GPU-Accelerated Optimization of Block Lanczos Solver for Sparse Linear System

A DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE AWARD OF THE DEGREE
OF

MASTER OF TECHNOLOGY

IN

INFORMATION SYSTEMS

SUBMITTED BY:

PRASHANT VERMA
2K18/ISY/07

UNDER THE SUPERVISION OF

Prof. KAPIL SHARMA
HOD & Professor

DEPARTMENT OF INFORMATION TECHNOLOGY
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042
June, 2020

DEPARTMENT OF INFORMATION TECHNOLOGY

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College Of engineering)

Bawana Road, Delhi-110042

<u>CANDIDATE DECLARATION</u>

I, **PRASHANT VERMA**, Roll No. **2K18/ISY/07** student of M.Tech (Information Systems), hereby declare that the Project Dissertation titled **“GPU-Accelerated Optimization of Block Lanczos Solver for Sparse Linear System”** which is submitted by me to the Department of Information Technology, Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of the degree of Master of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Prashant Verma

Place: DTU, Delhi **Prashant Verma**

Date: 29th June, 2020 **(2K18/ISY/07)**

DEPARTMENT OF INFORMATION TECHNOLOGY

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College Of engineering)

Bawana Road, Delhi-110042

CERTIFICATE

I hereby certify that the Project Dissertation titled **“GPU-Accelerated Optimization of Block Lanczos Solver for Sparse Linear System”** which is submitted by **PRASHANT VERMA**, Roll No **2K18/ISY/07** Department of Information Technology, Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of the degree of Master of Technology, is a record of the project work carried out by the student under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

[illegible]

Place: DTU, Delhi **(Prof. Kapil Sharma)**

Date: **SUPERVISOR**

HOD & Professor

ACKNOWLEDGEMENT

I am grateful to the God for the good health and wellbeing that were necessary to complete this project. I would like to express my gratitude to my mentor **Prof. Kapil Sharma, Head of the Department, Department of Information Technology**, for his useful comments, remarks and engagement through the learning process of this project. I place on record, my sincere thanks to all of the Department faculty members for their help and support. I also thank my parents for the unceasing encouragement, support and attention.

[illegible]

Prashant Verma

(2K18/ISY/07)

ABSTRACT

Solving large and sparse system of linear equations has been extensively used for several cryptanalytic techniques. Block Lanczos and Block Wiedemann algorithms are well known for solving large sparse systems. However, the time complexity of such popular methods makes it reluctant and hence, the concept of parallelism is made compulsory for such methods. This work introduced an optimization of the Block Lanczos method over the finite field using GPUs. Here we consider GF (2) finite field. The optimization of parallel Block Lanczos solver is performed using NVIDIA Compute Unified Device Architecture (CUDA) and Message Passing Interface (MPI) to take advantage of multilevel parallelism on multi-node and multi-GPU systems. CUDA-aware MPI has been extensively used to leverage GPU-Direct Remote Direct Memory Access (RDMA) and GPU-Direct Point to Point (P2P) for optimized inter and intra node communication. The proposed optimization of Block Lanczos solver explored the memory bandwidth on a single Tesla, multi Tesla K40 and multi Tesla P100 GPU nodes. The parallel efficiency is also achieved on the DGX system with Pascal P100 GPUs respectively.

Keywords: *Block Lanczos, Cryptanalysis, Graphics Processing Unit, GPU-Direct P2P, MIMD, Parallel-Processing, RDMA.*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS, ABBREVIATIONS AND NOMENCLATURES

NFS	Number Field Sieve
GPU	Graphics Processing Unit
ORNL	Oak Ridge National Laboratory
CUDA	Compute Unified Device Architecture
P2P	Point 2 Point
RDMA	Remote Direct Memory Access
GAPP	Geometric Arithmetic Parallel Processor
GF(2)	Galois Field For Binary
FPGA	Field Programmable Gate Array
M4RI	Method of Four Russians
DOK	Dictionary of Keys
LOL	List of lists
COO	Coordinated Lists
CSR	Compressed Sparse Row
CSC	Compressed Sparse Column
MIMD	Multiple Instructions Multiple Data
SpMV	Sparse Matrix Vector Multiplication
SpMTV	Sparse Matrix Transpose Vector Multiplication

CHAPTER 1
INTRODUCTION

1.1 OVERVIEW

In today’s world, the growth of digital information has been increased rapidly therefore information security is imperative for the security requirement of the digital world. There are various cryptanalytic techniques in which solving an extensive sparse system of linear equations over finite field become a challenge due to high computation. For instance, the problem like NFS for factorization of large integers, symmetric ciphers for crypt-analytic problem, discrete log problem and algebraic attacks involves solving large sparse linear systems over finite field. Solving system of sparse linear equations is one of the important and compute intensive step of such problems.

Block Lanczos [2] [11] and Block Wiedemann [3] [4] [5] are the popular methods to solve such compute intensive problems but the time complexity of such systems is cubic, therefore such system is computationally slow and practically not feasible. As the Block Wiedemann method has been well parallelized therefore we focus on optimizing the Block Lanczos method. To solve compute intensive problems in reasonable amount of time, accelerated units such as general-purpose graphics processing units (GPGPUs) are accomplished. Now a days the trend is to create a cluster kind of supercomputer facility where each socket hosts either single or multiple GPUs. If we see the top 500 supercomputer list [6] we found that most of them are GPUs based. For instance, the summit supercomputer created by the Oak Ridge National Laboratory (ORNL) [7] hosts up to 6 Volta V100 NVidia GPUs on each node of the cluster and it is 10 times more powerful than Titan which stands in fifth place of top 500 supercomputer list. Thus, it is necessary to develop applications in a way that it can be efficiently scaled over multiple GPGPUs and nodes. The original method for Block Lanczos algorithm [8] [9] is roughly split in to three steps i.e. pre-processing, Lanczos iterations and post-processing. At higher densities (>10%), the Block Lanczos is quite costly in terms of performance.

1.2 RESEARCH OBJECTIVE

This project describes the optimization exercise carried out on an existing GPU enabled code for Block Lanczos algorithm. The optimization exercise started with understanding, performance profiling of the current Block Lanczos method. For benchmarking the performance of the original as well as optimized process, linear systems with up to 30,000 unknowns and densities (number of non-zero elements) from 0.01% to 16% are considered. In the proposed work, we optimized the Block Lanczos parallel solver for sparse linear systems over binary Galois field, i.e. GF (2). The optimized block Lanczos solver is implemented using CUDA [27] version 7.5, CUDA driver version 352.55, 4 Tesla K40 and multi Tesla P100 GPU nodes. This kind of available parallel hardware architecture explores the computer capability GPGPUs. The optimized parallel block Lanczos solver is implemented using NVIDIA Compute Unified Device Architecture (CUDA) and Message Passing Interface (MPI) [28] to take advantage of multi-level parallelism on multi-node and multi-GPU systems. CUDA-aware MPI has been extensively used to leverage GPU-Direct Remote Direct Memory Access (RDMA) and GPU-Direct Point to Point (P2P) [29] for optimized inter and intra node communication. The proposed optimized Block Lanczos solver explored the memory bandwidth on an individual Tesla, multi Tesla K40 and multi Tesla P100 GPU nodes [30]. The proposed optimized solver is constructed in a way that load balancing and optimized communication can be achieved easily.

1.3 ORGANISATION OF THESIS

The rest of the work is formulated as follows. The next chapter gives the details of the work related to the area for solving an extensive sparse system of linear equations over GF(2) using the Block Lanczos algorithm [10] [11]. Chapter 3 describes the motivation and contribution of the proposed work. The proposed methodology for the optimization of Block Lanczos solver in multiple parallel hardware platforms is explained in Chapter 4. The next Chapter 5 shows the experiments results over different parallel hardware platform for performance and scalability, and finally, Chapter 6 concludes the work and future scope.

CHAPTER 2
LITERATURE REVIEW

There are various crypt-analytic techniques where solving a large dense or sparse system of linear equations over finite field become a challenge due to high computation. For in- stance, algorithms like number field sieve for integer factorization, symmetric ciphers for cryptanalysis, discrete log problem and algebraic attacks involves solving large sparse or dense linear systems over finite field. Here we consider GF(2) finite field. Gaussian Elimination is the popular and relevant method for solving large dense systems while Block Lanczos and Block Wiedemann algorithms are well known for solving large sparse systems. However, the time complexity of such popular method makes it reluctant and hence, the concept of parallelism is made compulsory for such methods.

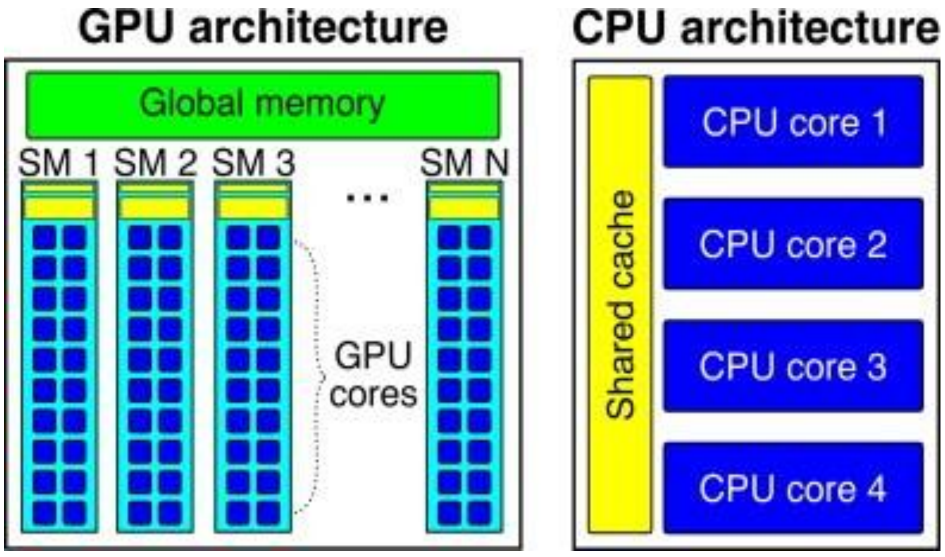


Fig. 2.1 CPU and GPU Architecture

To solve compute intensive problems in reasonable amount of time, accelerated units such as general-purpose graphics processing units (GPGPUs) are accomplished. The accelerators with thousands of cores available today, explore the bandwidth of memory and take advantage of multi- level parallelism on multi-node and multi-GPU units.
“Fig. 2.1” showed the architecture of CPU and GPU. Here, we consider NVidia GPUs like Kepler, Pascal and Volta along CUDA and MPI. Also, CUDA-aware MPI leverages GPU-Direct RDMA and P2P for inter and intra node communication.

2.1 RELATED WORK

In order to solve dense and sparse system of linear equations over GF(2) the methods that are available were implemented serially [12] [13]. The parallel implementation is also available, but they are not optimized with latest hardware platforms available and hence not fully utilized the available hardware resource of latest existing technology. Nvidia introduces series of accelerating cards for researchers to make their application parallel and solve bigger problems in a reasonable amount of time.

