

Most common way of pruning a neural network is fine grained pruning, where pruning is performed at the level of individual element and the sparsity obtained due to it is unstructured. For a given neural network, if K% of the model parameters are uniformly pruned across all layers, the computational complexity of the model reduces by a factor of 100/(100 - K). For example, pruning half of the parameters in a model decreases the computational complexity by 2x. But for fine grained pruning, the runtime benefit obtained by decrease in computational complexity is far from ideal on regular parallel hardware such as GPU.

To deal with runtime performance issues of unstructured sparse neural networks, researchers have resorted to pruning parameters in a more structured way and leverage the structure for runtime performance. Towards that end, Narang et al. [64] have performed block pruning in Recurrent Neural Networks (RNNs), and Wen et al. [82] have performed filter pruning. But the common observation is that for a given sparisty, the model obtained by these highly structured pruning (block, filter) have less accuracy than fine grained pruning.

Fine grained pruning lacks runtime performance, and highly structured pruning lacks model accuracy. But we would like our sparse networks to have both accuracy and runtime performance. In this work, we arrive at such sparse models using our proposed class of sparse neural networks called **HBsNN** (Hierarchical Block sparse Neural Networks).

Motivation : The importance of a parameter in a neural network is strongly correlated with its magnitude. When we perform highly structured pruning like block sparse, we lose significant number of high magnitude parameters due to the imposed structural constraints. Row 1 in Table 6.1, shows the percentage of top {10,20,30,40,50}% elements retained after pruning 50% of elements in a block sparse manner using 32x1 block size on a pretrained Resnet-v2-50 model. One can see that 24% of the top 10% elements are pruned out. It has been found empirically that high magnitude parameters play a significant role in generating sparse models with good model accuracies. One simple way to retain high magnitude weights is to bring fluidity to the sparsity structure. In Table 6.1, one can see that for a given sparsity of 50%, incorporating multiple levels of structure leads to improved top-* percentages. Based on this observation, we propose a class of sparse neural networks called **H**ierarchical **B**lock **s**parse **N**eural **N**etworks (HBsNN) which are more fluid and can retain high magnitude parameters. Sparse models obtained by fine grained pruning or block sparsity are a subset of HBsNN.

(block-heightx1)/sparsity	top-10	top-20	top-30	top-40	top-50
32/50	76.04	69.81	65.80	62.82	60.42
(32,16)/(75,75)	80.25	72.95	68.16	64.63	61.80
(32,16,8)/(75, 87.5, 87.5)	85.00	76.56	70.91	66.77	63.40
(32,16,8,4)/(75, 87.5, 93.75, 93.75)	89.83	80.06	73.49	68.63	64.75
(32,16,8,4,1)/(75, 87.5, 93.75, 96.875, 96.875)	100	90.71	79.23	72.03	66.84

Table 6.1: HBS configuration vs Retained percentage on ResNet-v2-50 model.

6.1 Approach

In HBsNN, the sparse parameter matrix M of a given layer is composed of multiple sparse parameter matrices i.e, $M = M_1 + ... + M_N$, where each M_i is a block sparse matrix with different block dimensions. As it is suboptimal to split the value of a non-zero in M across many matrix levels, a non-zero element in M is contributed by only one M_i i.e, $\forall_{j,k\in N}M_j * M_k = 0$. Apart from this, matrices have to satisfy hierarchical structure, where dimensions of block in M_{i+1} should divide dimensions of block in M_i i.e, $Dim(M_i)\%Dim(M_{i+1}) = 0$. In Figure 6.1, we have a 3 level configuration with block dimensions 4x4,2x2 and 1x1 respectively.



Figure 6.1: Hierarchical block sparse(HBS) neural network layer L with three parameter groups L_1, L_2 , and L_3 . The corresponding parameter group matrices M_1, M_2 , and M_3 are block sparse matrices with block sizes 4x4,2x2, and 1x1 respectively.

6.1.1 Pruning Methodology

For a given matrix I, block dimensions (bh, bw) and sparsity sp, block sparsity is generated by dividing the matrix I into a grid, where each grid element is of size (bh, bw). Each grid element is then given a rank using the absolute summation values of that grid block. We then sort these values and prune away sp % of blocks to generate a block sparse matrix. In case of hierarchical block sparsity with N levels, block sizes $BS = [(bh_1, bw_1), \dots (bh_N, bw_N)]$ and sparsities $SP = sp_1, \dots sp_N$ are provided for all the levels. Let I_k and M_k be the input and output matrices at level k. In level k, we perform a block sparse pruning with block size (bh_k, bw_k) and sparsity sp_k to generate M_k . We then generate input to layer k + 1 by removing elements of M_k from I_k i.e, $I_{k+1} = I_k - M_k$. Figure 6.2, shows an example of 2 level HBS pruning on 4x4 matrix with BS=[(2,2),(1,1)] and SP = [50,75]. In case of networks where parameters of a layer are arranged in more than two dimensions, block dimensions correspond to outer two dimensions. For example, in case of CNNs, blocking is performed on ofm(output feature maps) and ifm(input feature maps).



Figure 6.2: 2-Level block sparse generation : Block sizes=[(2x2),(1x1)] sparsities=[50,75]

6.1.2 Performance Model

In this section, we describe a performance model for evaluating performance of a layer in a HBsNN. For a given HBS configuration, let F_{dense} and F_{sparse} be the amount of compute for dense and sparse operations respectively. As a layer in HBsNN has multiple levels (L_1, \ldots, L_N) , $F_{sparse} = \sum_{i=1}^{i=N} F_{sparse}^{L_i}$, where $F_{sparse}^{L_i}$ is the amount of compute in i^{th} level of the layer. Due to irregularity in sparse computation, it is not always possible to realize the ideal speedup i.e, F_{dense}/F_{sparse} . The achievable speedup depends primarily on two factors: 1) Amount of sparsity and 2) Dimensions of blocks. We quantify the sub-optimal speedup with an irregular factor function irf(sparsity, blockDimensions) parameterized by those two factors. By taking these factors into effect, the cost of dense (C_{dense}) , and the cost of sparse neural network (C_{sparse}) are defined according to Equation 6.1 and 6.2 respectively. The achievable speedup can then be defined according to Equation 6.3. The $irf(\ldots)$ function in Equation 6.2 varies from system to system and has to be obtained by running micro benchmarks on that system. But on a regular parallel hardware such as a GPU, irf function is inversely proportional to block size. So, inorder to maximize performance, one needs to maximize sparsity for levels with smaller block sizes, and minimize sparsity for levels with larger block sizes.

$$C_{dense} = F_{dense} \tag{6.1}$$

$$C_{sparse} = \sum_{i=1}^{i=N} \frac{F_{sparse}^{L_i}}{irf(Sparsity(L_i), BlockDims(L_i))}$$
(6.2)

$$SpeedUp = \frac{C_{dense}}{C_{sparse}}$$
(6.3)

6.2 Results

6.2.1 ResNet-v2-50/Imagenet

We take a pretrained Renset-v2-50 model with top-1 and top-5 accuracy of 76.13% and 92.86% respectively and then generate sparse models from it using prune and retrain methodology from [29]. Except for the first convolution layer and the last fully connected layer, we prune all the layers. The pruned model is then trained for 18 epochs with the same set of hyper parameters as that of the pretrained model. The initial learning rate for training is set to $1/100^{th}$ of the base learning rate used for pretrained model. A step based learning rate decay is followed, where learning rate is decreased by a factor of 10 and 100 respectively at 9th and 14th epoch respectively.

Varying sparsity : In this experiment, we study how the accuracy varies with respect to sparsity. We vary sparsity from 50% to 87.50% with block size set to 1x1. From Table 6.2, we can see that accuracy decreases with sparsity. This is due to the fact that more number of elements are pruned away with increase in sparsity and this reduces the model capacity.

Sparsity	Top-1 Accuracy	Top-5 Accuracy
50	76.42 (+0.29)	93.03 (+0.17)
75	75.12 (-1.01)	92.34 (-0.52)
87.50	71.58 (-4.55)	90.58 (-2.28)

Table 6.2: Varying sparsity with block size set to 1x1.

Varying block size : In this experiment, we study how accuracy varies with respect to block size. Other parameters like sparsity and number of levels are kept the same. From Tables 6.3 and 6.4, we can see that the accuracy decreases with increase in the block size. This is due to the fact that as we increase the block size, a large number of high magnitude elements are pruned away due to increased structural constraint.

Block-size	Top-1 Accuracy	Top-5 Accuracy
4x1	75.73 (-0.40)	92.70 (-0.16)
8x1	75.19 (-0.94)	92.49 (-0.37)
16x1	75.03 (-1.10)	92.36 (-0.50)
32x1	74.52 (-1.61)	91.99 (-0.87)

Table 6.3: Varying block size with sparsity set to 50%.

Block-siz	e/sparsity	Accu	ıracy
Level-1 (L1)	Level-2 (L2)	Top-1	Top-5
4x1/53	1x1/97	75.93 (-0.20)	92.81 (-0.05)
8x1/53	1x1/97	75.67 (-0.46)	92.65 (-0.21)
16x1/53	1x1/97	75.55 (-0.58)	92.75 (-0.11)
32x1/53	1x1/97	75.26 (-0.87)	92.48 (-0.38)

Table 6.4: Varying block size with two levels. (Cumulative-Sparsity=50)

Varying sparsity distribution: In this experiment, we set sparsity to 50% and distribute it across multiple levels with block sizes ranging from 32x1 to 1x1 in a hierarchical fashion. Each row in Table 6.5 corresponds to a Hierarchical block sparse configuration and we can see that accuracy increases by having more fluidity in the structure imposed on sparsity. Another way of bringing fluidity and retaining structure is through Quasi block sparse configuration, which is a subset of Hierarchical block sparse configuration. In this case, there are two levels where one level has block sparsity and another level has unstructured sparsity with block size 1x1. In Quasi block sparse configuration, block sparsity caters for performance and unstructured sparsity caters for accuracy. From Table 6.6, we can see that the accuracy increases with increase in fluidity and with minimal loss to accuracy, significant amount of compute can be made regular.

	Sparsity					ıracy
L1-(32x1)	L2-(16x1)	L3-(8x1)	L4-(4x1)	L5-(1x1)	Top-1	Top-5
50	-	-	-	-	74.52 (-1.61)	91.99 (-0.87)
75	75	-	-	-	74.85 (-1.28)	92.37 (-0.49)
75	87.50	87.50	-	-	75.18 (-0.95)	92.49 (-0.37)
75	-	75	-	-	75.22 (-0.91)	92.43 (-0.43)
75	87.50	93.75	93.75	-	75.31 (-0.82)	92.50 (-0.36)
75	-	-	75	-	75.44 (-0.69)	92.63 (-0.23)
75	87.50	93.75	96.875	96.875	75.63 (-0.50)	92.67 (-0.19)

Table 6.5: Hierarchical block sparsity with varying distribution. (Cumulative-Sparsity=50)

6.3 Conclusion

In HBsNN models, levels with smaller block sizes cater for bridging accuracy gap and levels with larger block sizes cater for improving performance. Thus HBsNN models have better accuracies than highly structured sparse models and have better performance than unstructured sparse

Spar	sity	Accı	iracy
L1-(32x1)	L2-(1x1)	Top-1	Top-5
62.50	87.50	75.92 (-0.21)	92.79 (-0.07)
56.25	93.75	75.64 (-0.49)	92.62 (-0.24)
53	97	75.26 (-0.87)	92.48 (-0.38)

Table 6.6: Quasi block sparsity with varying distribution. (Cumulative-Sparsity=50)

models. This fluidity in structure in HBsNN models is essential to obtain better sparse models which are both accurate and performant.

Chapter 7

Regularized Multi Block Sparse Neural Networks

A large portion of the computation in sparse neural networks comprises of multiplying a sparse matrix with a dense matrix, denoted SDMM in this chapter. The SDMM operation with an unstructured sparsity pattern cannot be efficiently processed on modern architectures such as GPUs due to irregularity in compute and memory accesses. However, efficient parallel algorithms on a GPU can be designed for SDMM when the sparsity pattern is more structured. Thus, the run time performance of sparse neural networks on a GPU is dependent on the sparsity pattern present in the underlying matrix.

In sparse neural networks obtained using pruning based approaches, the choice of sparsity pattern not only effects the run time, but also effects the accuracy of the task for which the neural network is trained for. Sparsity patterns which have a good run time performance on a GPU may not have a good accuracy and vice-versa. The real challenge then is to, given a target architecture, identify a sparsity pattern, a storage format, and an algorithm for SDMM that leads to sparse neural networks which are efficient in both run time and accuracy.

Efficient parallel algorithms for sparse matrix operations can be designed when the sparsity pattern is known a-priori. The benefits of such an approach are, for instance, shown by Yang et al. [85] for multiplying a sparse matrix with a quasi-band sparsity with a dense vector, Vooturi and Kothapalli for multiplying two quasi-band matrices [80], and Picciau et al. for solving sparse triangular systems [70]. Such algorithms designed for specific classes of inputs outperform the corresponding general-purpose algorithms by even a factor of magnitude in some cases. This line of work is used in also parallel algorithms for various graph problems ranging from finding the strongly connected components [39], finding shortest paths and centrality values [16, 67], finding 2-connected components [24, 78], to name a few.

To design efficient parallel algorithms in the context of sparse neural networks, the effect of sparsity pattern on accuracy presents additional challenges when adapting the above mentioned line of work. In particular, given a degree of sparsity, the position of nonzero elements can impact the accuracy of the neural network. Therefore, on a given target architecture, sparsifying neural networks presents opportunities where an optimal sparsity pattern can be designed taking into account the effect of sparsity pattern on accuracy, and the characteristics of the target architecture.

7.0.1 Our Contributions

In this chapter, we aim to exploit these opportunities and propose to use a codesign approach that identifies a suitable sparsity pattern, a storage format, and the associated algorithms towards efficient sparse neural networks. To this end, we introduce a novel structured sparsity pattern, called the Regularized Multi-Block (RMB), that is highly suitable in the context of parallel algorithms for sparse neural networks on modern accelerator based architectures. We also show that using the RMB sparsity pattern we can reduce the amount of space needed to store a sparse matrix apart from allowing for efficient parallel algorithms for the SDMM operation. Following are our main contributions:

- We propose a novel, structured, flexible, and generic sparsity pattern called the RMB (Regularized Multi Block) sparsity pattern which results in more efficient sparse neural networks on a GPU.
- We develop an efficient GPU algorithm for an instance of the SDMM operation called RMBMM (matrix multiplication of a sparse matrix with a dense matrix, where the multiplicand has an RMB sparsity pattern). Our algorithm outperforms the generic CSRMM routine from cuSPARSE library by a factor of 6-9x (Figure 7.9).
- We evaluate the RMB sparsity pattern on an image classification task on the Imagenet dataset and generate sparse neural networks which are 4-5x and 2-3x more efficient in run time when compared to sparse neural networks with unstructured, and block sparsity patterns respectively (Figure 7.11).