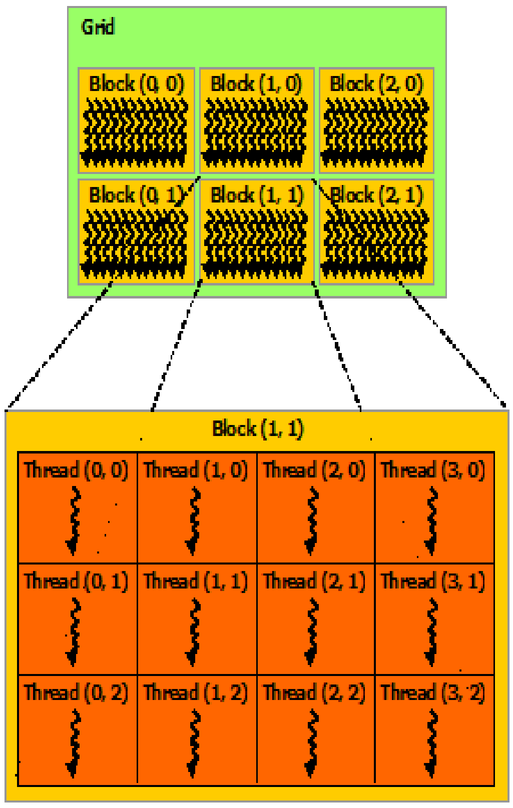


Fig. 2.2 Grid of Threads Blocks in GPU

“Fig. 2.2” shows how thread blocks in a grid is arranged in a GPU. The existing hardware platform makes the application efficient and scalable. For solving large system of dense linear equations Gaussian elimination is a prominent area for researchers and the research to optimize its parallel version is less focused. Koc and Arachchige [14] proposed Gaussian Elimination algorithm over a finite field GF(2) and implemented the same on the Geometric Arithmetic Parallel Processor known as GAPP. Parkinson and Wunderlich [15] proposed the parallel Gaussian Elimination for finite field GF(2) and the same was deployed on the parallel array processor named as ICL-DAP. Bogdanov et al. [16] used hardware that is parallel in architecture to solve Gaussian Elimination over a finite field GF(2) quickly. This architecture was implemented on a Field-programmable gate array (FPGA). In addition to this, the author also evaluates for a possible implementation based on ASIC architecture. All these solutions can solve only small systems of either dense or sparse linear equations over a finite field GF(2) and are very costly using special kind of architecture shown in “Fig. 2.3”.

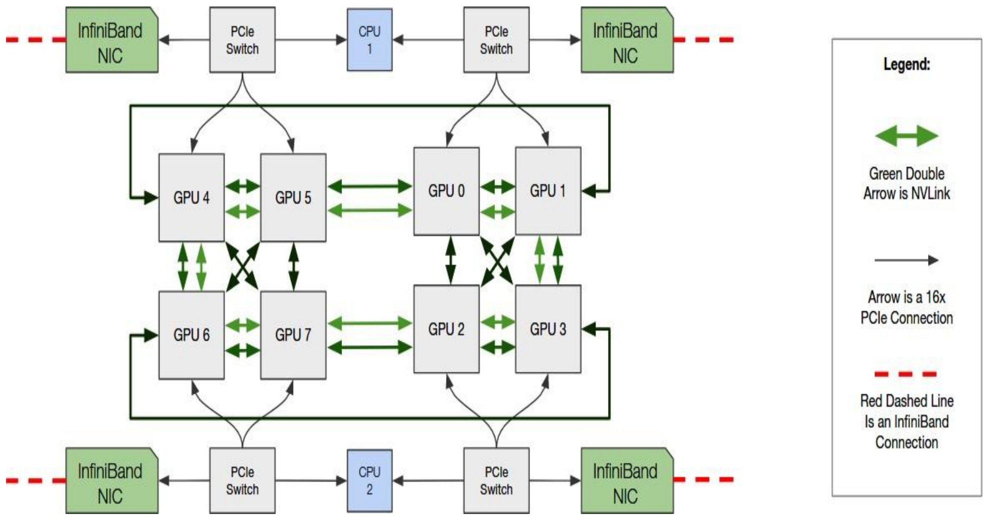


Fig. 2.3 Multi GPUs Connection to host

Albrecht and Pernet [17] proposed the solution of a dense system of linear equations over a finite field GF(2). The solution used multicore architectures and are very efficient and part of the M4RI (Method of four Russians) library [18]. This solution shows the performance results for 64 X 64 K linear systems of equations and presented that their method is as good as to the implementation by Allan Steel [19] for solving Gaussian Elimination over GF(2) using MAGMA library.

This is the first work to address Gaussian Elimination over GF(2) in a general-purpose processor. This solution shows the performance results for 64 X 64 K linear systems of equations and presented that their method is as good as to the implementation by Allan Steel [19] for solving Gaussian Elimination over GF(2) using MAGMA library.

2.1.1 DENSE LINEAR SYSTEM SOLVER OVER GF(2)

The system is of the form A * x = B (mod 2) where Matrix A is dense its 50% of the elements non-zero and no. of rows is greater than no. of columns. All arithmetic operations are over GF(2) which means that addition and multiplication is equivalent to logical XOR and logical AND respectively. The gaussian elimination to solve large dense system of equations has following steps:

(A) GENERATE RANDOM MATRICES

[1] Generate entries of A and x with pseudo-random number generator

[2] Compute A*x = B

[3] Solve linear system [A, B]

[4] Compare computed solution of linear system with reference.

(B) GENERATE LINEAR SYSTEM USING LFSR

[1] LFSR is initialized to random input

[2] Clocked multiple times to produce multiple bits of output

[3] The output bits are expressed as linear combination of initial condition

[4] With enough equations, a linear system of equations can be formed

Initial condition of LFSR as unknown

[5] Solve the linear system

Compare the computed initial condition with reference

(C) SINGLE & MULTI-GPU GAUSSIAN ELIMINATION

[1] Matrix is split in parts row-wise

[2] Each GPU own exactly one part

[3] All processing (3 kernels) on the part is done by owner GPU

[4] All operations are done in parallel by the GPUs

[5] Consensus about pivot is achieved after find Pivot operation

(D) OPTIMIZATION

[1] Performance is heavily influenced by memory access pattern.

[2] How should A be stored?

[3] Find pivot prefers column major, extracting pivot row prefers row major

[4] One coalesced and one stride accesses of memory is unavoidable

[5] Row reduction works better with column major

[6] Stores transpose A instead of A

[7] Memory access pattern

2.1.2 PARALLELIZATION OF GAUSSIAN ELIMINATION

The algorithms for Gaussian elimination over real field and over GF(2) are both identical. A high-level pseudo-code is as follows:

1. For iteration 1 to N

1.1. Find a row with non-zero pivot

1.2. Extract the pivot row

1.3. Reduce other rows using pivot row

2. Backsubstitution

GPU kernels: A GPU kernel, or simply kernel, is any function that is executed on the GPU. The implementation of Gaussian elimination includes programming kernels for Steps 1.1, 1.2 and 1.3 defined above.

For each iteration, the three kernels are launched and executed in order.

__global__ void findPivotRowAndMultipliers (stSolverState solverState, unsigned int* packedTransposeAB);

__global__ void extractPivotRow (stSolverState solverState, unsigned int* d_packedTransposeAB);

__global__ void rowElimination (unsigned int* d_packedTransposeAB, stSolverState solverState);

Each kernel has exactly two arguments, the augmented linear system matrix and a structure variable holding the current state of the solver. The solver state variable has two kinds of data members. Those that are updated on every iteration and those which are iteration invariant. The members that are updated every iteration include, extracted pivot row, column index of pivot element, indices of rows that are already reduced etc. The iteration invariant members are meta-data parameters about the linear system such as number of rows and columns etc. Kernels for finding pivot and extracting pivot row access the linear system as input and update the corresponding parts of solver state variable. The row elimination kernels read the solver state as input and modify the augmented linear system.

2.1.3 NON-SQUARE LINEAR SYSTEMS

The probability of a randomly generated square matrix over GF(2) being invertible is pretty low which means the probability of successfully solving N equations in N unknowns over GF(2) is also low. Typical strategy to increase this probability is to keep number of equations, M, more than number of unknowns N (M > N). The resulting matrix A for such linear system would be a non-square matrix. The current implementation is designed to operate on such non-square (tall) linear systems. All the M equations are used during Step 1.1, 1.2 and 1.3, while in Backsubstitution; only first

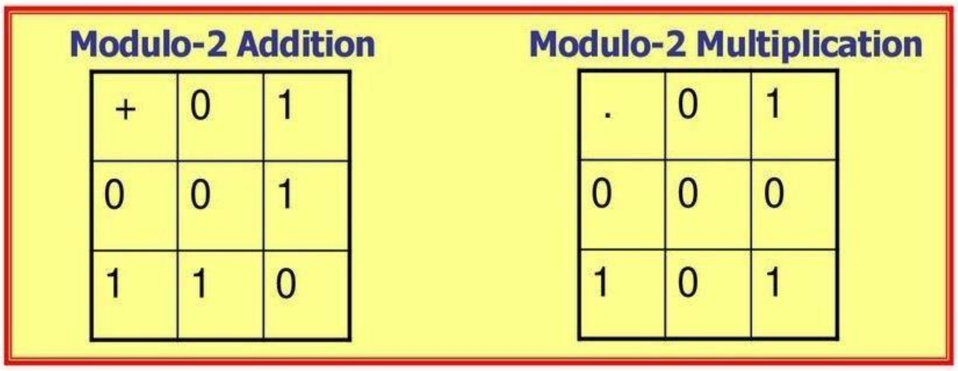
N equations are used. In case of unique solution, the last (M–N) equations will all be zero anyway and thus those equations are ignored for the back substitution step.

2.1.4 NULL COLUMN LIMITATION

The implementation handles non-square cases described in section 2.1.2 gracefully. It cannot work with systems having null columns i.e. a column with all elements as zero. Any null column must be removed from the linear system before passing it to the solver.

2.2 FINITE FIELD GF(2)

In the area of mathematics, Finite Field which is also called as Galois Field represents a field that contains the elements which is finite in number. For instance, GF(2) is a finite field for binary or Galois field that contains two elements zero or one and hence GF(2) is a simple way to express binary (0 or 1) data. It can be applied in various application areas of information theory and data processing. Galois Field for binary has specific arithmetic properties, i.e. the addition over GF(2) is equal to an exclusive OR i.e. (XOR) operation.



Modulo-2 Addition

+	0	1
0	0	1
1	1	0

Modulo-2 Multiplication

.	0	1
0	0	0
1	0	1

Fig. 2.4 Galois Field for Binary

In contrast, the multiplication over GF(2) is equal to an AND operation. It can be shown in "Fig. 2.4" that F {0, 1} is a Galois Field or Finite Field of the order two under modulo-two addition and modulo-two multiplication. Given a system of linear equations over Galois Field for Binary, i.e. GF (2) which has M number of equations and N number of unknowns. This system can be represented in the form of Ax = b where A is a sparse matrix having M number of rows and N number of columns, x is a solution vector also known as a column vector having N number of rows, and finally, b is a column vector having M number of rows. As the given system Ax = b is characterized in GF (2) therefore all the operations along with sparse matrix (A) , solution vector (x) and column vector (b) belong to Galois Field for Binary i.e. GF (2). To solve the system of linear equations means finding solution vector x given the sparse matrix and column vector A and b respectively and A cannot be non-invertible in Galois Field for binary, i.e. GF (2).

2.3 PARALLEL PROCESSING AND HYBRID ARCHITECTURE

This section describes the parallel processing overview using CUDA and MPI. It also explains the hybrid architecture along with their connectivity.

2.3.1 GENERAL PURPOSE COMPUTING IN GPUs USING CUDA

The kind of general-purpose computing on GPUs is termed as the general-purpose graphics processing unit, i.e. GPGPUs. GPGPU computing belongs to general-purpose computing on graphics processing units (GPUs). Earlier GPUs were mainly used for gaming purpose, but as the technology emerges and need of solving massively parallel application in a reasonable amount of time the GPUs with thousands of cores are used for the computational purpose. Now days Graphics Processing Unit i.e. GPUs are used in most of the devices like embedded systems, mobile phones, gaming consoles and personal computers. The Tesla, Pascal and Volta series of NVIDIA is well known and famous for accelerating the offloads and data-parallel computing. AMD and Intel also have their accelerated units, but here we are considered NVidia GPUs cards. If we talk about the GPUs architecture, the GPU is a device consisting of an array of multithreaded SMs (Streaming Multiprocessors), and each SM has set of CUDA cores. For instance, if we talk about the latest V100 GPU, it has a new redesign of the SM processor architecture, which has 84 SM and 5120 cores.

The CUDA is a C-based programming model from NVidia that explores the compute capability of GPUs by NVidia in general-purpose computing. In the context of CUDA, the CPU is called the host and the GPU is called device. Each host consists of one or more device, and the device is simply an add-on card attached through PCIe. Without CPU, GPI is nothing it acts merely as an accelerator card to offload the massively parallel part of an application from host to the device. In GPU a kernel is launched on the device as grid of thread blocks. A thread blocks contains a fixed number of threads and can stretch in 1, 2 or 3 dimensions. In the same way a grid can also stretch. “Fig. 2.5” represents an example of a kernel which is launched in a two dimensional grid having two dimensional thread blocks. These threads with in the block are merely identified based on their block and thread index correspondingly. A thread block can be run on one streaming multiprocessor, and multiple thread blocks can be executed in the same SM. A warp is a set of threads (either 32 or 64) depend on the architecture, within a thread block such that all the threads in a warp execute the same instruction and these threads are chosen sequentially by the SM. Warp is the smallest unit where threads are scheduled and run on an SM. Today for building GPU-accelerated applications, CUDA is the most powerful software development platform. AMD used OpenCL for offloading the massively parallel part to their accelerating unit. Recently CUDA 11 is introduced and supports latest NVidia A100, i.e. Ampere architecture and Arm processors. Besides, it also added performance-optimized libraries and new developer tools.

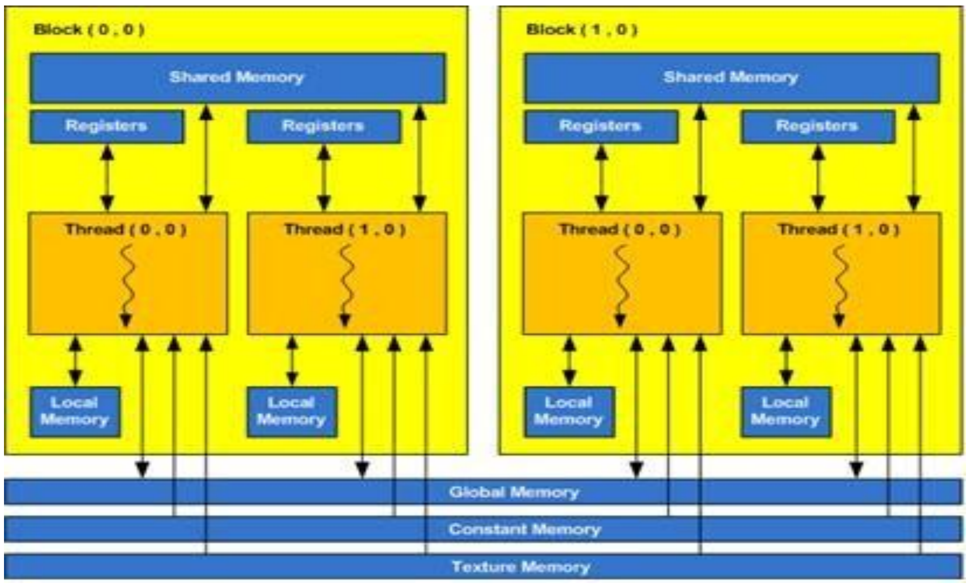


Fig. 2.5 CUDA Programming and Memory Model

CUDA 11 introduced the following capabilities:

[1] Mainly developed for the NVIDIA A100 GPU including scale-up and scale out AI based HPC Data Centers. It also supports DGX A100 and HGX A100 based on A100.

[2] Capability to accelerate mixed-precision matrix operations on different data types, including TF32 and Bfloat16 using new 3rd generation Tensor Cores.

[3] To improve GPU utilization it supports a multi-instance GPU virtualization.

[4] Optimization of library for linear algebra, matrix multiplication, FFTs etc.

[5] For activities like task graphs, asynchronous data movement and fine grained synchronization APIs are improved.

[6] Improvement to the Nsight tool for debugging, profiling, tracing and roofline analysis.

[7] Support heterogeneous architectures with GPUs including X86-64, POWER architectures Arm64 server.

2.3.2 MESSAGE PASSING INTERFACE (MPI)

Message Passing Interface (MPI) is a standard, not a programming language for writing parallel programs. It was developed by consensus of MPI forum having forty organizations including vendors, researchers, scientific library developers and many more and later on it was used as a standard for all in writing parallel applications. This MPI standard aims to establish an efficient, portable and flexible standard for passing messages in parallel programs. In MPI there exists a communication world where a set of processes exist which are communicated to each other through a handler known as MPI_COMM_WORLD.

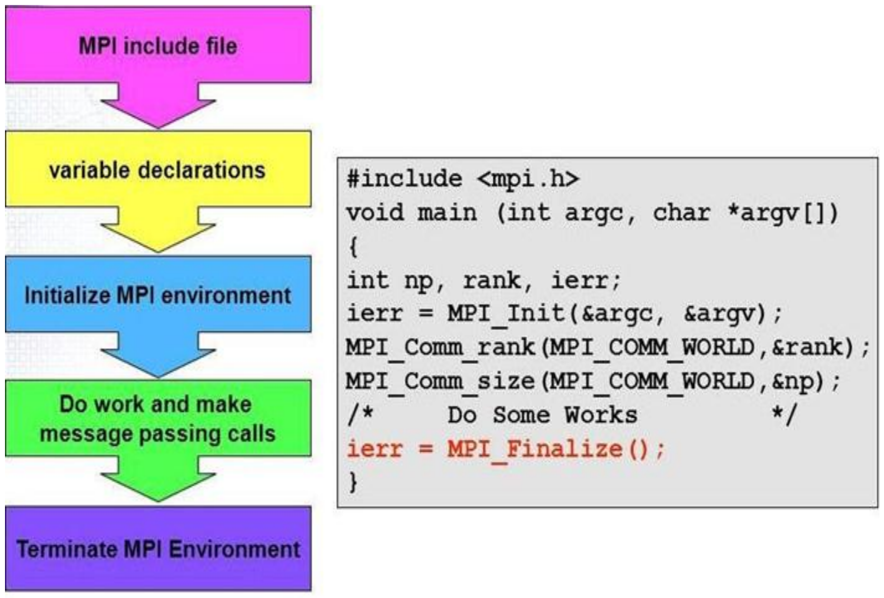
Routine	Purpose/Function
MPI_Init	Initialize MPI
MPI_Finalize	Clean up MPI
MPI_Comm_size	Get size of MPI communicator
MPI_Comm_Rank	Get rank of MPI Communicator
MPI_Reduce	Min, Max, Sum, etc
MPI_Bcast	Send message to everyone
MPI_Allreduce	Reduce, but store result everywhere
MPI_Barrier	Synchronize all tasks by blocking
MPI_Send	Send a message (blocking)
MPI_Recv	Receive a message (blocking)
MPI_Isend	Send a message (non-blocking)
MPI_Irecv	Receive a message (non-blocking)
MPI_Wait	Blocks until message is completed

Table. 2.1 MPI Routines with their function

Initially, MPI was used only in the distributed system means cluster kind of architecture where modules are independent of each other. Later on, it was also used with SMP kind of architecture where processes shared a standard memory. In SMP architecture, OpenMP was used, which is a pragma based language.

Today there are various applications which required hybrid kind of architecture, so MPI is used with SMP, GPU and FPGA architecture. An MPI standard has more than 450 APIs out of which MPI_INIT, MPI_COMM_WORLD, MPI_SEND, MPI_RECIEVE and MPI_FINALIZE are common. “Table 2.1” shows the useful MPI API’s along with their functions.

When an application launches a particular no of processes, then these processes are assigned over a set of computing nodes. Each process is associated with a rank whose range starts from zero and ends in less than one the number of processes. The process with rank 0 is the master or root, and rest processes are slaves. After completion of our work, each process sends the result back to the master or root, and finally, the results are processed and shown to the end-user.


MPI include file
variable declarations
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

```
#include <mpi.h>
void main (int argc, char *argv[])
{
int np, rank, ierr;
ierr = MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&np);
/*      Do Some Works          */
ierr = MPI_Finalize();
}
```

Fig. 2.6 MPI-Skelton

The MPI library provides routines for the processes to communicate and synchronize. The available methods in MPI are Point to Point Communication, Collective Communication, Derived Data type Routine, and Group and Communicator Management Routines. “Fig. 2.6” outlines the steps in a typical MPI program. It is confusion among developers whether MPI is a library or a programming language. The answer to their question is MPI is a standard not a library. While writing parallel program simple includes the #include <mpi.h> to call all API’s available in MPI and rest of the program in either C or FORTRAN.

2.3.3 HYBRID PROGRAMMING

In hybrid programming each process can have multiple threads executing simultaneously i.e. all threads within a process share all MPI objects, Communicators, requests, etc. The hybrid architecture is shown in “Fig. 2.7”.

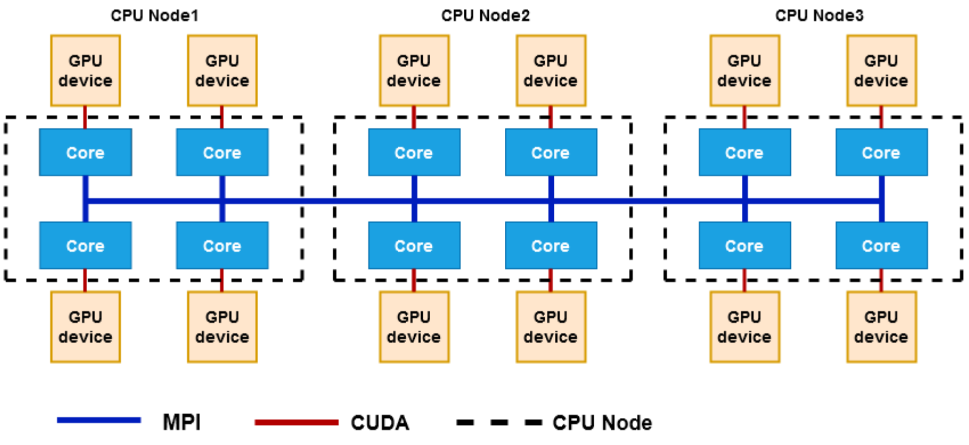


Fig. 2.7 Hybrid Architecture

MPI defines four levels of thread safety:

[1] MPI_THREAD_SINGLE: One thread exists in program

[2] MPI_THREAD_FUNNELED: Multithreaded but only the master thread can make MPI calls. Master is one that calls MPI_Init_thread()

[3] MPI_THREAD_SERIALIZED: Multithreaded, but only one thread can make MPI calls at a time

[4] MPI_THREAD_MULTIPLE: Multithreaded and any thread can make MPI calls at any time.

If more than single thread then use MPI_Init_thread instead of MPI_Init shown below

MPI_Init_thread (int required, int *provided)

Parallel scaling efficiency may be limited (Amdahl’s law) by MPI_THREAD_FUNNLED approach and Moving to MPI_THREAD_MULTIPLE does come at a performance price (and programming challenge).