Related work : On modern parallel architecture like GPU, one simple and effective way to overcome the performance issue of sparse neural networks is to resort to structured pruning methods. In structured pruning methods, pruning is performed with structured sparsity patterns like block, row, and column. Among them, pruning approaches which result in row and column sparsity



Figure 7.1: A multi-block matrix with two block matrices of block sizes (2,3) and (2,2).

patterns are deeply explored in [32, 50, 53, 57] due to the fact that SDMM operations with row or column sparsity pattern can be processed efficiently using existing fast dense kernels and does not require developing a new sparse kernel. But a major drawback with the row or column sparsity pattern is the rigidity with which pruning is performed. For a given sparsity, this rigidity limits the choice of elements to retain and thus results in sub-par sparse neural networks. When compared to the row and column sparsity patterns, block sparsity pattern is less rigid in nature. Narang et al. [64] explore a block sparsity pattern with a single block type. Vooturi et al. [81] explore a block sparsity pattern with a single block type. Vooturi et al. [81] explore a block sparsity pattern with multiple hierarchical block types. Both [64] and [81] result in sparse neural networks that either compromise on the accuracy or the regularity in computation, which is key for runtime performance. Our proposed RMB (Regularized Multi Block) sparsity pattern retains the accuracy while not compromising on the regularity in computation. In [14], bank balanced sparsity pattern, a subcase of RMB sparsity pattern is proposed to efficiently process sparse neural networks on a FPGA.

7.1 Preliminaries

7.1.1 Sparse Matrix Structures

In this section, we define our proposed sparse matrices using block matrix as the unit.

Block matrix (A_b) : A matrix is called a block matrix if the non-zero elements in the matrix are structured in the form of blocks of size (bh, bw) on a uniform matrix grid.

Multi Block Matrix (A_{mb}) : If a matrix can be expressed as a disjoint sum of block matrices $(A_{mb} = A_b^1 + \cdots + A_b^N)$, then such a matrix is called a multi-block matrix. Figure 7.1 shows an example of a multi-block matrix.

Blocklet Matrix (A_{bl}) : A blocklet matrix is a block matrix of block size (bh, bw) with following properties:



Figure 7.2: Negative and positive cases of a block matrix being a blocklet matrix. All the block matrices have block size 2x2.

- 1. The number of non-zero blocks is equal to the number of row blocks nrb, where $nrb = (A_{bl}.rows/A_{bl}.bh)$
- 2. Each row block has only one non-zero block.

Figure 7.2 shows negative and positive cases of a block matrix being a blocklet matrix. The block matrix in Figure 7.2a violates Property 1 as it has five non-zero blocks instead of four. The block matrix in Figure 7.2b violates Property 2 as the number of non-zero blocks in row blocks 0 and 2 is not one. Figures 7.2c and 7.2d shows valid cases of a block matrix being a blocklet matrix.

Multi Blocklet Matrix (A_{mbl}) : A matrix is called a multi-blocklet matrix if it can be expressed as a sum of blocklet matrices $(A_{mbl} = A_{bl}^1 + \dots + A_{bl}^N)$. A blocklet matrix is also a multi blocklet matrix with the number of blocklets equal to one.

Regularized Multi-Block Matrix (A_{rmb}): A regularized multi Block (RMB) Matrix is a block matrix where each block is either a multi-blocklet matrix or a zero matrix. Figure 7.3 shows an example of an RMB matrix with block size (8,8), where the three non-zero blocks are multiblocklet matrices.

7.1.2 Sparse Matrix Storage Formats

In this section, we propose our sparse matrix storage formats CBL,CMBL,CRMB for storing blocklet, multi blocklet, and regularized multi-block matrix respectively. **CBL**: The Compressed BlockLet (CBL) storage format is designed specifically for storing a blocklet matrix A_{bl} . It consists of a 2D array values of size $(A_{bl}.rows, A_{bl}.bw)$, and a 1D array *indices* of size equal to the number of row blocks $nrb (nrb = A_{bl}.rows/A_{bl}.bh)$. Non-zero blocks in A_{bl} are stored in values



Figure 7.3: A regularized multi block matrix (RMB) with block size (bh, bw) = (8, 8). Each block is either a multi-blocklet matrix or a zero matrix.

in such a way that k^{th} column in *values* is formed by concatenating the k^{th} column of all the non-zero blocks in the order of their row blocks. Similarly, the column block indices of the non-zero blocks in A_{bl} are stored in *indices* array in the order of their row blocks. In Figure 7.4, we can see that 0^{th} column of *values* for the second blocklet matrix i.e [2,6,4,8,6,2] is formed by concatenating 0^{th} columns of the three non-zero blocks i.e, [2,6], [4,8], and [6,2] in the order of their row blocks.

CMBL: The Compressed Multi-BlockLet (CMBL) storage format is used for storing a multiblocklet matrix A_{mbl} . As A_{mbl} is simply a collection of blocklet matrices, the CMBL format stores them as an array *bl_matrices* of type CBL. Figure 7.4 shows examples of multi-blocklet matrices stored in the CMBL storage format.

CRMB: The Compressed Regularized Multi-Block (CRMB) storage format is used for storing a regularized multi-blocklet matrix A_{rmb} . It consists of three arrays $mbl_matrices$, indices and rowBlockPtr. As all the non-zero blocks in A_{rmb} are multi-blocklet matrices, they are stored in $mbl_matrices$ array of type CMBL in the row major fashion. Similarly, the column block indices of non-zero blocks in A_{rmb} are stored in the *indices* array in the row major fashion. In rowBlockPtr, the i^{th} element points to the starting non-zero block of the i^{th} row block in the $bl_matrices$ array.

The CRMB data format is specifically designed to efficiently store and process sparse matrices with an RMB sparsity pattern. One could also use the generic CSR data format to store an RMB sparse matrix. While the storage cost of CSR depends only on the amount of sparsity, the storage



Figure 7.4: CRMB storage format for storing a RMB sparse matrix with block size (6,6). Each non zero block in the matrix is stored in CMBL storage format, which inturn uses CBL storage format to store blocklet matrices.

cost of the CRMB data format depends also on the blocklet configuration in RMB sparsity pattern. In Table 7.1, we compare storage costs of the CSR data format with the CRMB data format for three blocklet configurations (1,1), (2,2), and (4,4). From Table 7.1, we can see that for the finest possible blocklet configuration i.e, (1,1), the CRMB storage cost is within a meagre 5% of CSR storage cost and for coarser blocklets, storing an RMB sparse matrix in the CRMB data format can reduce storage costs by a factor of $\sim 2x$ when compared to the CSR storage format. This is due to the fact that for higher blocklet types, only one index needs to be stored for a single block.

Storage format	sparsity = 50%	sparsity = 75%	sparsity = 87.5%
CSR	64.02	32.02	16.02
CRMB-(1,1)	66.25	33.25	16.75
CRMB-(2,2)	41.25	20.75	17.50
CRMB-(4,4)	34.75	17.50	8.88

Table 7.1: Storage cost (in MB) of the CRMB and the CSR data formats for storing a sparse matrix of size 4096×4096 with an RMB sparsity pattern. CRMB-(x,y) corresponds to an RMB sparsity pattern with blocklets of type (x,y)).

7.2 Our Algorithm for RMBMM

In this section, we show our algorithm for RMBMM by building upon algorithms for multiplying a blocklet sparse matrix with a dense matrix (BLMM) and for multiplying a multi-blocklet sparse matrix with a dense matrix (MBLMM).

7.2.1 BLMM

The BLMM operation $(CT = AT_{bl} \times BT)$ is an instance of the SDMM operation with the multiplicand having a blocklet sparstiy pattern. The BLMM operation is processed according to Algorithm 15, with AT_{bl} stored in the CBL storage format. The outer for loop in Algorithm 15 runs for $AT_{bl}.bw$ steps and in k^{th} step, the computation associated with the k^{th} column of all the nonzero blocks in AT_{bl} is processed.

In a given step, multiple rows from B may be accessed as non-zero blocks in A_{bl} may not be in the same column block. Lines 4-6 describe the mapping between an element in the k^{th} column of $A_{bl}.values$ and its choice of row in B. First, the row block into which the element belongs is calculated and then its associated column block index is accessed from $A_{bl}.indices$. Subsequently, the index of the associated row in B is calculated using the column block index of the element and the current step number. The time complexity of the BLMM operation using Algorithm 15 is $O(AT_{bl}.bw \times AT_{bl}.rows \times BT.columns)$. Our algorithm is efficient as it does not process computation corresponding to zero blocks. But if AT_{bl} were to be stored as a dense matrix and processed as a dense matrix, the complexity would be $O(AT_{bl}.columns \times AT_{bl}.rows \times BT.columns)$ irrespective of the block size in AT_{bl} .

Algorithm 15 Matrix multiplication of a blocklet matrix AT_{bl} with a dense matrix BT. ($CT = AT_{bl} \times BT$).

1:	for k in $[0:AT_{bl}.bw)$ do
2:	for i in $[0:AT_{bl}.rows)$ do
3:	for j in $[0:BT.columns)$ do
4:	$rbInd = i/AT_{bl}.bh$
5:	$cbInd = AT_{bl}.indices[rbInd]$
6:	$rowB = (cbInd \times AT_{bl}.bw) + k$
7:	$CT[i][j] + AT_{bl}.values[i][k] \times BT[rowB][j]$
8:	end for
9:	end for
10:	end for

7.2.2 MBLMM

The MBLMM operation $(CT = AT_{mbl} \times BT)$ is an instance of the SDMM operation with the multiplicand having a multi-blocklet sparsity pattern. As AT_{mbl} matrix is a sum of blocklet matrices $(AT_{mbl} = \sum_{n=1}^{n=N} AT_{bl}^n)$, the MBLMM operation involving AT_{mbl} can be expressed as a sum of sub-products $(CT = \sum_{n=1}^{n=N} AT_{bl}^n \times BT)$ where each sub-product corresponds to a BLMM operation. Algorithm 16 is used to process the MBLMM operation with AT_{mbl} stored in the CMBL format.

Algorithm 16 Matr	ix multiplication	of a multi-block	tlet matrix AT_m	bl with a dense	e matrix B
$(CT = AT_{mbl} \times BT_{mbl})$	Г).				

1: $num_blocklets = AT_{mbl}.bl_matrices $	
2: for n in $[0, num_blocklets)$ do	Loop through blocklets
3: $AT_{bl} = AT_{mbl}.bl_matrices[n]$	
4: $CT \neq AT_{bl} \times BT$	Process using Algorithm 15
5: end for	

7.2.3 **RMBMM**

RMBMM ($C = A_{rmb} \times B$) is an instance of the SDMM operation with the multiplicand having an RMB sparsity pattern. For processing the RMBMM operation, we use a tiling based approach similar to the one used for efficiently processing GEMM (dense matrix multiplication) operation [65]. In the tiling approach, the matrix C is divided into tiles of size (TM, TN). Each output tile CT in C is then processed in steps. In each step a tile AT of size (TM, TK) from A_{rmb} and its corresponding tile BT of size (TK, TN) from B are multiplied together to produce a partial output of CT. For a given hardware, tiling parameters TM, TK and TN are chosen to maximize performance.

Algorithm 17 describes the RMBMM operation using the tiling approach with A_{rmb} stored in the CRMB storage format. In RMBMM, tiling parameters TM and TK are set to $A_{rmb}.bh$ and $A_{rmb}.bw$ respectively. Lines 6-17 in Algorithm 17 describe the computation required for processing an output tile in C with index (rb, cb). As each output tile is computed in steps, the number of steps required for processing an output tile $CT_{rb,cb}$ is equal to the number of non-zero tiles/blocks in the row block rb of A_{rmb} . In each step, an MBLMM operation is performed as a tile/block in A_{rmb} is a multi-blocklet matrix. Our algorithm is optimal as we do not process computation associated with zero tiles in A_{rmb} and uses an efficient MBLMM operation for processing non-zero tiles.
Algorithm 17 Tiled Matrix multiplication of a regularized multi-block matrix A_{rmb} with a dense matrix B ($C = A_{rmb} \times B$) parameterized by TN.

1:	$TM = A_{rmb}.bh$	
2:	$TK = A_{rmb}.bw.$	
3:	nrb = C.rows/TM	▷ Number of row blocks in C
4:	ncb = C.cols/TN	▷ Number of column blocks in C
5:	for rb, cb in $[0: nrb) \times [0: ncb)$ do	
6:	$rowRangeA = [rb \times TM : (rb + 1) \times TM)$	
7:	$colRangeB = [cb \times TN : (cb + 1) \times TN)$	
8:	$start_mid = A_{rmb}.rowBlockPtr[rb]$	
9:	$end_mid = A_{rmb}.rowBlockPtr[rb + 1] - 1$	
10:	for mid in [$start_mid$: end_mid] do	
11:	$AT_{mbl} = A_{rmb}.mbl_matrices[mid]$	⊳ A tile
12:	$ind = A_{rmb}.indices[mid]$	
13:	$rowRangeB = [ind \times TK : (ind + 1) \times TK)$	
14:	BT = B[rowRangeB][colRangeB]	⊳ B tile
15:	$CT += AT_{mbl} \times BT$	Process using Algorithm 16
16:	end for	
17:	C[rowRangeA][colRangeB] = CT	
18:	end for	

7.3 RMBMM on a GPU

In this section, we first show how the RMBMM operation can be efficiently mapped onto a GPU using our GPU algorithm RMBMM-GPU. Later we explore the space of RMB sparsity patterns and study its effect on the run time of the RMBMM operation on GPU.

7.3.1 RMBMM-GPU

The RMBMM operation can be efficiently implemented on a GPU by exploiting the properties of RMB sparsity pattern. In Algorithm 17, the outer for loop which runs through all the output tiles in C can be parallelized. So on a GPU, we map the computation of an output tile CT in C to a thread block. From Line 15 of Algorithm 17, we can infer that a thread block has to perform a series of MBLMM operations, $(AT_{mbl} \times BT)$, where each operation computes a partial output of CT. As elements of AT_{mbl} and BT are repeatedly accessed in the MBLMM operation, their values are loaded onto the shared memory before performing the operation. For processing the MBLMM operation, we use Algorithm 18, a reframed version of Algorithm 16 and show how it can be mapped onto a GPU. In Algorithm 18, the output tile CT is further tiled into subtiles, where each subtile corresponds to a group of elements of size (GR, GC). The for loop in Line 3 of Algorithm 18 which runs through all the element groups in CT can be parallelized. So on a GPU, we map the computation corresponding to an element group to a thread. For a given thread, we further avoid multiple accesses to shared memory by loading them once to the local variables and reuse multiple times. Shared memory requirement of our algorithm is $TK \times (TM + TN)$ words and registers required for storing operands and output in Line 19 is $GR + GC + GR \times GC$ words. As each SMX has a limited amount of shared memory and registers, we choose TN, GR, GC in such a way that it maximizes run time efficiency. Note that the value of GR should divide the block height of all the blocklets in the matrix A_{rmb} . This is because, for the Algorithm 18 to work, the values loaded into Areg in Algorithm 18 should come from a single block.