This is the hybrid programming with SMP kind of architecture where OpenMP is used as a pragma based programming language. In the next section we will discuss CUDA-aware MPI which is suitable for CPU-GPU hybrid architecture.

2.3.4 CUDA-AWARE MPI AND NVIDIA GPU-DIRECT

Generally, MPI communication routines operate on the data that are residing in the host (CPU) memory. This means that an application which needs to communicate data residing in device (GPU) memory has to unusually copy it first to the host (CPU) before passing it to the point to point communication routines such as MPI_Send or MPI_Recieve. To solve this issue, CUDA-aware MPI implementation allows the pointer to data in device memory has been passed to MPI communication and Point to Point calls or routines. By this, a considerable performance boost will take place as it avoids unnecessary copy of data from the host to device and device to host.

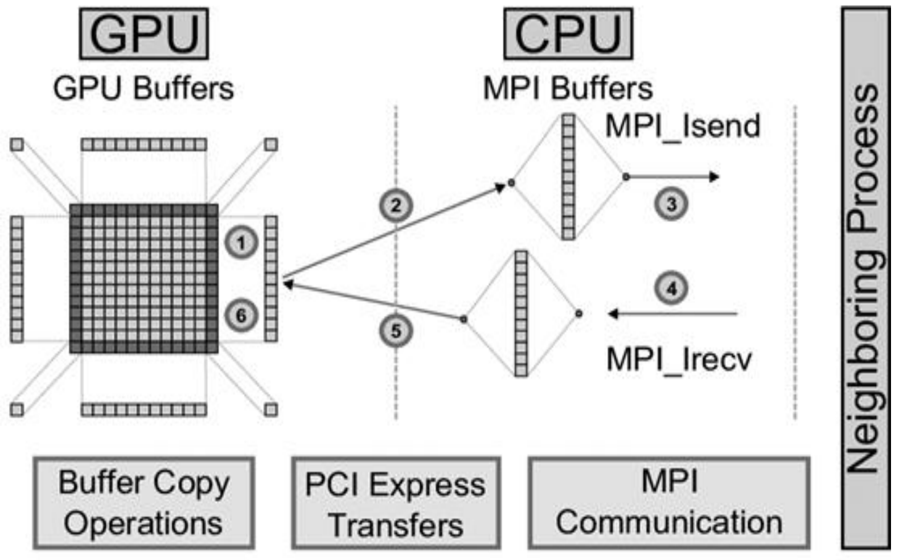


Fig. 2.8 Communication flow between CPU and GPU

“Fig. 2.7” shows the communication architecture involves between CPU and GPU. Another technology that is introduced by NVidia is GPU-Direct.

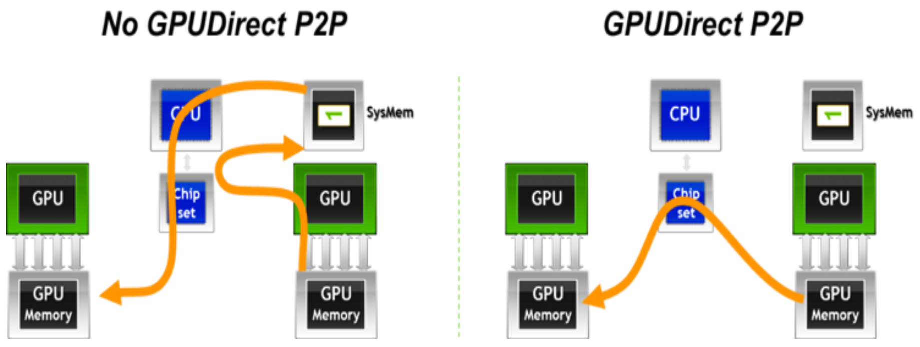


Fig. 2.9 GPU-Direct P2P and No GPUDirect P2P

This technology enables enhanced communication between GPUS and NVidia GPU-Direct Point 2 Point provides accelerated transfer between GPUs on the same node through DMA (Direct Memory Access) transfer technology. NVidia GPU-Direct RDMA uses transfers between GPUs and other PCIe devices through remote direct memory access technology. It eliminates the involvement of host (CPU) as shown in “Fig. 2.8”. CUDA aware MPI significantly improved the performance of MPI point to point and communication routines. The compute requirements in AI and high-performance computing (HPC) drive a need for multi-GPU systems with seamless connections between GPUs, so that they can act as one large accelerator together. But while PCIe is standard, it often creates a bottleneck because of its limited bandwidth. A speedier, more flexible interconnection is required to create the most robust end-to-end computing platform.

2.4 PERFORMANCE OPTIMIZATION OF THE KERNELS

To ensure the optimal performance of the kernels the following ways of optimization are described below:

2.4.1 PACKAGING OF BITS

Since the large sparse system is defined over Galois Field GF(2) then each element of either sparse matrix *A*, solution vector *x* and column vector *b* is zero or one. In contrast to that, the CPU and GPU architectures mend for handling a significant bit of data at once, i.e. either it will control 32 bit or 64-bit integers or floating-point numbers of single and double precision simultaneously. Thinking of treating the individual bit seems the underutilization of the corresponding hardware architecture and hence the solution to solve such problems is to combine the multiple bits and use the bitwise processor operations.

To optimize the performance of the kernel general packaging of 32 successively columns of the augmented matrix, i.e. **[A: b]** together that can fit into the unsigned integer data type. The all packed bits at once performed by these operations. Even in some situation an individual bit is extracted and worked on for particular purposes, for instance, finding in pivot kernel.

2.4.2 INTEGRATED DATA ACCESS AND DATA REUSE

The [A: b] augmented matrix which can be stored either in row-major or column-major format. The row-major format means all the elements belong to a row are stored successively, and column-major format means all the elements belong to a column are stored successively. If we store the augmented matrix in the column major composition, then it is equivalent to operate on the transpose of the augmented matrix and to work on it performs an excellent result for better reusability of data in the row kernel elimination. However, threads of CUDA thread block need to put in one dimension of the row, i.e. pivot-row at once to all single rows which result in the only pass across the row that is the pivot.

Besides it, row-major database, results in consecutive threads that would reduce consecutive items of the similar row, thus pervades consecutive attitude of the pivot row. A single column of the augmented matrix is also accessed at a time by the kernel which computes the pivot row. So it will result in many passes across the mid-row, and hence the augmented matrix single column is also accessed at once by the kernel which computes the pivot row. Thus a large storage column ensures coalesced data access. To store the augmented matrix in a column-major format, a stridden access concludes by extracting the pivot row. Since the most time taking operation is the row elimination because its purview in the entire augmented matrix, therefore, use an essential column-major format and also operates solely on the transpose of the [A: b] augmented matrix.

2.4.3 DATASETS IN VECTORIZED FORM

Proper use of the GPU memory bandwidth is crucial for ensuring maximum performance. If we compare vector and scalar architecture, then obviously vector loads and saves out- put with excellent memory throughput and less number of instruction count and latency compared to scalar loading and saving. Because of that, it is better to access four elements at once from the augmented matrix and packed it into the unit4, i.e. unsigned integer kind of data type. In the next chapter, we explained how the solver based on single GPU optimized to solver based on multiple GPUs by transmitting the data over various sockets or nodes using the MPI standard. We also highlighted more optimizations that have been done to ensure excellent parallel performance. The system has hundreds of thousands of unknowns [21] [22] in order.

2.5 CHALLENGES WITH THE SPARSE MATRIX

The challenge with the sparse matrix is to reduce the substantial memory requirements by accumulating the only non-zero elements. Depending on the sparsity factor, different data structures can be used to save a tremendous amount of memory.

Formats to keep only non-zero elements can be divided into mainly two groups:

[1] The first groups are those that support modification efficiently. For instance, Dictionary of Keys (DOK), List of lists (LOL), or Coordinate List (COO) comes under this category and typically used for constructing the matrices.

[2] The second group that helps efficient access and matrix operations, such as CSC (Compressed Sparse Column) has shown in "Fig. 2.9" and CSR (Compressed Sparse Row) [20] shown in "Fig. 2.10".

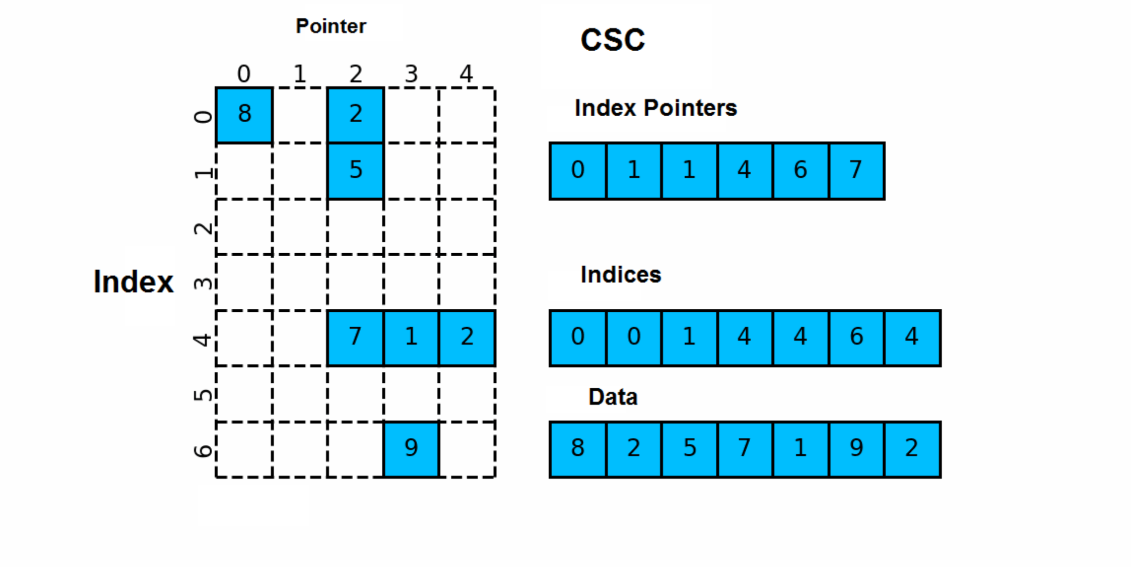


Fig. 2.10 Sparse matrix storage CSC format representation

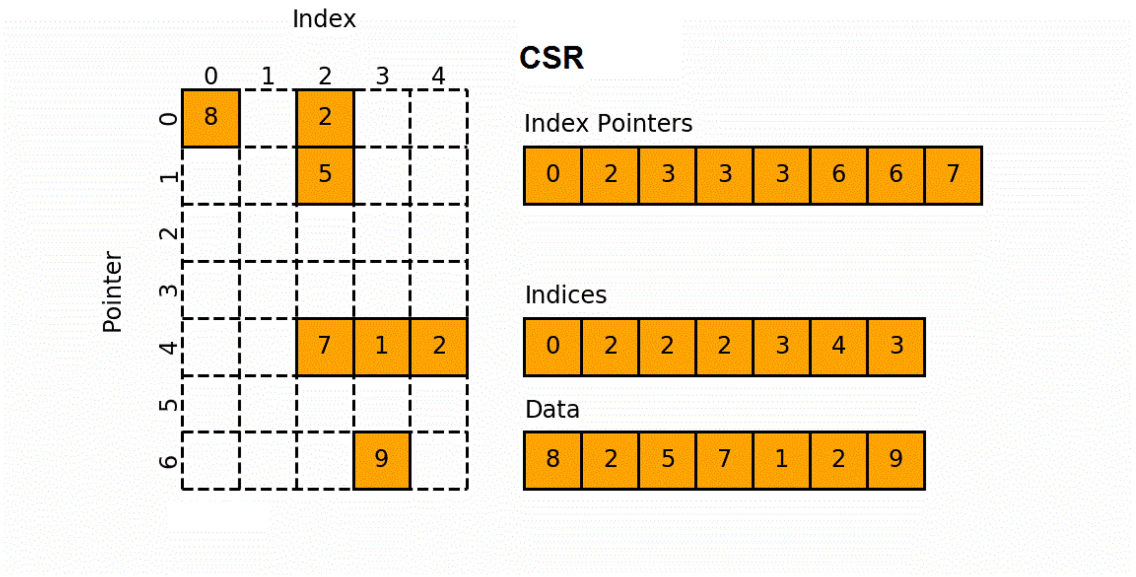


Fig. 2.11 Sparse matrix storage CSR format representation

In this work, we proposed GPU-accelerated optimized Block Lanczos solver for an extensive sparse system of linear equations. It is hard to exploit the more significant degree of parallelism to solve such extensive sparse system of linear equations in a reasonable amount of time. In addition to this, the amount of memory required for such arrangements would not sustain in single node memory. Therefore, we propose the implementation of an efficiently optimized block Lanczos solver for large sparse systems on MIMD (Multiple Instruction Multiple Data).The work that we propose, to the best of our knowledge, is not yet available that describes an optimized, scalable block Lanczos solver for the size- able sparse system over GF(2) and scales efficiently over MIMD architecture with hybrid technology.

2.6 PARALLELIZATION ON MULTIPLE GPUs

The way of implementing such kind of solver in Multi-GPU platform uses MPI. Generally the no. of GPUs used to solve the system of linear equations is equal to the no. of MPI processes initiated. There is one process of MPI attached with each GPU, and the rank of the MPI process, i.e. its local rank is used to fix the GPU affinity.

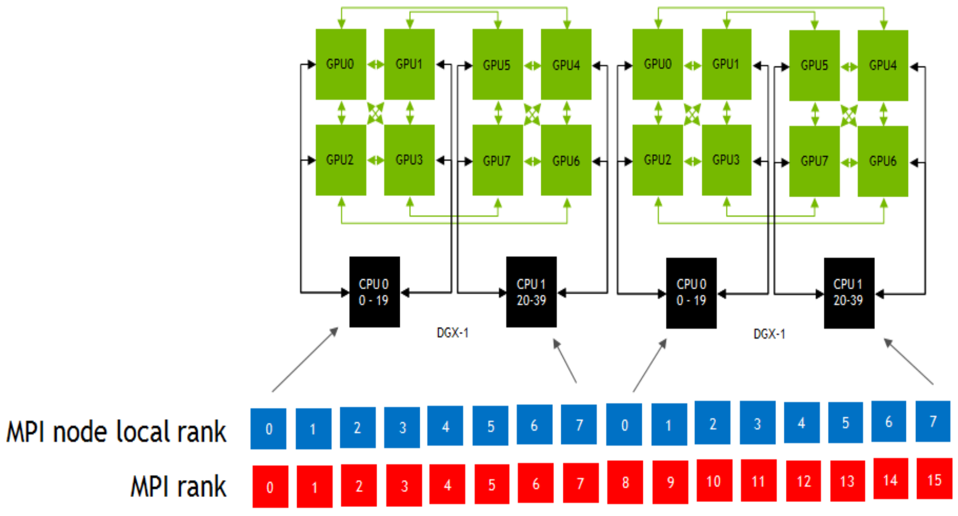


Fig. 2.12 Handling multi-GPU nodes using MPI

How GPUs are attached with the MPI processes shown in “Fig. 2.11” and how the rank of the MPI process, i.e. its local rank is cruised to the Application Programming Interface of CUDA known as cudaSetDevice() to set the value of the GPU-Affinity.

The process associate with rank zero of MPI, also known as a master process, built the transpose of the **[A: b]** i.e. augmented matrix and circulated it to all other processes of MPI row-wise. An individual MPI process calculates in parallel for its input component. To ensure the solver is successful in parallel, the computational load must be balanced, and the communication must be optimized.

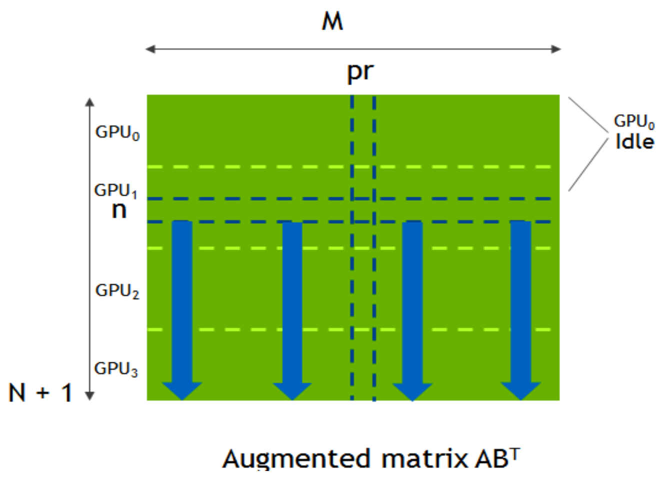


Fig. 2.13 Enhanced block distribution caused load unbalancing, GPU0 idle

"Fig. 2.12" shows how, after a fixed number of iterations, GPU 0 becomes idle when using an ABT distribution of blocks, i.e. ABT is scattered into blocks of a fixed number of rows, and an individual block is being processed by a particular GPU.

CHAPTER 3
MOTIVATION AND CONTRIBUTION

3.1 MOTIVATION

The proposed work for optimizing Block Lanczos solver for the large sparse system over GF(2) is motivated by the following research gaps.

[1] Matrix has a large size and high sparsity, so we require an efficient storage format.

[2] Characteristics of GF(2) must be exploited to boost the time and space efficiency of the algorithm.

[3] Parallelization is a must due to Computation Intensive.

3.2 CONTRIBUTION

To summarize, the main contributions of our research are as follows:

3.2.1 MATRIX FORMAT

[1] An array stores the cumulative number of nonzero entries till a particular row.

[2] A double array stores the column numbers of the nonzero entries for each row.

Example:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

a.) 1 3 6 b.) {0}, {1, 2}, {1, 2, 3}

3.2.2 EXPLOITING GF(2)

Block Lanczos Algorithm itself exploits GF(2) by generating N (computer word size) null vectors per iteration.

[1] Bit packing is done to store all the vectors in the code (each element takes one bit).

[2] Bitwise operations are used for computation (XOR for addition, AND for multiplication) [23].

[3] This not only saves space but also makes computation faster.

3.2.3 PARALLELIZATION

Block Lanczos has three major computational steps.

[1] Matrix-Vector Product.

[2] Vector-Vector Product.

[3] Vector-Vector Addition.

Matrix-Vector Product is the most expensive and dominant computation and needs to be parallelized.

3.2.4 DATA DISTRIBUTION

[1] Each processor reads n/p rows from the input file (n is the number of rows, p is the number of processors).

[2] Matrix Vector product [24] is done in parallel by offloading the massively parallel part to GPUs and result sent back to the root node.

[3] Dependencies are broadcasted and each processor and co-processor modifies part of the matrix it has.

3.2.5 CUDA-AWARE MPI

It uses GPUDirect P2P transfers for communication of data between the GPUs hosted in a node [25] [26]. NVlink technology also enables peer to peer access.

CHAPTER 4
PROPOSED METHODOLOGY FOR OPTIMIZATION

4.1 OPTIMIZATION OF BLOCK LANCZOS ALGORITHM

Given a system of linear equations over GF (2) and the task is to find out the equations linearly dependent on others and remove them. Consider the system of equations where no of equations equals to no of variables and of order O(10^6) or higher with density ranging from 0.2 to 2%. To solve such problem, the possible approach is to select any n equations (where n is the number of variables) and try to solve them. Find the rank of the matrix using LaMacchia and Odlyzko's Algorithm or Diagonally Scaled Wiedemann Algorithm. Then Solve Mx = 0 (i.e. find the null space of M) which is equivalent of finding dependencies in the matrix, e.g.

Input system:

1. *m equations over n variables*
2. *Format Ax = b: A (m x n), b (m x 1)*
3. *Construction of M : Augment b to A and take its transpose to get M (n + 1 x m)*
4. *Solve Mx = 0 to get a null vector x*
5. *Each null vector x gives a dependency relation over the columns of the matrix M*
6. *So we get a dependency relation over equations of the Input System*
7. *Solving Mx = 0*

We use the Block Lanczos Algorithm to solve this equation. The Input is Matrix B (n1 x n2) and to solve: Bx = 0. The original method for Block Lanczos algorithm is roughly split into three steps shown in "Fig 4.1".

- *Pre-processing*
- *Lanczos iteration*
- *Post-processing*

In the pre-processing step, operations such as memory allocation, initialization and loading of the linear system data are done. The Lanczos step involves the iterative part of the code that computes the solution. Finally, in post-processing step solution is written to file.

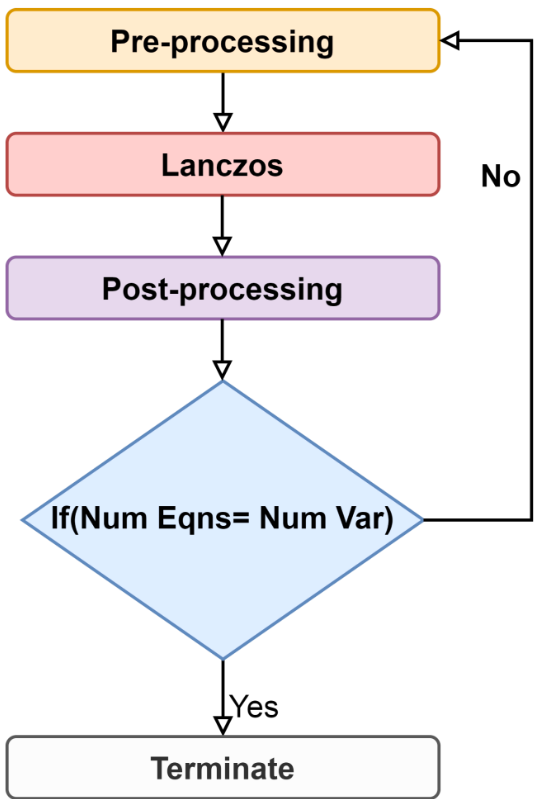


Fig. 4.1 Steps in Block Lanczos Algorithm

The steps for pre-processing are defined as below:

1. *Compute $A = B^T B$ (symmetric matrix).*
2. *Solve $Ax = 0$ where A (n2 x n2).*
3. *Generate random matrix Y (n2 x N) where N is computer word size (generally 32 or 64).*
4. *Solve $AX = AY$ so that column vectors of (X-Y) will belong to the null space of A.*

The procedure for Lanczos algorithm is explained as below:

1. *$V_0 = AY$*
2. *Algorithm terminates when $V_i^T AV_i$, say for i = m*
3. *The two distinguish cases are:*
   a. *If $V_m = 0$ then $AX = AY$ otherwise*
   b. *V_m itself belongs to the null space of A.*

The steps for post-processing are described as below:

1. *Define Z = [X-Y, Vm]*
2. *Compute BZ to find which vectors of Z belong to the null space of B.*

3. *Each null vector gives a dependence relation.*
4. *Remove as many dependent equations as possible.*

The Lanczos step is no longer the most compute-intensive step and Pre-processing and post-processing need to be improved especially at lower densities; both these steps involve a considerable amount of file I/O operations. Therefore, avenues for better I/O performance should be explored. At higher densities (>10%), the Block Lanczos is quite costly in terms of performance. Block Lanczos is the preferred method for solving sparse linear systems of equations, i.e. systems in the form $Ax=b$, where A is a sparse matrix over GF(2).