7.3.2 Experimental Platform

We use an Nvidia Volta V100 GPU with 16GB of RAM for all our experiments. The V100 has a total of 5120 cores organized in 80 streaming multi-processors (SMX). It has a peak single precision floating point performance of 15.7 Tflops and a peak memory bandwidth of 900 GBps. Each SMX also has a combined data cache and shared memory of 128KB, which can be configured as per requirements of the workload. For programming the V100 GPU, we use CUDA Version 9.0.

7.3.3 Exploring the RMB Sparsity Pattern Space

For a given amount of sparsity, the run time of an RMBMM operation $(C = A_{rmb} \times B)$ depends on many parameters like the block size in RMB matrix, the number of blocklet matrices per block, the type of blocklet for each blocklet matrix etc. In this section, we explore these factors and how they effect performance of the RMBMM operation on a GPU. We set the dimensions of all the matrices in our experiments to be 4096×4096.

7.3.3.1 Varying Block Size

As the block size is a significant aspect of the RMB sparsity pattern, we study the effect of block size (bh, bw) in the A_{rmb} matrix on the performance of Algorithm RMBMM-GPU. Each non-zero block in A_{rmb} is set to be a multi-blocklet matrix with only one blocklet matrix of type (bh, bw). We perform the following two experiments.

Algorithm 18 Grouping based matrix multiplication of a multi-blocklet matrix AT_{mbl} with a dense matrix $(CT = AT_{mbl} \times BT)$) parameterized by grouping parameters GR, GC.

```
1: nrg = CT.rows/GR
 2: ncg = CT.columns/GC
 3: for rg, cg in [0:nrg) \times [0:ncg) do
       Areg[GR], Breg[GC], Creg[GR][GC]
                                                                                        ▷ Registers
 4:
       Creg = 0
 5:
       num\_blocklets = |AT_{mbl}.bl\_matrices|
 6:
       for n in [0, num\_blocklets) do
 7:
           AT_{bl} = AT_{mbl}.bl\_matrices[n]
 8:
           for k in [0: AT_{bl}.bw) do
 9:
               rbInd = (rg \times GR)/AT_{bl}.bh
10:
               cbInd = AT_{bl}.indices[rbInd]
11:
               rowB = (cbInd \times AT_{bl}.bw) + k
12:
               for gr in [0:GR) do
13:
14:
                   Areg[gr] = AT_{bl}.values[rg \times GR + gr][k]
               end for
15:
               for gc in [0:GC) do
16:
                   Breg[gc] = BT[rowB][cg \times GC + gc]
17:
18:
               end for
19:
               Creg += Areg \times Breg
           end for
20:
       end for
21:
22:
       rgRange = [rg \times GR : (rg + 1) \times GR)
       cgRange = [cg \times GC : (cg + 1) \times GC)
23:
24:
       CT[rgRange][cgRange] += Creg
25: end for
```

In the first experiment, we study the effect of increasing the block size from 4×4 to 64×64 in multiples of 2. The grouping parameters GR and GC are set to 1. In Algorithm RMBMM-GPU, the tiling parameters TM and TK are set to bh and bw respectively and the tiling parameter TN is chosen to maximize occupancy for a given block size. From Figure 7.5a we can see that the time taken by Algorithm RMBMM-GPU decreases with increase in block size. This is because as the block size increases, the number of times an element has to be loaded into shared memory decreases and thus increases performance.

In the second experiment, we study the effect of changing the block size from a skinny block with block width 2 to a fat block with block width 64 in multiples of 2. We use three configurations for (TM,TN) = [(32,32), (64,64), (128,64)]. The grouping parameters GR and GC are each set to 4 resulting in a 16 way grouping. Sparsity is set to 50%. From Figure 7.5b, we see that as block width or TK increases, the run time decreases and then starts to increase. This is because of the trade-off between the number of steps taken and the amount of shared memory required. When TK is reduced by 2, it decreases the shared memory requirement by half. This helps in increasing occupancy. But this also increases number of steps and synchronization operations.

So in order to get performance for RMBMM-GPU operation, the choice of the block size should be big and not skinny.



(a) Varying block size for a given sparsity.

(b) Varying block width (TK) from skinny to bulky.

Figure 7.5: Varying block size in RMB sparsity pattern.

7.3.3.2 Varying Grouping

Grouping is parameterized by two parameters namely GR and GC. In grouping, $GR \times GC$ elements in C matrix are mapped onto a single thread. To study the effect of grouping on Algorithm RMBMM-GPU, we perform the following two experiments.

In the first experiment, we focus on grouping and vary grouping from 1 to 64 in multiples of 2 with grouping performed only along the column axis i.e, GR = 1. We choose the following three tiling configurations TM,TK,TN = [(32,32,32), (64,64,64), (128,32,128)]. Sparsity is set to 50%. From Figure 7.6a, we can see that as we increase grouping, the runtime decreases up to a certain point and then increases. This is because the number of registers required increases with grouping and this hampers occupancy of the SMX on a GPU.

In the second experiment, we fix the amount of grouping to 16 and change the layout in which grouping is applied. The total number of registers required in Algorithm RMBMM-GPU is GR+GC+GR×GC (beyond any other additional registers used), and the number of operations performed is GR×GC before resetting register buffers. When grouping is (1,16), 33 registers are required for doing 16 operations whereas only 24 registers are required for doing the same 16 operations when grouping is (4,4). If the total grouping amount is *G*, then using a grouping of (\sqrt{G}, \sqrt{G}) results in optimal register usage. But grouping in 2D can result in increased bank conflicts while accessing elements from shared memory. From Figure 7.6b, we can see the tradeoffs between 2D grouping and decreasing bank conflicts. In case of (128,32,128) tiling configuration, (4,4) layout has the best performance. This is because square layout is optimal and as each warp takes care of a single column in *CT*, it has no bank conflicts in accessing elements of *BT* from shared memory.

From the above experiments, we conclude that limited grouping is good. Further, grouping layout should to be chosen to optimize register usage and minimize shared memory bank conflicts.

7.3.3.3 Varying Blocklet Matrix

The type of a blocklet matrix is defined by its block size. In this experiment, we study how the choice of a blocklet matrix of a given type effect the run time of the RMBMM-GPU Algorithm. For a given blocklet matrix A_{bl} of type (bh, bw), there are $(A_{bl}.cols/bw)^{(A_{bl}.rows/bh)}$ possible blocklet matrices of that type. Among those possibilities, we choose three representative blocklet matrix have zero column block index. In *diagonal*, the column block index of a non zero block is equal to its row block index. And in *random*, column block indices for non zero blocks are chosen at



Figure 7.6: Varying grouping in RMBMM-GPU Algorithm.

random. The block size of A_{rmb} is set to (128,32) and the tiling parameter TN is set to 64. We perform a 32 way grouping with a (4,8) layout. Each block in A_{rmb} is set to be a multi-blocklet matrix with only one blocklet matrix of type (bh, bw). From Figure 7.7, we can see that for a given blocklet type (bh, bw), all the three representative patterns have the same run time. This implies that for a given blocklet type, the performance of RMBMM-GPU is invariant to the choice of blocklet matrix with that type. The reason for this invariance is because we load tiles from A_{rmb} and B into shared memory and all accesses for processing any blocklet matrix of a given type are made from the shared memory.

7.3.3.4 Varying Multi-Blocklet Matrix

In this experiment, we study how the choice of a multi-blocklet matrix effects the run time of the RMBMM-GPU Algorithm. Let A_{mbl} be a multi-block matrix of size (128,32). A_{mbl} can have N blocklets and each blocklet can be of any type. If A_{mbl} has 50% sparsity, then some of the possible combinations are: 1 blocklet matrix of type (64,16), two blocklet matrices of type (32,8) etc. From Figure 7.8, we can see that for a given sparsity run time increases with increase in number of blocklets in multi-block matrices. The reason for that is that more blocklets leads to more data loads due to increased number of indices and increased number of iterations in the algorithm.



Figure 7.7: Varying blocklet matrix for a givenFigure 7.8: Varying multi blocklet matrix con-
figuration.

7.3.4 Comparing RMBMM with CSRMM and BSRMM

Given an RMBMM operation $(C = A_{rmb} \times B)$, it can be efficiently processed on a GPU using our RMBMM-GPU Algorithm described in Section 7.3.1. We compare our RMBMM-GPU approach with following two benchmarks namely CSR-B and BSR-B.

- CSR-B: Any sparse matrix can be stored in CSR storage format irrespective of its sparsity pattern. In this benchmark, we store sparse matrix A_{rmb} in CSR format and process RMBMM operation using CSRMM API call from cuSPARSE library.
- BSR-B: As an RMB matrix is also a multi block matrix, it can be expressed as a sum of block matrices (A_{rmb} = A¹_b + ··· + A^N_b). Thus the RMBMM operation can be expressed as a sum of SDMM operations with block sparsity pattern (C = ∑^{i=N}_{i=1} Aⁱ_b × B). In this benchmark, we process each partial product Aⁱ_b × B using the BSRMM API call from the cuSPARSE library and accumulate the run times.

For A_{rmb} , we set the block size to (128, 32) and generate random RMB matrices with blocklet types (64,16), (32,8), (16,4), (8,2), (4,1). For each non-zero block, the number of blocklets and the types of the blocklets are chosen at random. From Figure 7.9, we see that processing the RMBMM operation as a CSRMM or a BSRMM leads to large penalties in terms of run time. This corroborates our idea of using a codesign approach of choosing the sparsity pattern (RMB), storage format (CRMB) and the associated GPU algorithm (Algorithm RBMMM-GPU).



Figure 7.9: Speedup of RMBMM wrt CSR-B and BSR-B.

7.4 Regularized Multi Block (RMB) Sparse Neural Networks

In this section, we generate sparse neural networks with RMB sparsity pattern and compare them with unstructured, and block sparse neural networks on the metrics of accuracy, and run time. To that end, we consider two AI tasks, namely image classification and image segmentation.

7.4.1 Generation of Sparse Neural Networks

Sparse neural networks are generated using the pruning approach [29] from a pre-trained dense neural network. In a pruning approach, unimportant parameters from each layer of a pre-trained dense network are pruned/removed based on some criterion. The resultant sparse network is then retrained to recover the lost accuracy from pruning. Specifics on how the pruning approach is applied with unstructured, block, and RMB sparsity patterns for a given layer are described below. For a given layer, let A be the dense parameter matrix to be pruned in that layer and N be the number of parameters or size of A.

7.4.1.1 Unstructured Pruning

Parameters are pruned at an individual level based on their magnitude. Given the amount of pruning/sparsity percentage sp, parameters are first sorted and then the smallest sp% of the parameters are pruned. We store the obtained sparse matrix in CSR storage format and process the SDMM operation using the CSRMM API call from the cuSPARSE library.

7.4.1.2 Block Pruning

Parameters are pruned in the units of blocks of size (BH, BW). In order to prune the blocks, we first quantify the strength of a block by summing up the magnitude of elements in that block. Once block strengths are computed, blocks are sorted and the smallest sp% of the blocks are pruned. The sparse matrix obtained from block pruning is stored in the BSR format and the SDMM operation is processed using the BSRMM API call from cuSPARSE.

7.4.1.3 RMB Pruning

The core operation in RMB pruning is to prune a dense block Ab of size (BH, BW) in A into a multi-blocklet matrix Ab_{mbl} with K blocklet matrices $Ab_{bl}^1, \ldots, Ab_{bl}^K$ of types (blh_1, blw_1) , $\ldots, (blh_K, blw_K)$ respectively. The pseudo-code for RMB pruning is provided in Algorithm 19. The outer for loop goes through all the dense blocks in A. And for a given dense block Ab, the multi-blocklet matrix Ab_{mbl} is generated in K steps, where in the i^{th} step, the blocklet matrix Ab_{bl}^i is generated in K steps, where in the i^{th} step, the blocklet matrix Ab_{bl}^i is generated from the current residual matrix Ab_{res} using Algorithm 20 (Also see Figure 7.10). The residual matrix Ab_{res} is initialized to Ab and the multi-blocklet matrix Ab_{mbl} is initialized to 0. In each step, Ab_{res} and Ab_{mbl} are respectively updated by subtracting and adding the generated blocklet matrix in that step. The resultant sparse matrix obtained after RMB pruning is stored in the CRMB format and is processed using our RMBMM-GPU Algorithm.

The time complexity of Algorithm 20 is linear in the size of the input matrix as it consists of only piece-wise *reduction* operations for calculating block strengths and max operations for choosing the blocklet pattern. As Algorithm 20 is invoked K times for each dense block in A, the total time complexity of RMB pruning is $O(K \times N)$. As K is typically a small number, the time complexity of RMB pruning is linear in the size of the input matrix A.

7.4.2 Image Classification

In Image classification, the task of a neural network is to correctly classify the image into one of the predefined classes. For our experiments, we use three neural network architectures, namely VGG11 [77], Resnset18 [31] and Resnet50 [31] architectures and two datasets, namely CIFAR-100 [46] and ILSVRC2012 [75] datasets. Multiple sparse neural networks with different sparsity amounts are generated for each sparsity pattern using corresponding pruning methods described in Section 7.4.1. For block pruning, we set the block size to 4x4 so as to avoid generating sparse

Ab _{res}						Block	Ab _{bl}				
3	7	8	9	2	8	Strengths			8	9	
6	4	6	2	4	1	202515			6	2	
1	3	2	4	2	1	1510 5	1	3			
2	9	1	3	1	1	102015	2	9			
3	2	6	4	3	2	···			6	4	
2	3	2	8	7	3				2	8	

Figure 7.10: Separation of a blocklet matrix Ab_{bl} of type (2,2) from an input matrix Ab_{res} of size (6,6). Block strengths are first calculated for each 2x2 block in A_{res} and a block with maximum strength is chosen for each of the three row blocks.

Algorithm 19 RMB pruning of a dense parameter matrix A into a sparse matrix A_{rmb} with RMB sparsity pattern. Each dense block Ab of size $(BH \times BW)$ is pruned into a multiblocklet matrix Ab_{mbl} composed of K blocklet matrices $Ab_{bl}^1, \ldots, Ab_{bl}^K$ with blocklet types $(blh_1, blw_1), \ldots, (blh_K, blw_K)$.

1: nrb = A.rows/BH2: ncb = A.cols/BW3: $A_{rmb} = 0$ 4: for (rb, cb) in $[0, nrb) \times [0, ncb)$ do $rRange = [rb \times BH : (rb + 1) \times BH)$ 5: $cRange = [cb \times BW : (cb + 1) \times BW)$ 6: Ab = A[rRange][cRange]7: ▷ Dense block $Ab_{res} = Ab$ Residual matrix initialization 8: 9: $Ab_{mbl} = 0$ Multi-blocklet matrix initialization for i in [1:K] do 10: $Ab_{bl}^{i} = separate_blocklet(Ab_{res}, blh_{i}, blw_{i})$ 11: $Ab_{res} = Ab_{res} - Ab_{bl}^i$ 12: ▷ Update residual matrix $Ab_{mbl} += Ab_{bl}^{i}$ ▷ Update multi-blocklet matrix 13: end for 14: $A_{rmb}[rRange][cRange] = Ab_{mbl}$ 15: 16: end for 17: return A_{rmb}

Algorithm 20 Separating blocklet matrix Ab_{bl} of type (BLH, BLW) from input matrix Ab_{res} .