This work describes the optimization exercise carried out on an existing GPU enabled code for Block Lanczos algorithm. The optimization exercise started with understanding, performance profiling of the existing Block Lanczos method. For benchmarking the performance of the original as well as optimized method, linear systems with up to 30,000 unknowns and densities (number of non-zero elements) from 0.01% to 16% are considered. The optimization work that has been carried out is further explained.

4.2 BETTER TEST DATA GENERATION

The code requires sparse linear systems as input for benchmarking the performance. A program for generating such systems is also included in the original method. However; the program is quite slow and has hardcoded dependency relations. In the optimized method, a new data generating module is added, which is faster and can generate arbitrary relations between columns of the matrix. For random matrix generation, we have to select density, rows and columns parameter. Also manufactured dependencies like col (77) = col (1) XOR col (13) XOR col (51) is used for generation of it. If we talk about completely random matrix generator, then generate non-square matrices and bound to have some dependencies.

4.3 OPTIMIZING SpMV AND SpMTV OPERATIONS

The Lanczos step involves repeated calls to two GPU kernels, the Sparse Matrix-Vector multiplication (SpMV) and Sparse Matrix Transpose Vector multiplication (SpMTV). “Fig. 4.2” shows the percentage of total execution time spent on these two kernels combinedly in the original method. The high percentage share of these two kernels makes them a primary candidate for optimization. Performance of both the kernels is improved with the following techniques. The SpMV and the SpMTV are both matrix-vector multiplication. The matrix-vector multiplication is composed of multiple dot products. Multiple dot products can be executed in parallel.

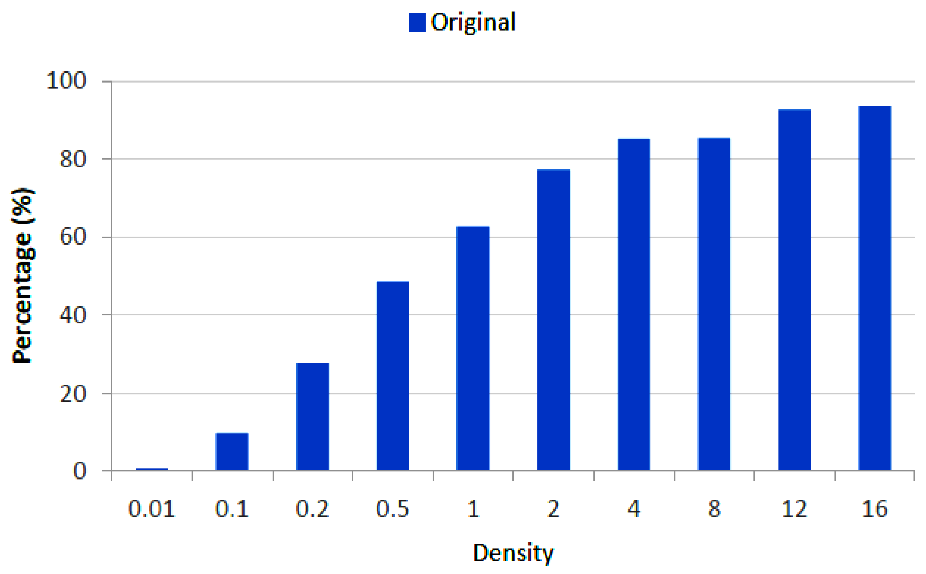


Fig. 4.2 Execution timeshare of SpMV and SpMTV in original method (30,000 unknowns)

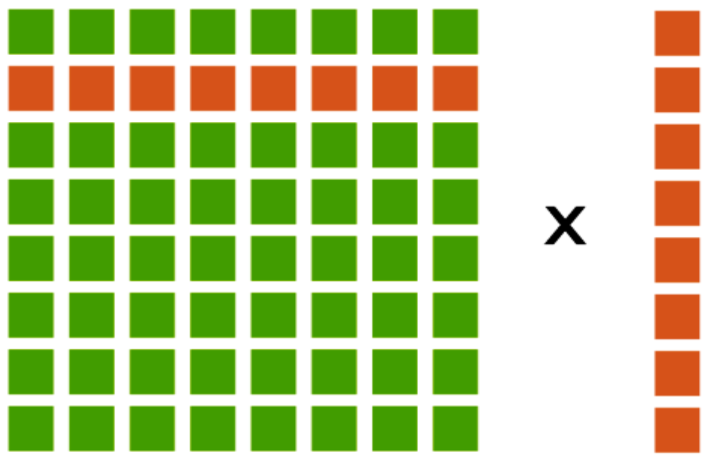


Fig. 4.3 Dot product performed by warp

“Fig. 4.3” depicts dot product of two vectors (in orange color). In the original method, one single dot product is computed per thread. In the new method, the entire warp (vector of 32 threads) is dedicated for computing one dot product. This modification leads to better work distribution among threads of a warp and reduces warp divergence significantly. Warp level approach also results in more coalesced memory access.

4.4 OCCUPANCY OPTIMIZATION

In the optimized method, the launch configurations of kernels are also tweaked formaxi-mum occupancy. Results in terms of higher achieved occupancy can be seen in the profile of the new method. Improvement in occupancy leads to better utilization of GPU resources. The dot product operation involves two steps. First point-wise multiplication and second are adding all multiplication results together. The point-wise multiplication can be done in parallel by each thread of the warp. However, for adding the multiplication results together, is reduction operation and thus threads need to cooperate.

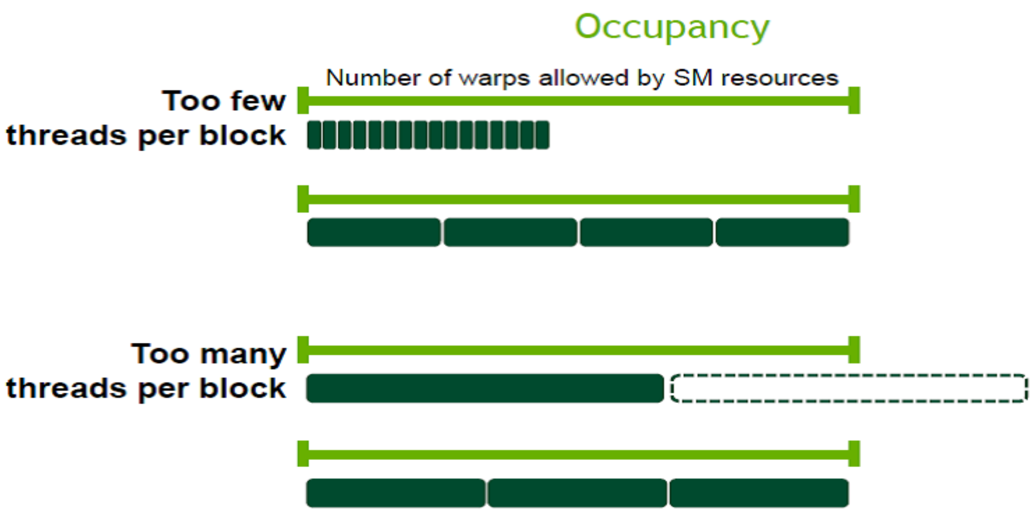

Occupancy
Number of warps allowed by SM resources
Too few threads per block
Too many threads per block

Fig. 4.4 Occupancy optimization through Register/Threads, block size and shared memory

Occupancy is critical to performance. By changing resource, consumption occupancy can change. The factors for optimizing occupancy are Register/threads, block size and shared memory shown in "Fig. 4.4". In addition, changed launch configuration for better occupancy.

4.5 WARP LEVEL REDUCTION OPTIMIZATION

The Kepler architecture introduced four shuffle instructions: *_shfl(), _shfl_down(), _shfl_up(),* and *_shfl_xor()*. "Fig. 4.5" shows shuffle down operation on eight threads. Shuffle instructions allow faster cooperation between threads from same warp.

Effectively, threads can read registers of other threads in the same warp. The reduction operation in a new version of *SpMV* is implemented using shuffle instructions. The shuffle based reduction performs better than even the shared memory atomics based implementation shown in “Fig. 4.6”.

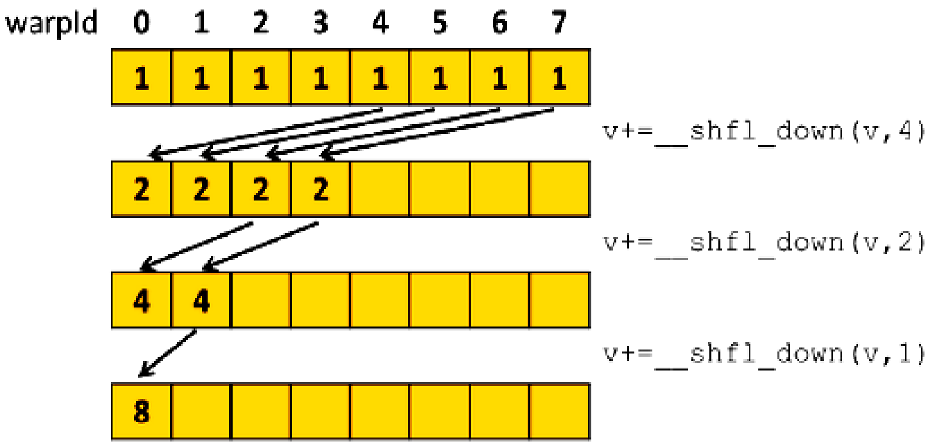
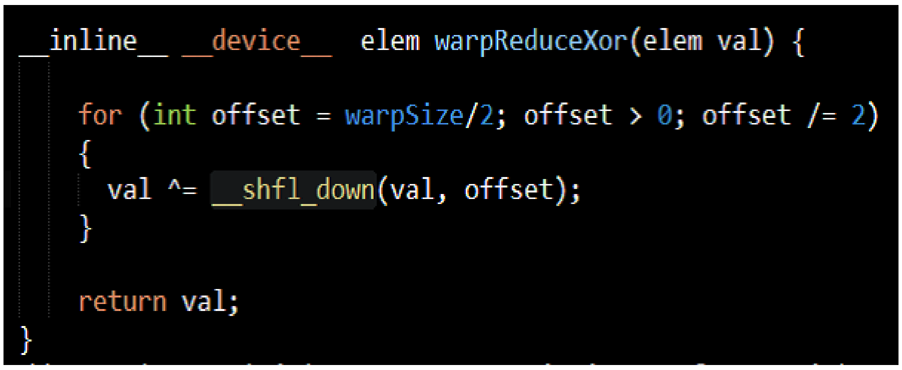


Fig. 4.5 Warp Shuffle Instruction



```
__inline__ __device__  elem warpReduceXor(elem val) {

    for (int offset = warpSize/2; offset > 0; offset /= 2)
    {
      val ^= __shfl_down(val, offset);
    }

    return val;
}
```

Fig. 4.6 Shuffle Based Reduction Operation

4.6 MISCELLANEOUS CHANGES

1. A new Makefile is added to the source, which can be used to build all the required binaries. It also adds an option for running a test case.
2. Reduced number of steps needed for running the 2 and 3. Manual editing of the matrix file is also removed.
3. Fixed bug in CUDA version checking.