1: separate_blocklet(Ab_{res} , BLH, BLW) : 2: $nrb = Ab_{res}.rows/BLH$ ▷ Number of row blocks 3: $ncb = Ab_{res}.cols/BLW$ ▷ Number of column blocks 4: $b_strengths = zeros(nrb, ncb)$ ▷ Block strengths 5: for (rb, cb) in $[0: nrb) \times [0: ncb)$ do $rRange = [rb \times BLH : (rb + 1) \times BLH)$ 6: $cRange = [cb \times BLW : (cb + 1) \times BLW)$ 7: $b_strengths[rb][cb] = sum(|M[rRange][cRange]|)$ 8: 9: end for 10: $Ab_{bl} = 0$ Blocklet matrix initialization 11: **for** rb in [0:nrb) **do** $m_c b = max_i nd(b_s trengths[rb][:]) \triangleright$ Index of the block with maximum strength in row 12: block rb. 13: $rRange = [rb \times BLH : (rb + 1) \times BLH)$ $cRange = [m_cb \times BLW : (m_cb + 1) \times BLW)$ 14: $Ab_{bl}[rRange][cRange] = Ab_{res}[rRange][cRange]$ 15: 16: end for 17: return Ab_{bl}

models with high accuracy loss. For RMB, we set the block size to be 128x32 and limit blocklet matrices to be of type (1,1).

Training setup: CIFAR100 dataset has a total of 60K images coming from 100 image classes. Out of which, we use 50K images for training, and 10K images for testing. For CIFAR100, we also augment the data for training, by taking a 32x32 random crop from a 4 padded 32x32 original image or its horizontal flip. For networks trained over CIFAR100, the base dense network is trained for 120 epochs. For VGG11 and Resnet18, the pruned model is retrained for 16 epochs with initial learning rate set to 0.01. Learning rate is decreased by a factor of 10 at epochs 8 and 12.

The ILSVRC12 dataset has 1.2 million images coming from 1000 image classes. Out of them 50K images are used for testing and the rest are used for training. For data augmentation in training networks with ILSVRC12, a random 224x224 crop is taken from the resized original image or its horizontal flip. For Resnet50, the base dense network is trained for 100 epochs. Initial learning rate is set to 0.1 and is decreased by a factor of 10 at epochs 30, 60 and 90. For Resnet50, the pruned model is retrained for 16 epochs with initial learning rate set to 1e-3. The learning rate is decreased by a factor of 10 at epochs 8 and 12. For the loss function, we use the cross entropy loss, and for the optimization, we use the stochastic batched gradient descent optimizer with batch

	sparsity = 50%		sparsity =	75%	sparsity = 87.5%		
Sparsity	Accuracy	Time	Accuracy	Time	Accuracy	Time	
pattern		(ms)		(ms)		(ms)	
Unstructured	68.46	132	67.73	86	66.81	56	
Block	67.31	64	65.63	35	64.11	19	
RMB	68.57	25	68.04	15	66.38	10	

Table 7.2: VGG11 network over CIFAR100 dataset. Accuracy of the pretrained dense model is 68.58%.

size 256, weight decay of 1e-4, and momentum of 0.9. For all the considered networks, namely VGG11, Resnet18, and Resnet50 all the layers are pruned except for the first convolutional layer and the last fully connected layer, which are directly connected to input and output respectively. Run time is reported for processing a 64 batched input.

7.4.2.1 VGG11/CIFAR-100

From Table 7.2, we can see that when compared to unstructured models, RMB models have the same level of accuracy and are 5-6x faster for the same number of parameters. And when compared to block models, RMB models are more accurate and are up to 2-3x faster for the same number of parameters. From Table 7.2, we can also see that despite having 2x less number of parameters, RMB model with 87.5% sparsity is more accurate and is 3.5x faster than block model with 75% sparsity.

7.4.2.2 Resnet18/CIFAR100

From Table 7.3, we can see that when compared to unstructured models, RMB models achieve similar accuracy levels and are 3x faster. Further as we increase sparsity, the accuracy gap between block and RMB increases and this shows the inability of block pruning to generate sparse neural networks with high degree of sparsity. From Table 7.3, we can see that for the same level of accuracy, RMB model with 75% sparsity takes 2x less parameters and is 2.8x faster when compared to block model with 50% sparsity.

	sparsity = 50%		sparsity =	75%	sparsity = 87.5%		
Sparsity	Accuracy	Time	Accuracy	Time	Accuracy	Time	
pattern		(ms)		(ms)		(ms)	
Unstructured	72.92	58	71.95	42	71.11	30	
Block	71.57	36	69.88	20	67.32	11.2	
RMB	72.64	20	71.84	13	70.86	9.4	

Table 7.3: Resnet18 network over CIFAR100 dataset. Accuracy of the pretrained dense model in 72.9%.

7.4.2.3 Resnet50/ILSVRC12

The ILSVRC12 dataset is a strong test for our proposed RMB sparsity pattern as it has 20x more images and 10x more classes than that of CIFAR100 dataset. From Figure 7.11, we see that when compared to block sparsity, RMB models are more accurate and are 2-3x faster for the same number of parameters. And also when compared to unstructured models, RMB models are 4-5x faster and have the same level of accuracy except for the case when sparsity is 87.5%. This is because when sparsity becomes very high, the effect of structural constraints in RMB sparsity pattern on accuracy becomes more acute.



Figure 7.11: Resnet50 network over ILSVRC12 dataset. Accuracy of the dense model is 76.13%.

	sparsity = 50%		sparsity =	62.5%	sparsity = 75%		
Sparsity	Accuracy	Time	Accuracy	Time	Accuracy	Time	
pattern		(ms)		(ms)		(ms)	
Unstructured	55.71	550	55.37	441	53.64	373	
Block	51.25	331	48.5	258	39.9	192	
RMB	56	123	54.87	103	52.74	84	

Table 7.4: FCN-Resnet50 network over VOC2012 dataset. Accuracy of the pretrained dense model is 56.12%.

7.4.3 Image Segmentation

In image segmentation, the task of a neural network is to correctly classify each individual pixel in an image into one of the predefined classes. For our experiments we use two FCN (Fully Convoluted Networks), FCN-Resnet50, and FCN-Resnet101 over PASCAL VOC12 dataset.

Training setup: The VOC12 dataset has a total of 2931 images with pixels labelled into one of the 20 predefined classes. Of there, half of the images are used for training and other half for testing. For data augmentation, a random 480x480 crop is taken from the resized original image or it's horizontal flip. For FCN-Resnet50 and FCN-Resnet101, the base network is trained for 30 epochs with initial learning rate set to 0.01. For the loss function, we use cross entropy loss and for optimizer, we use batched stochastic gradient descent optimizer with batch size 8, weight decay 1e-4, and momentum of 0.9. The pruned network is retrained for 10 epochs with initial learning rate set to 1e-3. For both FCN-Resnet50 and FCN-Resnet101, we prune all the layers except for the first convolutional layer connected to the input and last two convolutional layers before the output. Run time is reported for processing 4 batched input.

7.4.3.1 FCN-Resnset {50,101}/VOC12:

From Tables 7.4 and 7.5, we see that for the same number of parameters, RMB sparse models are 4-5x and 2-3x faster than unstructured and block sparse models respectively. It is also clear that the rate at which accuracy drops with respect to sparsity is very high for block sparsity pattern when compared to unstructured and RMB patterns. At higher sparsities, RMB performs slightly worse than unstructured. This is due to the fact that any kind of structured pattern cannot be as flexible as unstructured pattern.

	sparsity = 50%		sparsity =	62.5%	sparsity = 75%		
Sparsity	Accuracy	Time	Accuracy	Time	Accuracy	Time	
pattern		(ms)		(ms)		(ms)	
Unstructured	59.92	1876	59.45	1596	58.72	1314	
Block	56.3	1133	53.87	840	47.08	654	
RMB	59.73	363	59.33	299	57.67	235	

Table 7.5: FCN-Resnet101 network over VOC2012 dataset. Accuracy of the pretrained dense model is 60.09%.

7.4.4 Exploring the RMB Sparsity Pattern Space

The RMB sparsity pattern offers a vast space of configurations and the choice of configuration has varied effects on run time and accuracy of the sparse model. In Section 7.3.3, we have explored the space of RMB sparsity pattern from the viewpoint of run time performance and in this section we will explore RMB sparsity pattern from the viewpoint of accuracy.

7.4.4.1 Varying Block Size

In this experiment, we study the effect of varying block size in RMB sparsity pattern on accuracy and run time of RMB sparse models. Using RMB pruning (Algorithm 19), we generate sparse models with an RMB sparsity pattern for block sizes 8x8, 16x16, and 32x32 for different sparsity amounts. In the imposed RMB sparsity pattern, we limit blocklet matrices to be of type (1,1). From Tables 7.6, 7.7, 7.8, and 7.9 we can see that for a given sparsity, the accuracy of the model is robust to block size and as the block size increases, run time performance improves significantly. This robustness to the block size allows us to choose higher block size in the RMB sparsity pattern and improve run time performance without any loss in accuracy.

	sparsity = 50%		sparsity = 75%		sparsity $= 87.5\%$	
Block size	Accuracy	Time	Accuracy	Time	Accuracy	Time
		(ms)		(ms)		(ms)
8x8	68.26	62	67.38	51	66.36	45
16x16	68.51	45	67.65	32	66.65	26
32x32	68.45	37	67.95	25	66.42	19

Table 7.6: Effect of varying block size in RMB on accuracy and runtime of VGG11 sparse model.

	sparsity = 50%		sparsity = 75%		sparsity $= 87.5\%$		
Block size	Accuracy	Time	Accuracy	Time	Accuracy	Time	
		(ms)		(ms)		(ms)	
8x8	72.78	40	71.51	33	70.68	29	
16x16	72.83	31	72.07	23	70.82	19.2	
32x32	72.63	24	71.85	17	70.61	13.7	

Table 7.7: Effect of varying block size in RMB on accuracy and runtime of Resnet18 sparse model.

	sparsity = 50%		sparsity =	62.5%	sparsity = 75%		
Block size	Accuracy	Time	Accuracy	Time	Accuracy	Time	
		(ms)		(ms)		(ms)	
8x8	55.65	302	54.75	274	51.85	244	
16x16	56.07	230	54.91	200	52.41	171	
32x32	56	188	54.87	163	52.74	137	

Table 7.8: Effect of varying block size in RMB on accuracy and runtime of FCN-Resnet50 sparse model.

7.4.4.2 Varying Multi-Blocklet Matrix

In this experiment, we study how the choice of a multi-blocklet matrix in RMB sparsity pattern effects the accuracy and run time of the model. We fix the block size in RMB sparsity pattern to be 128x32 and limit a multi-blocklet matrix to have up to two unique blocklet types. Using RMB pruning, we generate sparse models for different multi-blocklet configurations across different sparsity amounts. From Table 7.10,7.11, 7.12, and 7.13 we can see that for a given sparsity as we move towards coarse blocklet types, the accuracy of the model decreases and run time performance improves. The accuracy decreases because the flexibility in pruning decreases with increase in the coarsity of blocklet types and the run time performance increases because coarser blocklets offer more regularity in computation and decreases the number of blocklets in a multi-blocklet matrix.

The space of RMB sparsity pattern offers rich structures which can be configured according to the runtime and accuracy needs of the usecase. As a general rule of thumb, use coarser blocklet types for maximizing runtime performance, and finer blocklet types for maximizing accuracy.

	sparsity = 50%		sparsity =	62.5%	sparsity = 75%		
Block size	Accuracy	Time	Accuracy	Time	Accuracy	Time	
		(ms)		(ms)		(ms)	
8x8	60.01	1013	59.01	916	57.31	811	
16x16	59.94	756	59.54	657	57.52	560	
32x32	59.93	608	59.44	521	57.74	433	

Table 7.9: Effect of varying block size in RMB on accuracy and runtime of FCN-Resnet101 sparse model.

	sparsity = 50%		sparsity = 75%		sparsity = 87.5%	
Blocklet	Accuracy	Time	Accuracy	Time	Accuracy	Time
types		(ms)		(ms)		(ms)
1x1	68.57	24.56	68.04	14.70	66.38	9.91
2x2	67.56	17.72	66.30	11.65	64.53	8.46
4x4	66.90	14.91	66.18	10.09	64.02	7.74
2x1,1x1	68.38	24.74	67.58	14.66	66.15	9.89
4x2,2x1	68.24	20.38	66.74	12.94	65.11	9.13
8x2,4x2	67.05	17.23	65.87	11.31	64.71	8.27

Table 7.10: Effect of varying blocklet types in RMB on accuracy and runtime of VGG11 sparse model.

7.5 Conclusion

Due to the dual effect of a sparsity pattern on the accuracy and runtime of a sparse neural network, we followed a codesign approach, where the sparsity pattern (RMB), its storage format (CRMB), and a corresponding algorithm (RMBMM-GPU) for SDMM are designed in tandem to result in sparse neural networks that are both accurate, and performant on a GPU. The space of the proposed RMB sparsity pattern is rich and can be configured according to the runtime and accuracy needs of an application.

Inorder to induce an RMB sparsity pattern, we proposed a simple homogeneous pruning approach (RMB pruning), where all the non zero blocks in RMB have the same multi-blocklet configuration. We believe that more efficient sparse neural networks can be generated by embracing heterogeneity.

	sparsity = 50%		sparsity = 75%		sparsity = 87.5%	
Blocklet	Accuracy	Time	Accuracy	Time	Accuracy	Time
types		(ms)		(ms)		(ms)
1x1	72.64	20.33	71.84	12.96	70.86	9.36
2x2	72.27	15.4	70.37	10.56	68.94	8.15
4x4	71.67	13.25	69.62	9.6	67.7	7.74
2x1,1x1	72.67	20.3	71.88	12.96	70.54	9.29
4x2,2x1	72.3	17.67	71.13	11.59	69.84	8.59
8x2,4x2	71.77	15.22	70.5	10.44	68.47	8.11

Table 7.11: Effect of varying blocklet types in RMB on accuracy and runtime of Resnet18 sparse model.

	sparsity = 50%		sparsity = 62.5%		sparsity = 75%	
Blocklet	Accuracy	Time	Accuracy	Time	Accuracy	Time
types		(ms)		(ms)		(ms)
1x1	56	123.14	54.87	103.23	52.74	83.51
2x2	54.03	96.46	51.89	84.65	46.57	73.29
4x4	50.54	87.07	46.53	77.6	36.11	68.8
2x1,1x1	55.95	122.35	54.78	102.26	52.08	82.7
4x2,2x1	54.86	106.67	52.71	92.24	49.38	78.07
8x2,4x2	52.36	94.95	49.22	84.01	41.76	72.99

Table 7.12: Effect of varying blocklet types in RMB on accuracy and runtime of FCN-Resnet50 sparse model.

	sparsity = 50%		sparsity = 62.5%		sparsity $= 75\%$	
Blocklet	Accuracy	Time	Accuracy	Time	Accuracy	Time
types		(ms)		(ms)		(ms)
1x1	59.73	362.79	59.33	298.83	57.67	235.45
2x2	57.86	268.38	55.98	238.28	51.82	203.42
4x4	55.88	247.71	51.7	219.56	42.78	189.4
2x1,1x1	60.2	370.3	59.24	304.58	57.42	239.45
4x2,2x1	59.28	318.78	57.6	271.33	54.48	225.09
8x2,4x2	56.9	279.9	54.25	244.2	47.55	208.5

Table 7.13: Effect of varying blocklet types in RMB on accuracy and runtime of FCN-Resnet101 sparse model.

Chapter 8

Ramanujan Bipartite Graph Products for Efficient Block Sparse Neural Networks

Deep Neural Networks can help many real world tasks to achieve high prediction accuracies. However, such solutions cannot be easily ported to edge devices because of the highly constrained runtime environment with low memory and power usage constraints. Sparsity is an essential tool for generating compute and memory efficient DNNs. Despite this, the predominant choice of DNNs in production edge or server devices is dense instead of sparse. This is mainly because unstructured sparse neural networks tend to have poor run time performance on majority of the AI hardware, which are primarily designed for accelerating dense neural networks. Hence for truly uncovering the potential of sparsity in edge devices, it is necessary to generate structured sparse neural networks that are in harmony with the dense AI hardware.

Structured sparse neural networks can be generated using two approaches, namely D2S (Dense to Sparse) and S2S (Sparse to Sparse). In D2S, we start with a dense neural network and make it sparse during the training process. In S2S, we start and end the training process with a sparse neural network. D2S can be further divided into two types, namely Train-D2S and Finetune-D2S. In Train D2S, the starting neural network is untrained, and in Finetune-D2S, the starting neural network is pretrained. In Chapter 5, we used a Train-D2S approach to generate efficient sparse neural networks with block sparsity pattern. In Chapters 6 and 7, we used a Finetune-D2S approach to generate sparse neural networks with hierarchical block (HB), and regularized multi block (RMB) sparsity patterns respectively. In theory, using a Train-D2S to generate complex structured sparsity patterns like HB and RMB will be more efficient because the starting point is not a trained network. The reason we did not do that is because it is computationally intensive to maintain the structure during the training due to multiple levels of blocking. One can avoid

this and still generate complex structured sparse neural networks using Static-S2S approach, where the connectivity between neurons is fixed throughout the training process. In this work, we use a static-S2S approach based on the concepts of Ramanujan graphs, and graph products to generate structured sparse neural networks that are highly effective in memory and runtime metrics, which are critical for edge devices.

The run time of a sparse neural network on a given hardware is dependent on the efficiency with which the multiplication of a Sparse Matrix with a Dense matrix (SDMM) operation can be implemented. On a hardware like a GPU with a deep memory hierarchy (Registers > Shared memory > L2 cache > DRAM), the SDMM operation will have good run time efficiency if and only if it maximizes data accesses from faster memory through data reuse. For a structured sparse neural network, the amount of reuse depends on the choice of the structured sparsity pattern. In addition, the chosen pattern should be well connected to allow for good flow of information in the neural network. In this work, we address these requirements and generate structured sparse networks that are performant and connected. Following are our main contributions:

- Propose the Ramanujan Bipartite Graph Product (RBGP) framework for generating structured sparse neural networks that have multiple levels of block sparsity, good connectivity, and takes less memory for storage.
- Using the RBGP framework, we propose the RBGP4 structured sparsity pattern for the GPU, a representative dense hardware, and achieve good run time efficiency for the multiplication of a Sparse Matrix with a Dense Matrix (SDMM) operation on a GPU.
- We demonstrate the utility of the RBGP4 sparsity pattern on image classification and machine translation tasks on an edge (Nvidia Jetson Nano 2GB) as well as on server class (Nvidia V100) GPUs, and obtain significant speedups when compared to commonly used sparsity patterns like unstructured and block.

Related work : In the static sparse training approach, training starts and ends with a sparse neural network with connections remain unchanged. In other words, the mask (choice of connections) in each layer of the sparse neural network is kept fixed throughout the training process. Prior works in static sparse training approach differ in the way the mask is chosen. Prabhu et al. [71] makes use of expander graphs, and generates a random mask with row uniformity pattern, where all the rows in the mask have equal number of non zeros. Sourya et al. [23] generate a random mask with both row and column uniformity. Frankle et al. [25] use an unstructured mask generated by pruning a trained dense model. Kepner et al. [43] use the idea of radix topology to generate a

mask with cyclical diagonal pattern. Blocking pattern is the key requirement for achieving run time performance on dense AI hardware, and none of the above works incorporate block sparsity pattern. In this work, we impose the block sparsity pattern at multiple levels using RBGP framework, and achieve good run time performance on GPU, a representative dense AI hardware.

8.1 Preliminaries

In this section, we setup various definitions and notations used throughout the paper. First we define various types of block sparsity patterns.

Block Sparse (BS) matrix: A BS matrix W_{bs} is a sparse matrix, where non zero elements are structured in the form of blocks of size (bh, bw). The matrix W_{bs} has $(W_{bs}.rows/bh \times W_{bs}.columns/bw)$ number of blocks, and a block in W_{bs} is either a zero block with all zeros or a non-zero block with some or all elements as non-zeros. Figure 8.1a shows an example of a BS matrix.

Cloned Block Sparse (CBS) matrix: A CBS matrix is a block sparse matrix with block size (bh, bw) where all the non zero blocks of size (bh, bw) have the same non-zero pattern. Figure 8.1b shows an example of a CBS matrix.

Uniform Block Sparse (UBS) matrix: A UBS matrix W_{ubs} is a block sparse matrix with block size (bh, bw) where all the row/column blocks of size $(bh, W_{ubs}.columns)/(W_{ubs}.rows, bw)$ have equal number of non-zero blocks of size (bh, bw). Figure 8.1c shows an example of a UBS matrix.

Cloned Uniform Block Sparse (CUBS) matrix: A CUBS matrix is a block sparse matrix with block size (bh, bw) that is both UBS and CBS matrix with block size (bh, bw). Figure 8.1d shows an example of a CUBS matrix.

Recursive CUBS (RCUBS) matrix: An RCUBS matrix W_s is a sparse matrix with K levels of blocking $B_1, ..., B_K$ and following recursion: W_s is a CUBS matrix with block size B_1 , and a non zero block of size B_i in W_s is again a CUBS matrix with block size B_{i+1} . Figure 8.3 shows an example of RCUBS matrix with three levels of blocking.

We consider the bipartite graph G = (U, V, E) representation of matrices (with dimension $|U| \times |V|$). In a biregular bipartite graph, all the vertices in U and V have same degree d_l and d_r respectively. The degree also characterizes the sparsity of such graphs. The eigenvalues of a graph G are the eigenvalues of its adjacency matrix and they characterize many graph properties including connectivity [18]. A bipartite graph with N vertices have eigenvalues $\pm \lambda_1, ..., \pm \lambda_{N/2}$, where $\lambda_1 \ge \lambda_2 ... \ge \lambda_{N/2}$. The spectral gap between λ_1 and λ_2 is a measure of the connectivity properties of the graph [6]. Ramanujan Graphs are graphs with optimal connectivity (as measured



Figure 8.1: Types of block sparse matrices with block size (4,4). In a BS matrix, a block is either completely zero or not. In a CBS matrix, all non zero blocks have the same non zero pattern. In a UBS matrix, all row/column blocks have equal number of non zero blocks. A CUBS matrix is both a UBS matrix and a CBS matrix.

by the spectral gap) for a given level of sparsity [56] as defined below.

Ramanujan bipartite graph: A Ramanujan bipartite graph is a (d_l, d_r) -biregular bipartite graph, where the second largest eigenvalue λ_2 is less than or equal to $(\sqrt{d_l-1} + \sqrt{d_r-1})$.

Bipartite Graph Product (\otimes_b) : A bipartite graph product $(G_p = G_1 \otimes_b G_2)$ takes two bipartite graphs, $G_1(U_1, V_1, E_1)$ and $G_2(U_2, V_2, E_2)$ as the input and produces a bigger bipartite graph $G_p(U_p, V_p, E_p)$ where $U_p = U_1 \times U_2$, $V_p = V_1 \times V_2$, and E_p is constructed using the cross product of edges from G_1 and G_2 i.e, $E_p = \{((u_1, u_2), (v_1, v_2)) | (u_1, v_1) \in E_1$ and $(u_2, v_2) \in E_2\}$. A bipartite graph product can also be viewed from a matrix viewpoint in the following way.

A bipartite graph G(U, V, E) can be represented as a bi-adjacency matrix BA of size (|U|, |V|), with $BA_{uv} = 1$ if $(u, v) \in E$, and zero otherwise. For the bipartite graph product $(G_p = G_1 \otimes_b G_2)$, the bi-adjacency matrix of G_p is equal to the tensor product (\otimes) of the bi-adjacency matrices of the input bipartite graphs G_1 and G_2 i.e., $BA_p = BA_1 \otimes BA_2$. Figure 8.2 shows an example of bipartite graph product from the viewpoint of both a graph and a matrix.

8.2 Ramanujan Bipartite Graph Product Framework

The connectivity between neurons in a layer L of a sparse neural network can be captured using a bipartite graph G, where left/right neurons in L correspond to left/right vertices in G, and the connections between left and right neurons in L correspond to undirected edges between left and right vertices in G. The core idea in the RBGP (Ramanujan Bipartite Graph Product) framework is



Figure 8.2: Bipartite graph product operation (\otimes_b) along with matrix view. Biadjacency matrix of the product graph has CBS (Cloned Block Sparse) pattern with block size (2,2).

to express G as a bipartite graph product of Ramanujan bipartite graphs i.e $(G = G_1 \otimes_b ... \otimes_b G_K)$, where K is the number of base graphs. In the rest of the section, we show how expressing the connectivity of a layer using bipartite graph products leads to sparse neural networks that have structured sparsity, good connectivity, and memory efficiency.

Structured Sparsity: In the bipartite graph product $(G_p = G_1 \otimes_b G_2)$, the biadjacency matrix of G_p is equal to the tensor product (\otimes) of the biadjacency matrices of G_1 and G_2 i.e, $BA_p = BA_1 \otimes BA_2$. In the tensor product, BA_p is constructed by replacing each non zero element in BA_1 with BA_2 matrix, and each zero element in BA_1 with zero matrix of size BA_2 . As BA_2 is repeated, BA_p will have a CBS (Cloned Block Sparse) sparsity pattern with block size equal to the size of BA_2 or $(|G_2.U|, |G_2.V|)$. Figure 8.2 shows an example of a bipartite graph product, where the biadjacency matrix of the product graph has a CBS pattern with block size (2, 2). Additionally, when G_1 is a biregular bipartite graph, BA_p will have a CUBS (Cloned Uniform Block Sparse) sparsity pattern as BA_1 will have equal number of elements in all rows and all columns.

In the RBGP framework, the bipartite graph G of a layer L in the neural network is constructed by performing a series of (K - 1) bipartite graph products on K base biregular bipartite graphs $(G = G_1 \otimes_b \cdots \otimes_b G_K)$ that are Ramanujan. The bipartite graph G can be rewritten as $G = G_1 \otimes_b CG_2$, where $CG_2 = (G_2 \otimes_b \cdots \otimes_b G_K)$. As G_1 is a biregular bipartite graph, BA (biadjacency matrix of G) will have the CUBS sparsity pattern with block size $(\prod_{i=2}^{i=K} |G_i.U|, \prod_{i=2}^{i=K} |G_i.V|)$. Going deeper, as $CG_i = (G_i \otimes_b CG_{(i+1)})$, and also as all the base graphs are biregular, BA will have an RCUBS (Recursive Cloned Uniform Block Sparse) sparsity pattern with (K - 1) blocking levels $B_1 \cdots B_{(K-1)}$, where $B_j = (\prod_{i=j+1}^{i=K} |G_i.U|, \prod_{i=j+1}^{j=K} |G_i.V|)$. Figure 8.3 shows an example bipartite graph generated using the RBGP framework with four base graphs and three block sizes, (16, 16), (8, 8), and (2, 2).



Figure 8.3: Biadjacency matrix BA of a bipartite graph generated using RBGP framework. BA has RCUBS (Recursive Cloned Uniform Block Sparse) sparsity pattern with three blocking levels (16, 16), (8, 8) and (2, 2).

Memory Efficiency: A sparse neural network can be efficiently stored by only storing the information related to the connections that are present in the sparse layers. For a sparse layer L and its associated bipartite graph G, |E(G)| memory is required for storing the parameters corresponding to connections, and another |E(G)| memory is required for storing connectivity information in the form of adjacency list of G. Thus a total of $2 \times |E(G)|$ memory is required for storing the information of a layer in a sparse neural network. But in a RBGP sparse neural network, the memory requirement can be reduced by reducing the memory required for storing connectivity information. In the RBGP sparse neural network, as G is constructed using K base bipartite graphs $(G = G_1 \otimes_b ... \otimes_b G_K)$, the connectivity information of G can be reduced from $\prod_{i=1}^{i=K} |E(G_i)|$ to $\sum_{i=1}^{i=K} |E(G_i)|$, by only storing the connectivity information of the individual base graphs. For example, the bipartite graph G generated using RBGP framework in Figure 8.3 has 512 edges $(8 \times 2 \times 8 \times 4)$, but it only requires storing 22 edges (8 + 2 + 8 + 4) from the base graphs to construct the connectivity information of G, thus leading to a 23x reduction in memory requirement for storing the connectivity information when compared to a random bipartite graph with same number of edges as G.

Advantageous for edge: Edge devices are constrained in memory and computational capability.

Thus, sparse neural networks are ideal for edge devices as they have a low memory footprint and compute. However, the only drawback is that the run time performance is subject to the sparsity pattern present. Our RBGP framework addresses this issue by generating structured sparse neural networks with RCUB (Recursive Cloned Uniform Block) sparsity pattern. The cloning and block-ing properties allow for effective utilization of memory hierarchy, and uniformity property helps in load balancing the compute. Together, these properties will improve runtime performance on any dense AI hardware, be it an edge or server (see Section 8.4). Furthermore, the cloning property in the RCUB sparsity pattern allows for minimizing the memory required for storing the connectivity information in a sparse neural network by storing the cloned pattern's connectivity information only once.