CHAPTER 5
RESULTS AND ANALYSIS

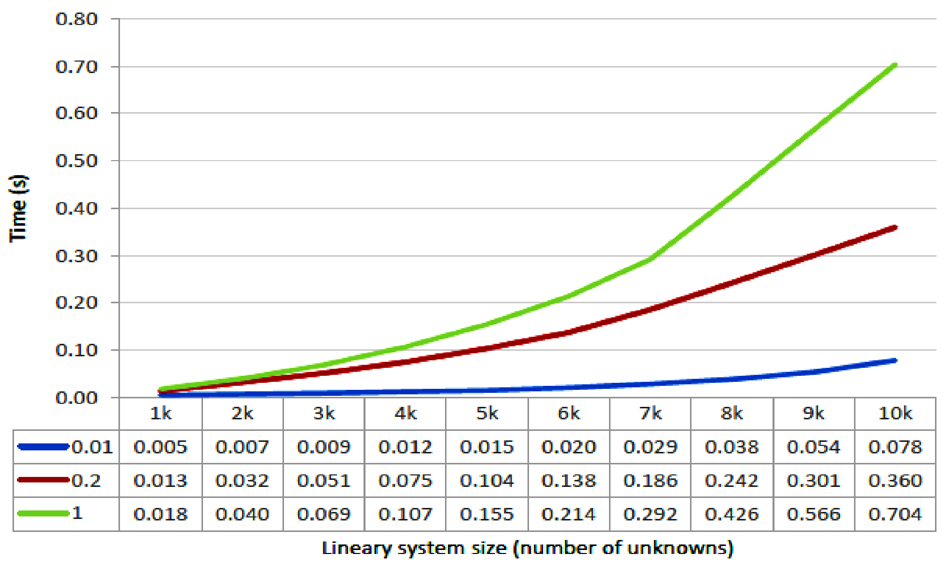
5.1 SYSTEM CONFIGURATION

The configuration used for benchmarking is given below.

- CUDA toolkit 7.5
- CUDA driver version 352.55
- GPU 4 x Tesla K40m
- CPU Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz
- Operating system Cent OS 6.6
- PBS Pro Scheduler
- PBS Compute Manager
- Primary memory (RAM) 128 GB

5.2 RESULTS ANALYSIS

For performance benchmarking of both original and optimized method, linear systems with number of unknowns ranging from 1,000 to 30,000 and density (number of non-zeros) ranging from 0.01% to 16% are generated and then solved. The computed solution is tested against the generated reference solution.



	1k	2k	3k	4k	5k	6k	7k	8k	9k	10k
🟦 0.01	0.005	0.007	0.009	0.012	0.015	0.020	0.029	0.038	0.054	0.078
🟥 0.2	0.013	0.032	0.051	0.075	0.104	0.138	0.186	0.242	0.301	0.360
🟩 1	0.018	0.040	0.069	0.107	0.155	0.214	0.292	0.426	0.566	0.704

Fig. 5.1 Overall wall-clock time needed for solving the linear system for the original method

"Fig. 5.1" depicts the wall-clock time taken by program to solve the linear system of equation of varying sizes and densities. The wall-clock time includes time needed for all the three steps (pre-processing, Lanczos, postprocessing). The wall-clock time increases with increase in size of system (number of unknowns). Increase in density also results in increased wall-clock time.

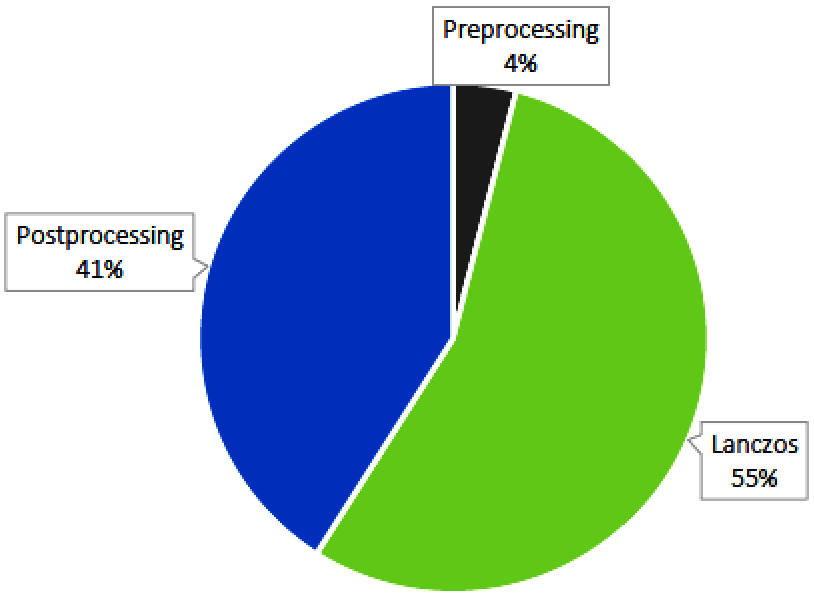
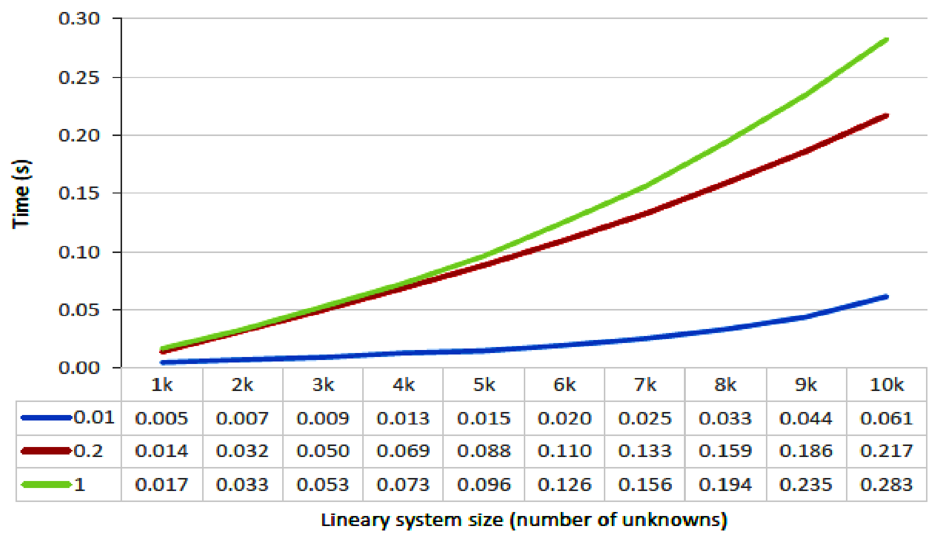


Fig. 5.2 Time share of different steps (original method)

"Fig. 5.2", shows average amount of time spent on the three steps of the program. The Lanczos step is the most time-consuming part of the method.



	1k	2k	3k	4k	5k	6k	7k	8k	9k	10k
0.01	0.005	0.007	0.009	0.013	0.015	0.020	0.025	0.033	0.044	0.061
0.2	0.014	0.032	0.050	0.069	0.088	0.110	0.133	0.159	0.186	0.217
1	0.017	0.033	0.053	0.073	0.096	0.126	0.156	0.194	0.235	0.283

Fig. 5.3 Overall wall-clock time needed for solving the linear system for the optimized method

“Fig. 5.3” shows wall-clock time of the optimized method. The improvement in wall clock time of optimized method is up to **2.5x** over original method.

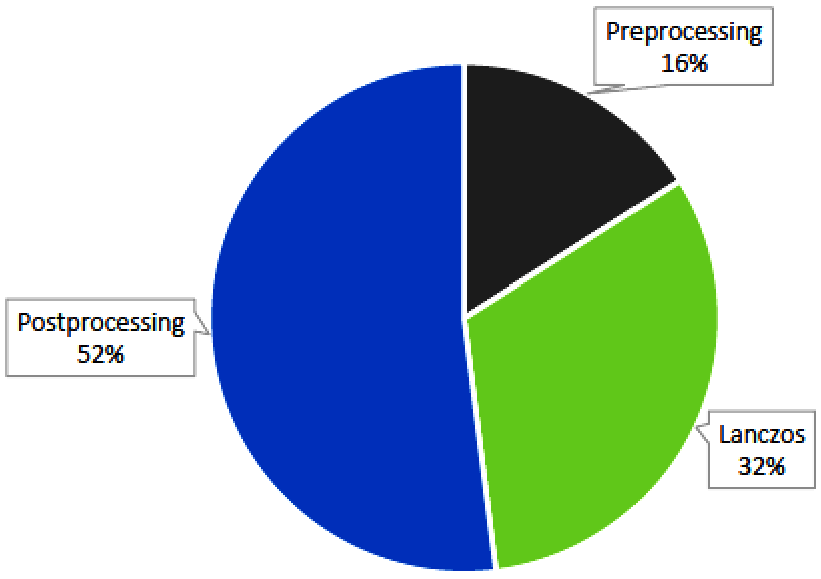


Fig. 5.4 Time share of different steps (optimized method)

Pre-processing and post-processing steps are same in original and optimized method. Therefore, the improvements shown in wall-clock time are from improvements in Lanczos step time. The resulting change in time share is shown in “Fig. 5.4”. It can be seen that, Lanczos step is no longer the most dominant step.

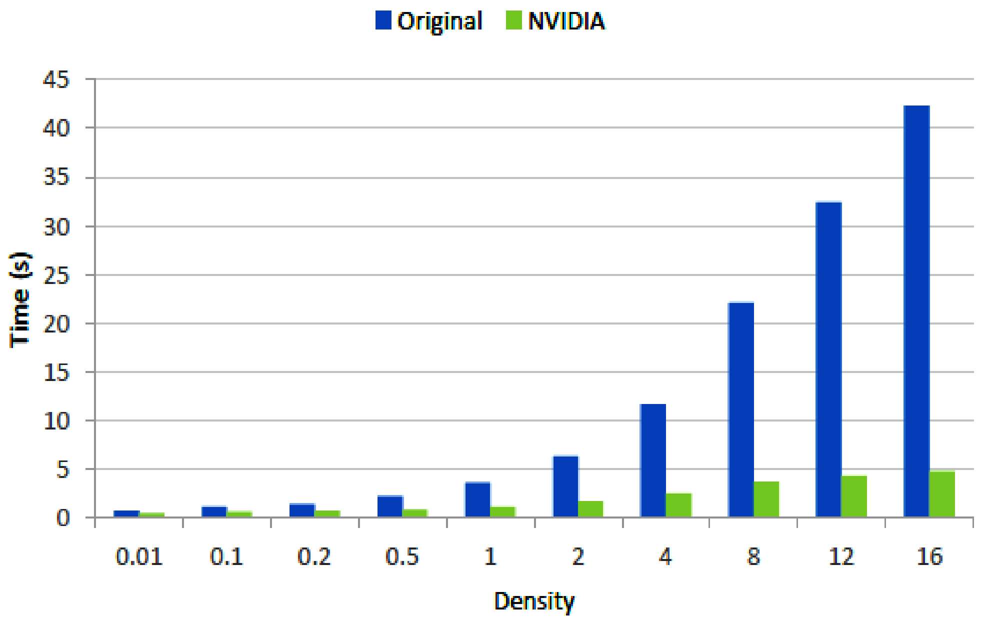


Fig. 5.5 Comparison of Lanczos step timing (20,000 unknowns)

"Fig. 5.5" and "Fig. 5.6" shows exclusive performance gains in the Lanczos step for system with 20,000 and 30,000 unknowns respectively.