Good connectivity: Connectivity in a sparse neural network is key for ensuring good flow of information. It is well known [6] that connectivity of the graph is characterized by the *spectral gap* between the largest and second largest eigenvalue (in absolute terms) of the adjacency matrix. In this section, we show that the spectral gap for the block sparse graph we construct using graph products, are optimal for any level of sparsity, for large graphs.

For a *d*-regular bipartite graph the largest eigenvalue in absolute value is d and -d. The next largest eigenvalue is considered as the second largest eigenvalue λ_2 . The spectral gap is $d - \lambda_2$, and the larger the spectral gap, the better connected the graph. Suppose the bipartite graph has n vertices on both sides, the degree d is αn with α being the density of the graph. For a given value of d, the best possible spectral gap of $d - 2\sqrt{d-1}$ is achieved by Ramanujan Graphs [6]. We construct block sparse graphs using graph products of smaller Ramanujan Graphs and show below that this construction has similar spectral gap as $n \to \infty$. For simplicity we consider the case where the bipartite graph G is the graph product of G_1, G_2 which are bipartite graphs with nvertices on each side and degree $d = \alpha n$. Note that G has degree d^2 and sparsity $1 - \alpha^2$.

Theorem 4. Let $G = G_1 \otimes_b G_2$ where G_i are bipartite graphs with n vertices on each sides and degree $d = \alpha n$. Then for any fixed level of density α or sparsity $(1 - \alpha)$,

$$\frac{IdealSpectralGap_{d^2}}{SpectralGap(G)} \to 1 \quad as \quad n \to \infty$$
(8.1)

where IdealSpectralGap_{d²} = $d^2 - 2\sqrt{d^2 - 1}$ is the best possible spectral gap for d^2 -regular graphs and SpectralGap(G) is the spectral gap of the block sparse graph G that we construct.

Proof. The biadjacency matrix of G is the tensor product of biadjacency matrices of G_1 and G_2 . Hence the eigenvalues of the biadjacency matrix are the product of eigenvalues of biadjacency



Figure 8.4: 2-lift operation on graph G. Clone graph G^c is first created and edges (u_1, v_1) and (u_2, v_2) are randomly chosen to cross over with the corresponding edges (u_1^c, v_1^c) and (u_2^c, v_2^c) respectively in the clone graph.

matrices of G_1 and G_2 . Since G_1 and G_2 are Ramanujan Graphs, their second largest eigenvalue is $2\sqrt{d-1}$. Hence, the second largest eigenvalue of G is $\lambda_2(G) = d \times 2\sqrt{d-1}$. The ideal value of the second largest eigenvalue for graphs of degree d^2 is $2\sqrt{d^2-1}$. Hence Equation 8.1, becomes

$$\frac{d^2 - 2\sqrt{d^2 - 1}}{d^2 - 2d\sqrt{d - 1}} = \frac{1 - 2\sqrt{1/d^2 - 1/d^4}}{1 - 2\sqrt{1/d - 1/d^2}}.$$

Hence, for any fixed level of density α or sparsity $(1 - \alpha)$, $n \to \infty$ (large matrices), $d \to \infty$, the left hand side of Equation 8.1 tends to 1.

8.2.1 Ramanujan Bipartite Graph Generation

A construction for Ramanujan Bipartite graph (RBG) was given by Bilu et al. [11]. The proof that this construction obtains the optimal eigenvalue gap was given by Marcus et al. [59]. We use algorithms (graph lifts) derived from these constructions to generate Ramanujan Bipartite Graphs for a given sparsity.

2-lift operation: A 2-lift is an operation applied on a graph G to produce a bigger graph G_L that is twice as big as G in both vertices and edges. In the 2-lift operation, a clone graph G^c is first created and the vertex set of G_L is set to be the union of vertex sets of G and G^c i.e, $V(G_L) = V(G) \cup V(G^c)$. The edge set of G_L i.e, $E(G_L)$ is then constructed in the following way: for an edge $(u, v) \in G$, and its corresponding clone edge $(u^c, v^c) \in G^c$, either the identity edge pair $\{(u, v), (u^c, v^c)\}$ or the crossover edge pair $\{(u, v^c), (u^c, v)\}$ is chosen at random and added to $E(G_L)$. Figure 8.4 shows an example of a 2-lift operation.

Generating sparse biregular bipartite graph: A 2-lift operation when applied on a biregular bipartite graph also results in a biregular bipartite graph that is twice as big with same left and right degrees. A biregular graph G(U, V, E) with sparsity $(1.0 - |E(G)|/(|G.U| \times |G.V|))$ sp, can be

generated by repeatedly applying $\log_2(1/(1-sp))$ 2-lift operations on a complete bipartite graph with $(1-sp) \times |G.U|$ left and $(1-sp) \times |G.V|$ right vertices.

Generating an RBG graph: A Ramanujan bipartite graph is a biregular bipartite graph with an additional constraint on the second largest eigenvalue of the adjacency matrix of the graph. To generate an RBG graph, we sample sparse biregular bipartite graphs generated using 2-lift operations until the sampled graph is Ramanujan. We found that an RBG graph with sizes in the order of thousands can be generated in the order of minutes. For a layer in the RBGP sparse neural network, the base Ramanujan graphs are generated only once before training and hence sampling approach is not a bottleneck.

8.3 The RBGP Framework for GPU

Dense matrix multiplication (GEMM) operation $(C = A \times B)$ can be efficiently implemented on hardware with a deep memory hierarchy like a GPU because of the high degree of data reuse present in GEMM operation. On the other hand, the efficiency of sparse matrix multiplication (SpGEMM) operation $(C = A_s \times B)$ depends on the sparsity pattern of A_s . For example, a structured pattern like a block sparsity pattern leads to more efficiency than an unstructured sparsity pattern. Our goal is to design a sparsity pattern such that the SpGEMM operation can be efficiently implemented. We do that by closely following the techniques of the GEMM implementation. The efficiency of the GEMM operation on a GPU is primarily due to two techniques: tiling, and register blocking.

Tiling: In tiling, matrices A, B and C are divided into tiles of sizes (TM, TK), (TK, TN), and (TM, TN) respectively and the tile CT_{ij} in C is computed in A.cols/TK or S steps i.e., $CT_{ij} = \sum_{k=0}^{S} (AT_{ik} \times BT_{kj})$. In each step, the tiles AT and BT are loaded into cache and the subproduct $(AT \times BT)$ is computed entirely out of cache. For tiling to be compatible with sparsity, the sparsity has to be at the tile level where a tile is either zero or not. Additionally, if the tile sparsity is uniform i.e., each (TM, A.cols) sized row block in A has an equal number of nonzero tiles, then all the tiles in C can be computed in equal number of steps. A uniform tile sparsity in A can be achieved using the RBGP framework by setting $G = G_o \otimes_b G_t$, with G_o as sparse, and size of G_t equal to the tile size of A.

Register blocking: In register blocking (RegBlock), tile matrices AT, BT and CT of sizes (TM, TK), (TK, TN), and (TM, TN) respectively are further divided into subtiles of sizes (BM, BK), (BK, BN), and (BM, BN) respectively. Further, elements in CT are divided into

Algorithm 21 RegBlockMM ($CTreg + = AT_s \times BT$) operation using register blocking (Reg-Block) technique. Pattern of AT_s is set as $(G_r \otimes_b G_i \otimes_b G_b)$, where base graphs G_r and G_b are dense with $|G_r.V| = 1$. Among RegBlock parameters (RM, RN, BM, BN), parameters RM, BM, and BK are set to $|G_r.U|, |G_b.U|$, and $|G_b.V|$ respectively. CTreg is stored as a 6D array of shape [EM, EN, RM, RN, BM, BN], where $EM = AT_s.rows/(RM \times BM)$ and $EN = BT.cols/(RN \times BN)$

- 1: **function** LOADBLOCK(matrix, (i, j), (bh, bw)) ▷ Load block of size (bh,bw) at block index (i,j)
- 2: **return** matrix[$i \times bh : (i+1) \times bh$][$j \times bw : (j+1) \times bw$]
- 3: end function

4:	for (em, en) in $[0, EM) \times [0, EN)$) do	Looping th	rough element groups
5:	Areg[RM][BM][BK]		▷ Registers
6:	Breg[RN][BK][BN]		▷ Registers
7:	for ik in $[0:G_i.d_l)$ do	▷ Looping thro	ugh non zero sub tiles
8:	for rm in $[0:RM)$ do		
9:	$bm = rm \times EM + em$		
10:	$Areg[rm] = LoadBlock(AT_s.data)$	(bm, ik), (BM, BK))	▷ Load into registers
11:	end for		
12:	$bk = G_i.adj_list[em][ik]$		
13:	for rn in $[0, RN)$ do		
14:	$bn = rn \times EN + en$		
15:	Breg[rn] = LoadBlock(BT, (bk, bk))	n), (BK, BN))	▷ Load into registers
16:	end for		
17:	for rm, rn in $[0, RM) imes [0, RN)$ do		
18:	CTreg[em][en][rm][rm] + = Areg[rm][rm]	$rm] \times Breg[rn]$	Computation out of
	registers		
19:	end for		
20:	end for		
21:	end for		

element groups, where each element group is of size $(RM \times RN \times BM \times BN)$, and is arranged as a strided 2D grid of subtiles with grid size as (RM, RN), and strides as (TM/BM)/RMin row dimension and (TN/BN)/RN in column dimension. Each element group is computed in TK/BK steps, where in step *i*, RM subtiles from *i*th column block of AT and RN subtiles from *i*th row block of BT are loaded into registers and the computation is performed entirely out of registers. Figure 8.5 shows the flow of RegBlock with (RM, RN) as (2,2). For RegBlock to be compatible with sparsity, the sparsity must be at the subtile level, and all RM row blocks in AT corresponding to an element group should have the same subtile sparsity pattern. Additionally, having all row blocks in AT to have an equal number of non zero subtiles allows for all the element groups in CT to be computed in an equal number of steps. The above requirements of sparsity in AT can be achieved by setting $G_t = G_r \otimes_b G_i \otimes_b G_b$, where graphs G_r and G_b are dense with sizes (RM, 1) and (BM, BK) respectively. Figure 8.5 shows a sparsity pattern that is compatible with the RegBlock technique.

8.3.1 The RBGP4 Sparsity Pattern

A sparsity pattern that is compatible with the tiling and register blocking techniques can be generated using the RBGP framework consisting of four base Ramanujan bipartite graphs G_o, G_r, G_i , and G_b i.e, $G = G_o \otimes_b G_r \otimes_b G_i \otimes_b G_b$, where graphs G_o and G_i as sparse, and graphs G_r and G_b as dense with $|G_r.V| = 1$. We call this sparsity pattern as the RBGP4 sparsity pattern.

CRBPG4 storage format: Any bipartite graph G generated using the RBGP framework is a biregular bipartite graph and because of that the RBGP4 sparse matrix A_s has an equal number of nonzeros in each row, where nnzr (the number of non-zeros in a row) is equal to $(G_o.d_l \times G_r.d_l \times G_i.d_l \times G_b.d_l)$. In the CRBGP4 format, the non-zeros in A_s are stored in the form of a matrix M of size $(A_s.rows, nnzr)$, where the non-zero elements in row r are stored in M[r]. Coming to the index information of A_s , we only store adjacency lists of graphs G_o and G_i as other base graphs G_r and G_b are dense.

RBGP4MM on GPU: We implement the RBGP4MM (Multiplication of an RBGP4 sparse matrix A_s with dense matrix B) operation ($C = A_s \times B$) as described in Algorithm 22 using tiling and register blocking (RegBlock) techniques. In lines 5-6, some of the tiling and RegBlock parameters are set using the RBGP4 configuration (G_o, G_r, G_i, G_b) of A_s . In line 9, we loop through the tiles in C and as tiles in C can be computed independently, we parallelize this for loop and map CT (a tile in C) to a thread block on GPU. As each row block in A_s has $G_o.d_l$ number of non zero tiles, CT is computed in $G_o.d_l$ steps. In line 12, we access the tile index *oind* from

Algorithm 22 RBGP4MM ($C = A_s \times B$) operation using tiling and RegBlock(Register Blocking) techniques. Among tiling parameters(TM, TK, TN), and RegBlock parameters (RM, RN, BM, BK), parameters TM, TK, RM, BM, BK are set using RBGP4 configuration ($G = G_o \otimes_b G_r \otimes_b G_i \otimes_b G_b$) of A_s .

1: **function** LOADBLOCK(matrix, (i, j), (bh, bw)) ▷ Load block of size (bh,bw) at block index (i,j)

```
2: return matrix[i \times bh : (i+1) \times bh][j \times bw : (j+1) \times bw]
```

3: end function

```
4: G_t = G_r \otimes_b G_i \otimes_b G_b
 5: TM, TK = |G_t.U|, |G_t.V|
 6: RM, BM, BK = |G_r.U|, |G_b.U|, |G_b.V|
 7: EM = TM/(RM \times BM)
 8: EN = TN/(RN \times BN)
 9: for (cm, cn) in [0, C.rows/TM)X[0, C.cols/TN) do
                                                                     ▷ Looping through tiles in C
       CTreg[EM][EN][RM][RN][BM][BN] = 0  > Size of CTreg is equal to tile size of C
10:
   i.e, TM \times TN
       for ok in [0, G_o.d_l) do
                                                 \triangleright Loop through non zero tiles in row block of A_s
11:
12:
           oind = G_o.adj\_list[cm][ok]
           AT_s.data = LoadBlock(A_s.data, (cm, ok), (TM, G_t.d_l))
                                                                               ▷ Load from main
13:
    memory to cache
           AT_s.pattern = G_r \otimes_b G_i \otimes_b G_b
14:
           BT = LoadBlock(B, (oind, cn), (TK, TN))  > Load from main memory to cache
15:
           ___syncthreads()
16:
           RegBlockMM(AT_s, BT, CTreg)
                                                       \triangleright Perform (CTreg + = AT_s \times BT) using
17:
    Algorithm 1
           __syncthreads()
18:
       end for
19:
       for (em, en) in [0, EM) \times [0, EN) do
                                                               ▷ Looping through element groups
20:
           for (rm, rn) in [0, RM) \times [0, RN) do
21:
               for (m, n) in [0, BM) \times [0, BN) do
22:
23:
                   row = cm \times TM + rm \times (TM/RM) + em \times BM + m
                   col = cn \times TN + rn \times (TN/RN) + en \times BN + n
24:
                   C[row][col] + = CTreg[em][en][rm][rm][m][n]
25:
               end for
26:
           end for
27:
28:
       end for
29: end for
```



Figure 8.5: Sparse matrix multiplication using register blocking technique with a compatible sparsity pattern. For a sparsity pattern to be compatible, RM row blocks in AT corresponding to an element group in CT should have the same sparsity pattern.

the adjacency list of graph G_o . In lines 13 and 15, threads in a thread block collectively load tile data from main memory into shared memory using step and tile indices. In line 38, we perform the RegBlockMM operation $(CTreg + = AT_s \times BT)$.

We implement the RegBlockMM operation as described in Algorithm 21. Elements in CT are divided into element groups, where each element group is of size $(RM \times BM \times RN \times BN)$. As groups of elements within a CT can be computed independently, we parallelize the for loop in line 4, and map an element group in CT to a thread on GPU. From line 7, we can see that each thread performs $G_i.d_l$ steps to process a group of elements in CT. Lines 8-19 describe the computation performed in each step. In lines 8-11, RM subtiles from AT_s are loaded into registers using step-index ik, and in lines 13-16 RN subtiles from BT are loaded into registers based on subtile index bk accessed in line 12 from adjacency list of G_i . In lines 17-19, subtiles loaded into registers are multiplied to generate an intermediate output for the associated element group. Instead of writing the intermediate output into the main memory, they are accumulated in registers.

In Algorithm 22, after processing all non zero tiles corresponding to CT in A_s using Reg-BlockMM calls, each thread writes the result of its associated element group into main memory from registers.

RBGP4 runtime characteristics: In RBGP4 sparsity pattern ($G = G_o \otimes_b G_r \otimes_b G_i \otimes_b G_b$), as G_r and G_b are dense or complete graphs, sparsity in G is solely due to presence of graphs G_o and G_i . If sp, osp, isp are the sparsities of G, G_o , and G_i respectively, then $sp = 1 - (1 - osp) \times (1 - isp)$. For a given sparsity in G, the sparsity distribution among G_o and G_i plays a key role in the efficiency obtained for the RBGP4MM operation ($C = A_s \times B$). As RBGP4 sparse matrix can be divided into two classes, namely a dense tile (DT) and a sparse tile (ST). In DT, isp = 0 and this means that a non zero tile in A_s is either full or empty. In ST, (isp > 0) and a non zero tile is





Figure 8.6: Efficiency of RBGP4MM operation on GPU. In RBGP4 sparse matrix, sparsity is due to base graphs G_o and G_i . The sparsity of G_o is set based on *isp* (Sparsity of G_i) and total sparsity. For denser tiles, speedups are close to ideal and as *isp* increases, efficiency decreases. Size of matrices are set to 4096x4096.

Figure 8.7: Matrices A_s, B , and C are divided into 4,2, and 2 tiles respectively. Both dense, and sparse tile cases have the same 50% sparsity. But to process a tile of C in dense tile case using tiling approach, only half the B matrix needs to be accessed. Whereas in the sparse tile case, entire Bmatrix needs to be accessed.

sparse. Figure 8.7 shows an example of DT and ST class matrices with 50% sparsity, where the DT case requires one step to compute a tile in C and requires to access only half the B. However, the ST case requires two steps, and requires to access all of B. This affects the efficiency of the RBGP4MM operation as number of steps and memory accesses to main memory increase. In Figure 8.7, we can see that DT case (isp = 0) has the highest efficiency. This is because the number of memory accesses from main memory decreases proportionally with the sparsity. However, for the ST case (isp > 0) as the tile gets sparser, the efficiency decreases. This is because number of steps required to compute a tile in C increases, which in turn increases the number of memory accesses from the main memory.

8.4 Results

The goal of the RBGP framework is to design sparse neural networks that can be efficiently processed on AI hardware designed to accelerate dense neural networks. In Section 8.3, using the RBGP framework, we proposed the RBGP4 sparsity pattern for the GPU, a dense AI hardware that is used across the spectrum from edge to server, and showed that the RBGP4 sparsity pattern leads to better run time efficiency on GPU. In this section, we evaluate sparse neural networks

Sparsity	Pattern	CIFAR10 Accuracy	CIFAR100 Accuracy	Memory (MB)	V100 Time (ms)	Nano Time (ms)
00.00	Dense	93.14	70.64	77.39	22	1040
50.00%	Unstructured	92.67	70.31	77.39	165	13276
	Block	92.45	70.75	41.12	94	12760
	RBGP4	92.58	70.48	38.76	20	1886
75.00%	Unstructured	91.99	69.32	38.71	86	8822
	Block	91.93	68.72	20.57	48	6314
	RBGP4	91.99	68.34	19.40	13	998
87.50%	Unstructured	90.88	65.41	19.37	79	6412
	Block	90.62	65.37	10.30	25	3202
	RBGP4	90.48	65.39	9.72	8	582
93.75%	Unstructured	90.01	62.33	9.70	50	4748
	Block	89.40	62.90	5.16	14	1688
	RBGP4	89.32	62.79	4.88	6	490

Table 8.1: Memory and runtime of sparsity patterns for image classification on CIFAR10 and CIFAR100 datasets using VGG19 networks running on V100 and Jetson Nano GPUs. For block pattern, we set block size to be (4, 4). Memory is given in MB, and time is given in milliseconds for one forward pass in training with 256 batch size.

with the RBGP4 sparsity pattern on two applications, namely image classification and machine translation, and compare them with unstructured, and block sparsity patterns. For training sparse neural networks, we use the predefined approach (choice of connections are chosen apriori to the training process) and benchmark them on edge (Nvidia Jetson Nano 2GB) GPU as well as server (Nvidia V100) GPU. For processing unstructured and block sparse neural networks, we use the cuSparse library provided by Nvidia.

8.4.1 Image Classification

We perform the image classification task on CIFAR datasets using VGG19 [77] as adapted by Liu et al. [54], and WideResnet-40-4 [88] networks. CIFAR10 and CIFAR100 datasets have 50K training images and 10K validation images each. We incorporate an equal amount of sparsity in all the layers for both the networks except for the first convolutional layer connected to the input and the final classifier layer. The convolution operation in a convolutional layer can be expressed as a matrix multiplication operation [17], where a 4D parameter tensor W of shape (K, C, R, S)

Sparsity	Pattern	CIFAR10	CIFAR100	Memory	V100 Time	Nano Time
		Accuracy	Accuracy	(MB)	(ms)	(ms)
00.00	Dense	95.01	77.20	34.10	40	2082
50.00%	Unstructured	95.42	77.92	34.10	241	23247
	Block	95.49	77.52	18.12	165	21877
	RBGP4	95.34	78.27	17.13	32	3249
75.00%	Unstructured	95.10	76.89	17.05	135	16470
	Block	94.92	76.50	9.07	85	10956
	RBGP4	94.72	76.80	8.57	20	1903
87.50%	Unstructured	94.48	75.21	8.53	102	12399
	Block	94.56	74.55	4.54	45	5611
	RBGP4	94.38	75.25	4.30	16	1231
93.75%	Unstructured	93.57	73.09	4.27	69	9507
	Block	93.55	71.86	2.27	26	2976
	RBGP4	93.53	72.44	2.16	14	1024

Table 8.2: Memory and runtime of sparsity patterns for image classification on CIFAR10 and CIFAR100 datasets using WideResnet-40-4 networks running on V100 and Jetson Nano GPUs. For block pattern, we set block size to be (4, 4). Memory is given in MB, and time is given in milliseconds for one forward pass in training with 128 batch size.



Figure 8.8: Throughput for VGG19 and WRN-40-4 networks on V100 GPU.

is viewed as 2D matrix \tilde{W} of shape $(K, C \times R \times S)$. For our experiments, we impose the sparsity pattern on \tilde{W} . For the optimizer, we used SGD optimizer with a momentum of 0.9, weight decay of 1e-4, and an initial learning rate of 0.1. The VGG19/WideResnet-40-4 network is trained for 160/200 epochs with a batch size of 256/128. For VGG19, the learning rate is multiplied by 0.1 at epochs 80 and 120. And for WideResnet-40-4, the learning rate is multiplied by 0.2 at epochs 60, 120 and 160. Furthermore, we used the knowledge distillation [37] technique for improving the training performance.

RBGP4 vs others: In Tables 8.1 and 8.2, we compare our RBGP4 sparsity pattern with unstructured and block sparsity patterns for sparsity levels of 50%, 75%, 87.5% and 93.75%. We can see that RBGP4 is as accurate as unstructured and block sparsity patterns while being faster and smaller. On V100/Nano GPU, RBGP4 is up to 10x/11x faster and takes 2x less memory when compared to unstructured sparsity pattern, and is up to 5x/7x faster when compared to block sparsity pattern.

Throughput vs batch size: GPUs are throughput-oriented machines that maximize run time efficiency by hiding memory access time with compute. The throughput for image classification task is defined as the number of images classified per second or simply images/sec. In this experiment, we study the effect of batch size on throughput. For V100 and Nano, we change the batch size from 32 to 2048 and 1 to 128, respectively, in multiples of 2. From Figure 8.8 and 8.9, we see that



Figure 8.9: Throughput for VGG19 and WideResnet-40-4 (WRN) networks on Jetson Nano2GB edge GPU.

on both V100 and Nano GPUs, RBGP4 has higher throughput than unstructured and block sparsity patterns. In the unstructured sparsity pattern, throughput remains flat because of the irregular memory and compute patterns present in the unstructured pattern. However, for the block sparsity pattern, it is either flat or marginally increases due to some amount of regularity in block structure. However, in the case of RBGP4 sparsity pattern, the throughput increases with batch size and stabilizes. This is because some of the layers in the RBGP4 sparse networks are extremely bandwidth bound at smaller batch sizes and offer limited data reuse across the memory hierarchy on the GPU.

8.4.2 Machine Translation

For the machine translation (MT) task, we train the transformer network [79] on the IWSLT dataset [15] with German (de) as the source language and English (en) as the destination language. The IWLST dataset has 160K and 7K sentence pairs for training and testing, respectively. We use the transformer network because it is widely used in the NLP (Natural Language Processing) community and is shown to be effective on many NLP tasks. We configure the transformer network to have six encoder blocks, six decoder blocks, and four attention heads. In the MT task, each sentence is divided into word tokens, where each token is represented as an embedding vector, and we set its size to 512. In transformer network, both encoder and decoder blocks have two fully connected layers fc1 and fc2 at the end. The number of input feature maps (ifm) for fc1 and the number of output feature maps (ofm) for fc2 are equal to the embedding size. As fc1 and fc2 are consecutive, ofm of fc1 and ifm of fc2 are equal and is set to 1024. We trained the
	BLEU at sparsity				Memory (MB) at sparsity			
Pattern	50.00%	75.00%	87.50%	93.75%	50.00%	75.00%	87.50%	93.75%
Unstructured	34.38	33.34	32.53	31.26	150.56	100.56	75.56	63.06
Block	33.81	33.40	32.13	31.13	103.68	77.13	63.84	57.20
RBGP4	34.36	33.59	32.54	31.28	100.74	75.73	63.15	56.90

Table 8.3: Machine translation on IWSLT (de to en) dataset using Transformer network. For block pattern, we set the size to be 4x4. Dense transformer model has a BLEU score of 34.51 takes up 150.56MB of memory.

network for 50 epochs using Adam optimizer with beta values of 0.9 and 0.98. The initial learning rate is set to 5×10^{-4} and is decreased gradually by dividing it by the square of the training step. We use a dropout value of 0.3 and weight decay of 10^{-4} for regularization. The first encoder and the first decoder block are kept as dense, and an equal amount of sparsity is incoroporated in the rest of the encoder and decoder blocks.

RBGP4 vs others: In the MT task, the BLEU (bilingual evaluation understudy) score metric is used for evaluating the quality of the translated text. From Table 8.3, we can see that across different sparsities, the RBGP4 sparsity pattern has BLEU scores on par with unstructured and block sparsity patterns. When compared to the unstructured pattern, RBGP4 takes up to 30% less memory. It has to be noted that the memory includes fixed components like dictionaries, which take up a significant amount of memory. There are three key matrix multiplication operations in a transformer network that repeatedly occur across encoder and decoder blocks. These three kernels are of the form E-E-N, E-F-N, and F-E-N, where E (512) is the embedding size, F (1024) is the number of input feature maps of the last fully connected layer in encoder and decoder blocks, and N is the number of tokens in the batch.(kernel X-Y-Z should be interpreted as the matrix multiplication of two matrices of size (X,Y) and (Y,Z)). In Table 8.4, we compare the runtime performance on these kernels across different sparsities. From Table 8.4, we can see that on V100/Nano GPU, our RBGP4 approach is up to 12x/25x and 7x/8x faster when compared to unstructured and block sparsity patterns, respectively.

Throughput vs batch-tokens (N): In the machine translation task, an input sentence to be translated is first divided into tokens that are either words or sub-words. In a batch of sentences, each sentence can have a different number of tokens. Batch-tokens is the cumulative sum of tokens of all sentences in a batch. In this experiment, we study how batch-tokens (N) affect the run time of

		V100 runtime (ms) at sparsity				Nano runtime (ms) at sparsity			
Kernels	Pattern	50%	75%	87.5%	93.75%	50%	75%	87.5%	93.75%
512-512-N	Unstructured	5.68	2.92	1.59	1.29	607	448	349	258
	Block	3.89	2.00	1.06	0.60	535	272	127	75
	RBGP4	0.65	0.34	0.24	0.15	73	36	20	11
512-1024-N	Unstructured	11.34	5.69	5.46	2.76	1096	762	587	447
	Block	7.69	3.91	2.04	1.10	1061	535	272	129
	RBGP4	1.26	0.66	0.45	0.25	146	70	38	20
1024-512-N	Unstructured	11.30	5.75	3.11	2.49	1217	899	700	550
	Block	7.75	3.97	2.10	1.18	1070	544	280	135
	RBGP4	1.16	0.61	0.42	0.25	153	71	39	21

Table 8.4: Runtime performance of key kernels in transformer network for unstructured,block, and RBGP4 sparsity patterns on V100 and Nano GPUs. For block pattern, we set the size to be 4x4.Kernel notation X-Y-N corresponds to matrix multiplication with matrix sizes (X,Y) and (Y,N). The value of N is set to 16K.

512-512-N, a representative kernel. For V100 and Nano, we vary N from 1K to 16K and 64 to 1K respectively in multiples of 2. From Figure 8.10, we can see that as the value of N increases, throughput of RBGP4 increases and stabilizes. This is because at lower N values, the amount of compute is less, and the kernel is heavily bound by bandwidth.

8.5 Conclusion

We used ideas from extremal graph theory and combinatorics to make sparse neural networks runtime efficient. Ramanujan graphs, which give optimal connectivity for a given level of sparsity, are used to model connections in a neural network layer. Furthermore, we obtain structured block sparsity by using products of Ramanujan graphs. We prove that the product graph also has the optimal connectivity for large matrices. For the specific case of GPUs, we describe how the block sparsity can be efficiently implemented in hardware, by exploiting the memory hierarchy through data reuse. Benchmarks of this implementation are shown to give significant runtime improvements on both edge and server GPUs. Similar ideas could be used to generate structured sparsity patterns that result in runtime efficient implementations in other hardware. We expect such methods will be useful specifically for edge devices where the runtime environment is highly constrained. For



Figure 8.10: Throughput of the key kernel (512-512-N) in Transformer network on V100 and Jetson Nano GPUs.

future work, generating combinatorial structured sparsity patterns like RBGP4 during the training process could lead to more accurate models.

Chapter 9

Conclusions and Future Directions

9.1 Conclusions

Computing machines have become indispensable in today's world. Applications run on computing machines using computational approaches. The need for enabling more applications and to run them efficiently and accurately fuels the innovation in the design of both computing machines and computational approaches. There are only two ways to make applications run more efficiently on existing computing machines. The first way is to efficiently run the underlying computational tasks of an existing computational approach. The second way is to design new computational approaches. In this thesis, we provide solutions to both ways. For the first way, we proposed a "codesign approach for computational tasks", a methodology where algorithms and data structures are co-designed to improve the runtime performance of the computational task by being aware of the input and the computing machine. For the second way, we proposed a "hardware aware computational approach", a methodology where the characteristics of the computing machine guide the design of the computational approach, resulting in efficient computational approaches.

Among the computing machines, the GPU computing machine has created a space for itself by super charging the field of Artificial Intelligence(AI) through acceleration of Artificial Neural Networks(ANN) based computational approaches. Since the first proof point of AlexNet [47] in 2012, the progress has been phenomenal in the field of AI. Because of such importance and the potential impact, we chose ANN based computational approaches for the study. Sparse ANNs achieve similar accuracy when compared to dense ANNs, while taking less compute and memory requirements. But the only problem with sparse ANNs is that they have poor runtime performance on the GPU due to irregularity in the compute and memory patterns. We made three contributions: 1) Developed a computational approach for generating block sparse neural networks, which can be easilty accelerated on the GPU 2) We proposed GPU friendly structured sparsity patterns like Hierarchical Block(HB), and Regularized Multi Block (RMB) and improved runtime performance on the GPU when compared to unstructured sparse ANNs without comprimising on the accuracy. 3) We proposed a novel way to generate structured connectivity patterns aimed for the edge devices using Ramanujan graphs and bipartite graph products.

9.2 Future Directions

During the course of our work, we came across several different areas which can be investigated more to generate optimal sparse neural networks. Generating sparse neural networks has many subtasks, like ranking and pruning, choosing the structure, pruning schedule, etc. Some interesting topics are discussed below.

9.2.1 Optimal Structured Pruning Algorithms

Pruning parameters in a neural network layer is essential for generating sparse neural networks. In order to prune parameters, the parameters have to be ranked using some criterion, say magnitude of the parameter, and then the low rank parameters are pruned to achieve the target sparsity. However, this results in an unstructured sparsity pattern, which has poor runtime performance on the GPU. We have shown that multi-level block sparsity patterns like HB(Chapter 6) and RMB(Chapter 7) leads to better runtime performance on GPU and proposed a simple pruning algorithm to generate multi-level block sparsity patterns. But we believe that more optimal pruning algorithms can be developed to increase sparsity without affecting performance and accuracy.

9.2.2 GPU Aware Multi Block Sparse Training

A sparse neural network can be generated either by finetuning or by training from scratch. In finetuning, the starting point is a pretrained dense model, and because of that the obtained accuracy using finetuning approach is very sensitive to the structure imposed. The optimal way to generate a sparse neural network is to use the training approach and induce sparsity through the training process. In Chapter 5, we provided an algorithm for inducing block sparsity pattern during training because of its suitability for the GPU. However, block sparsity is a very rigid structure, and we believe that more accurate sparse models can be generated by using multi block sparsity patterns like HB and RMB while retaining good runtime performance on the GPU.

Related Publications

- Dharma Teja Vooturi, Girish Varma, Kishore Kothapalli. Ramanujan bipartite graph products for efficient block sparse neural networks at *Journal on Concurrency and Com*putation: Practice and Experience, pages e6363, 2021. (https://doi.org/10.1002/cpe.6363)
- Dharma Teja Vooturi, Kishore Kothapalli . Efficient Sparse Neural Networks Using Regularized Multi Block Sparsity Pattern on a GPU at International Conference on High Performance Computing (HiPC), pages 215-224, Hyderabad, 2019.
- 3. Dharma Teja Vooturi, Girish Varma, Kishore Kothapalli . Dynamic Block Sparse Reparameterization of Convolutional Neural Networks at Workshop On Compact and Efficient Feature Representation and Learning(CEFRL), held in conjunction with International Conference on Computer Vision(ICCV), pages 3046-3053, Seoul, 2019.
- 4. Vooturi, Dharma Teja, Dheevatsa Mudigere, and Sasikanth Avancha. **Hierarchical block** sparse neural networks in *arXiv preprint arXiv:1808.03420 (2018)*.
- Dharma Teja Vooturi, Kishore Kothapalli, Upinder Singh Bhalla. Parallelizing Hines Matrix Solver in Neuron Simulations on GPU at International Conference on High Performance Computing (HiPC), pages 388-397, Jaipur, 2017.
- Dharma Teja Vooturi, Kishore Kothapalli. Parallel Algorithm for Quasi-Band Matrix-Matrix Multiplication, at Parallel Processing and Applied Mathematics(PPAM), pages 106-115, Krakow, 2015.

Bibliography

- [1] Cuda development toolkit. https://developer.nvidia.com/cuda-toolkit.
- [2] cusparse library. https://developer.nvidia.com/cusparse.
- [3] Intel math kernel library, https://software.intel.com/en-us/articles/intel-mkl/.
- [4] Opencl development framework. https://www.khronos.org/opencl/.
- [5] E. Agullo, A. Buttari, M. Byckling, A. Guermouche, and I. Masliah. Achieving high-performance with a sparse direct solver on Intel KNL. PhD thesis, Inria Bordeaux Sud-Ouest; CNRS-IRIT; Intel corporation; Université Bordeaux, 2017.
- [6] N. Alon. Eigenvalues and expanders. Combinatorica, 6(2):83–96, 1986.
- [7] 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, et al. The landscape of parallel computing research: A view from berkeley. 2006.
- [8] G. A. Ascoli, D. E. Donohue, and M. Halavi. Neuromorpho. org: a central resource for neuronal morphologies. *Journal of Neuroscience*, 27(35):9247–9251, 2007.
- [9] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–11, 2009.
- [10] R. Ben-Shalom, G. Liberman, and A. Korngreen. Accelerating compartmental modeling on a graphical processing unit. *Frontiers in neuroinformatics*, 7:4, 2013.
- [11] Y. Bilu and N. Linial. Lifts, discrepancy and nearly optimal spectral gap. Combinatorica, 26(5):495–519, Oct. 2006.
- [12] J. M. Bower and D. Beeman. The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System. Springer Science & Business Media, 2012.
- [13] A. Buluc and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In 2008 37th International Conference on Parallel Processing, pages 503–510. IEEE, 2008.

- [14] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 63–72, 2019.
- [15] M. Cettolo, J. Niehues, S. Stüker, L. Bentivogli, and M. Federico. Report on the 11th iwslt evaluation campaign, iwslt 2014. *Proceedings of the International Workshop on Spoken Language Translation, Hanoi, Vietnam*, 57, 2014.
- [16] M. Chaitanya and K. Kothapalli. Efficient multicore algorithms for identifying biconnected components. *International Journal of Networking and Computing*, 6(1):87–106, 2016.
- [17] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.
- [18] F. R. K. Chung. Spectral Graph Theory. American Mathematical Society, 1997.
- [19] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
- [21] Y. L. Cun, J. S. Denker, and S. A. Solla. Optimal brain damage. In Advances in Neural Information Processing Systems, pages 598–605. Morgan Kaufmann, 1990.
- [22] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS), 38(1):1–25, 2011.
- [23] S. Dey, K.-W. Huang, P. A. Beerel, and K. M. Chugg. Pre-defined sparse neural networks with hardware acceleration. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):332– 345, 2019.
- [24] D. Dutta, M. Chaitanya, K. Kothapalli, and D. Bera. Applications of ear decomposition to efficient heterogeneous algorithms for shortest path/cycle problems. In 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 864–873. IEEE, 2017.
- [25] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. International Conference on Learning Representations (ICLR), 2018.
- [26] A. Gharaibeh, L. B. B. Costa, E. Santos-Neto, and M. Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pages 851–862. IEEE, 2013.

- [27] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. *SIGARCH Comput. Archit. News*, 44(3):243–254, June 2016.
- [28] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [29] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In Advances in neural information processing systems, pages 1135–1143, 2015.
- [30] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In Advances in neural information processing systems, pages 164–171, 1993.
- [31] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.
- [32] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1389–1397, 2017.
- [33] J. L. Hennessy and D. A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2011.
- [34] M. Hines. Efficient computation of branched nerve equations. International journal of bio-medical computing, 15(1):69–76, 1984.
- [35] M. L. Hines, H. Eichner, and F. Schürmann. Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. *Journal of computational neuroscience*, 25(1):203–210, 2008.
- [36] M. L. Hines, H. Markram, and F. Schürmann. Fully implicit parallel simulation of single neurons. *Journal of computational neuroscience*, 25(3):439–448, 2008.
- [37] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *NIPS Deep Learning and Representation Learning Workshop*, 2015.
- [38] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500–544, 1952.
- [39] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2013.
- [40] Z. Huang and N. Wang. Data-driven sparse structure selection for deep neural networks. In Proceedings of the European Conference on Computer Vision (ECCV), pages 304–320, 2018.

- [41] S. B. Indarapu, M. Maramreddy, and K. Kothapalli. Architecture- and workload- aware heterogeneous algorithms for sparse matrix vector multiplication. In *Proceedings of the 7th ACM India Computing Conference*, 2014.
- [42] J. Kepner and J. Gilbert. Graph algorithms in the language of linear algebra. SIAM, 2011.
- [43] J. Kepner and R. Robinett. Radix-net: Structured sparse matrices for deep neural networks. 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 268–274, 2019.
- [44] Khronos Group. SYCL 2020 Specification, 2020. https://www.khronos.org/registry/ SYCL/specs/sycl-2020/html/sycl-2020.html.
- [45] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.
- [46] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.
- [47] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.
- [48] J. L. Larriba-Pey, M. Mascagni, A. Jorba, and J. J. Navarro. An analysis of the parallel computation of arbitrarily branched cable neuron models. In *PPSC*, pages 373–378, 1995.
- [49] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In Advances in neural information processing systems, pages 598–605, 1990.
- [50] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *In International Conference on Learning Representations (ICLR)*, 2016.
- [51] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky. Sparse convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.
- [52] W. Liu and B. Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 370–381. IEEE, 2014.
- [53] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. *Proceedings of the IEEE International Conference on Computer Vision* (*ICCV*), pages 2736–2744, 2017.
- [54] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. *arXiv* preprint arXiv:1810.05270, 2018.
- [55] C. Louizos, M. Welling, and D. P. Kingma. Learning sparse neural networks through *l*_0 regularization. arXiv preprint arXiv:1712.01312, 2017.

- [56] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. Combinatorica, 8(3):261–277, 1988.
- [57] J.-H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.
- [58] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally. Exploring the regularity of sparse structure in convolutional neural networks. arXiv preprint arXiv:1705.08922, 2017.
- [59] A. W. Marcus, D. A. Spielman, and N. Srivastava. Interlacing families i: Bipartite ramanujan graphs of all degrees. *Annals of Mathematics*, 182(1):307–325, 2015.
- [60] M. Mascagni. A parallelizing algorithm for computing solutions to arbitrarily branched cable neuron models. *Journal of neuroscience methods*, 36(1):105–114, 1991.
- [61] K. K. Matam, S. R. K. Bharadwaj, and K. Kothapalli. Sparse matrix multiplication on hybrid cpu+ gpu platforms. In Proc. of 19th Annual International Conference on High Performance Computing (HiPC), Pune, India, pages 1–10, 2012.
- [62] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. *ICLR*, 2016.
- [63] S. Narang, E. Elsen, G. Diamos, and S. Sengupta. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017.
- [64] S. Narang, E. Undersander, and G. Diamos. Block-sparse recurrent neural networks. *arXiv preprint arXiv:1711.02782*, 2017.
- [65] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.
- [66] J. D. Owens, M. Houston, D. Leubke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- [67] C. Pachorkar, M. Chaitanya, K. Kothapalli, and D. Bera. Efficient parallel ear decomposition of graphs with application to betweenness-centrality. In 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pages 301–310. IEEE, 2016.
- [68] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. S. Emer, S. W. Keckler, and W. J. Dally. SCNN: an accelerator for compressed-sparse convolutional neural networks. *CoRR*, abs/1708.04485, 2017.
- [69] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*, 2016.

- [70] A. Picciau, G. E. Inggs, J. Wickerson, E. C. Kerrigan, and G. A. Constantinides. Balancing locality and concurrency: Solving sparse triangular systems on gpus. In 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pages 183–192. IEEE, 2016.
- [71] A. Prabhu, G. Varma, and A. Namboodiri. Deep expander networks: Efficient deep networks from graph theory. *The European Conference on Computer Vision (ECCV)*, September 2018.
- [72] K. R. Ramamoorthy, D. S. Banerjee, K. Srinathan, and K. Kothapalli. A novel heterogeneous algorithm for multiplying scale-free sparse matrices. In 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pages 637–646. IEEE, 2015.
- [73] S. Ray, R. Deshpande, N. Dudani, and U. S. Bhalla. A general biological simulator: the multiscale object oriented simulation environment, moose. *BMC Neuroscience*, 9:1–2, 2008.
- [74] E. Romera, J. M. Alvarez, L. M. Bergasa, and R. Arroyo. Erfnet: Efficient residual factorized convnet for real-time semantic segmentation. *IEEE Transactions on Intelligent Transportation Systems*, 19(1):263–272, 2018.
- [75] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.
- [76] A. R. Scott Gray and D. P. Kingma. Gpu kernels for block-sparse weights. 2017.
- [77] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.
- [78] G. M. Slota and K. Madduri. Simple parallel biconnectivity algorithms for multicore platforms. In 2014 21st International Conference on High Performance Computing (HiPC), pages 1–10. IEEE, 2014.
- [79] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.
- [80] D. T. Vooturi and K. Kothapalli. Parallel algorithm for quasi-band matrix-matrix multiplication. In Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part I 11, pages 106–115. Springer, 2016.
- [81] D. T. Vooturi, D. Mudigere, and S. Avancha. Hierarchical block sparse neural networks. *arXiv preprint arXiv:1808.03420*, 2018.
- [82] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082, 2016.

- [83] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1492–1500, 2017.
- [84] W. Yang, K. Li, and K. Li. A hybrid computing method of spmv on cpu–gpu heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 104:49–60, 2017.
- [85] W. Yang, K. Li, Y. Liu, L. Shi, and L. Wan. Optimization of quasi-diagonal matrix-vector multiplication on gpu. *The international journal of high performance computing applications*, 28(2):183–195, 2014.
- [86] J. Ye, X. Lu, Z. Lin, and J. Z. Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. *arXiv preprint arXiv:1802.00124*, 2018.
- [87] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis. Nisp: Pruning networks using neuron importance score propagation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.
- [88] S. Zagoruyko and N. Komodakis. Wide residual networks. Proceedings of the British Machine Vision Conference (BMVC), pages 87.1–87.12, September 2016.