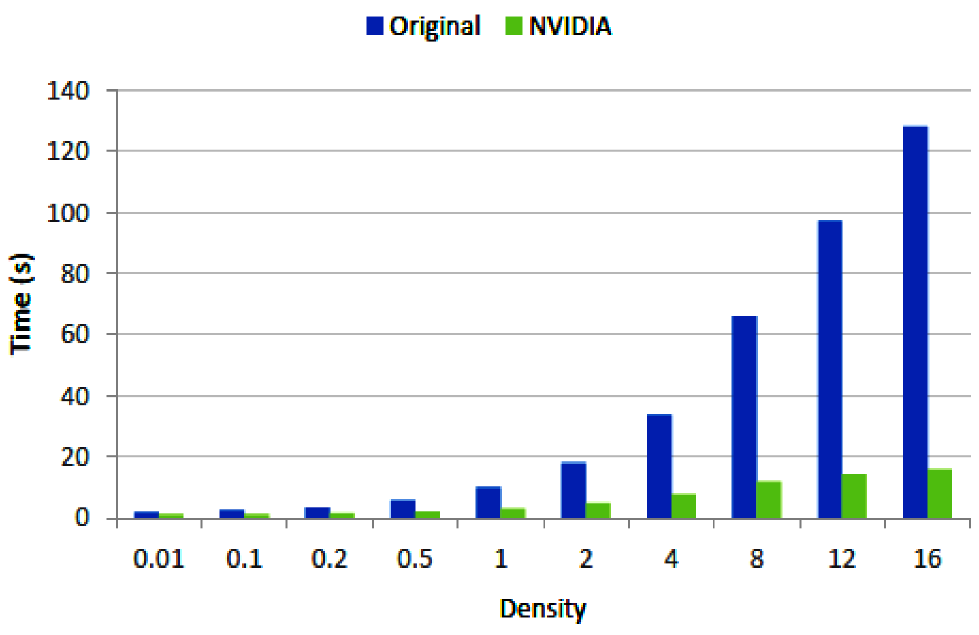


Fig. 5.6 Comparison of Lanczos step timing (30,000 unknowns)

"Fig. 4.2" can be modified to reflect the change in percentage share of SpMV and SpMTV kernels time.

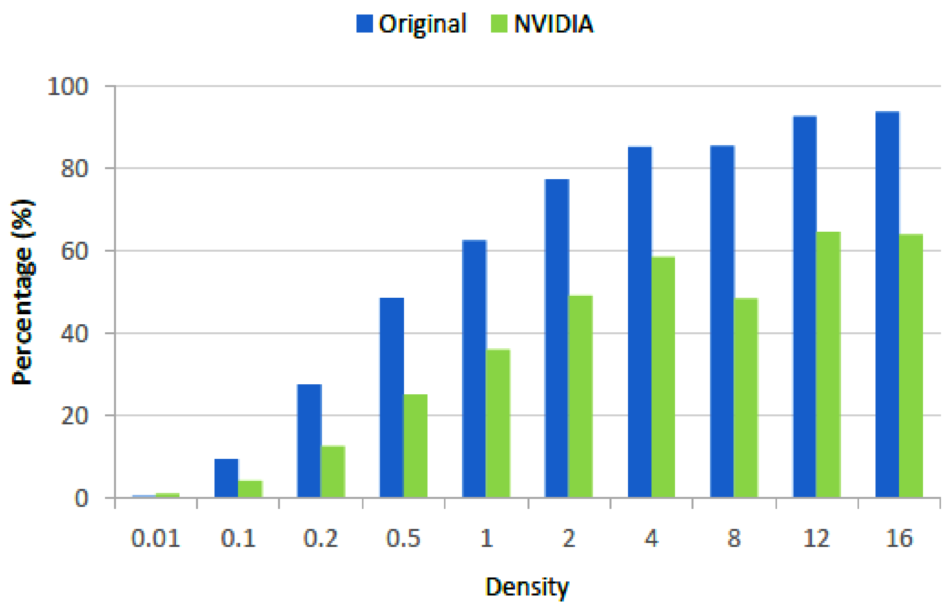


Fig. 5.7 Execution time share of SpMV and SpMTV in original and optimized method (30,000 unknowns)

“Fig. 5.7” shows the percentage share of the two kernels. “Fig. 5.8” shows the gain achieved by of optimized version of method over original method. Note that for this comparison, only execution time for SpMV and SpMTV are considered.

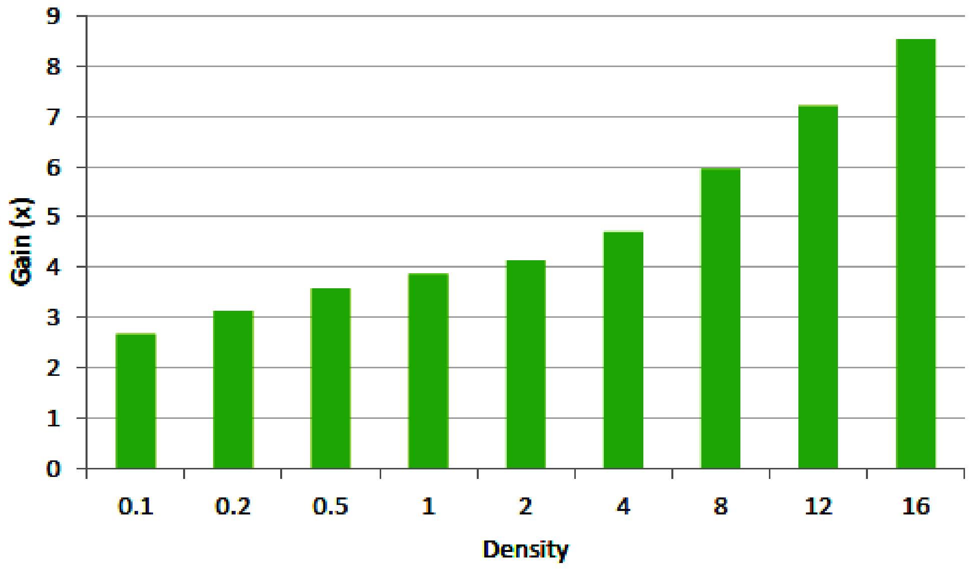


Fig. 5.8 Performance gain in SpMV and SpMTV (30000 unknowns)

The code is also benchmarked on P100 which is based on the latest NVIDIA GPU architecture Pascal. “Table 5.1” shows the performance comparison between P100 and K40.

	P100	**K40**
Compute	5.3 TF DP ,10.6 TF SP, 21.2 TF HP	1.43 TF DP, 4.29 TF SP
Memory	HBM2: 720 GB/s 16 GB	GDDR5 288 GB/s 12 GB
Interconnect	NVLink (up to 8 way) + PCIe Gen3	PCIe Gen 3
Programmability	Page Migration Engine Unified Memory	N/A
CUDA cores	2880	3584
Architecture	Kepler	Pascal
Max power	245 Watt	250 Watt
Base clock	745 MHz	1189 MHz
Fabrication process	28nm	16nm
Memory type	GDDR5	HBM2

Table. 5.1 Pascal 100 vs. Kepler 40 performance

“Fig. 5.9” illustrates the performance gained by running the same code on P100 vs. K40 (Kepler architecture) without any modifications to code. The code on an average executed 2-3x faster on P100 compared to K40.

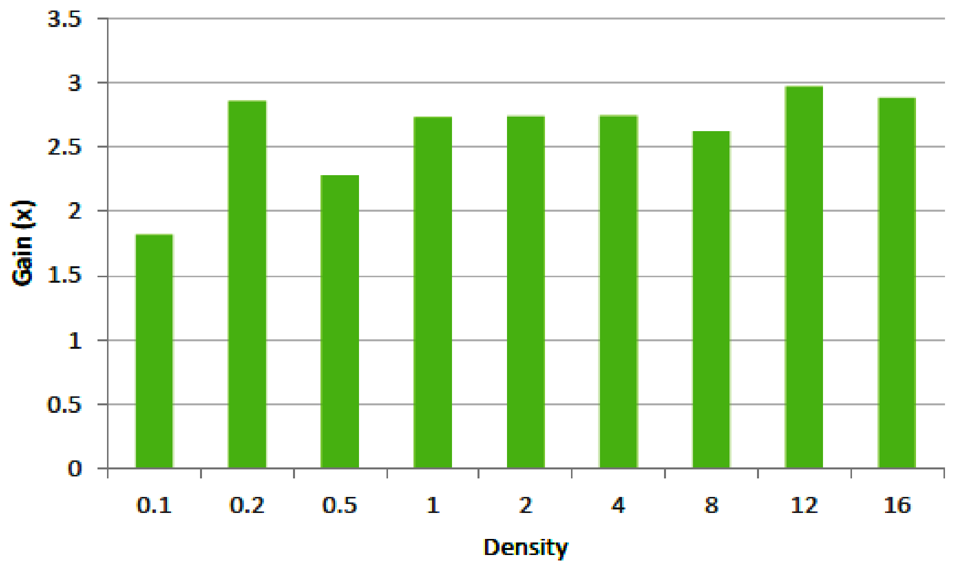


Fig. 5.9 Gains achieved on P100 over K40

CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

The Lanczos step is no longer the most compute-intensive step in the code. Especially at lower densities, Pre-processing and post-processing need to be improved. Both these steps involve a considerable amount of file I/O operations. Therefore, avenues for better I/O performance should be explored. At higher densities (> 10%), the Block Lanczos is quite costly in terms of performance. For such cases, even the dense solver such as Gaussian elimination can be tried. The SpMV and SpMTV are essentially matrix-vector operations. In SpMV the matrix is in the normal format while in SpMTV the matrix is in the transposed format. This change leads to a huge change in the performance of the code. The transpose multiply is **3-4x** slower than the normal multiply. This penalty can be avoided by pre-computing the transpose of the matrix and then calling the same matrix-vector multiply for both operations. The overhead of this approach in terms of execution time is, the time needed for transposing the matrix and in terms of memory is doubling the storage space of the matrix. The computational overhead should be negligible because transposing is roughly equivalent of two calls to SPMV (one round trip of entire matrix data).

6.2 FUTURE SCOPE

[1] SpMTV is 3-4x slower than SPMV, therefore, Pre-compute transpose and use SpMV always.

[2] Tiling multiplication Kernels because Tiling kernels for SpMV and SpMTV are 2-3x quicker is some cases.

[3] Post-processing optimization.

6.3 OTHER POSSIBLE ENHANCEMENT

[1] Refactoring

- Separation of housekeeping and algorithm
- Modular decomposition
- Consistent variable naming

[2] Remove duplication of work

[3] Binary input-output

[4] Reduce the number of steps needed to run the program.

APPENDIX

1. STEPS OF CUDA BASED BLOCK LANCZOS ALGORITHM FOR CSR MATRIX STORAGE FORMAT

To generate a random matrix

gcc –D_FILE_OFFSET_BITS=64 eqngen.c −o eqn

./eqn <num of equations in Ax = b > <density in percent > <num of forced dependencies>

Example: The above mentioned sample file was generated by running the “eqn” as follows:

./eqn 10 50 2

This code generates the matrix A for the system Ax = b (read thesis chapter 1)

This generates following two files

a) EQUATIONS.TXT

First lines contain number of rows and number columns

Then onwards it contains all the elements (zeroes and non-zeroes both) of the matrix A row-wise, with each line containing one row

b) A.TXT

At present the matrix is written in following format:

11 12

55 7

4 6 4 6 7 6 6 6 3 2 5 0

0 1 7 8

…………………………..

…………………………..

…………………………..

so on

We need to convert it into following:

11 12 55

4 6 4 6 7 6 6 6 3 2 5

0 1 7 8

....

…………………………..

…………………………..

so on

We have to make two changes in the A.txt generated by eqngen.c

1) Merger line one and two, and remove the last entry (12 in this case).

2) Remove the last entry from the third line (this entry would be always zero.)

2. STRUCTURE OF FILE CONTAINS THE MATRIX

Name of the File: A.txt

<Line 1>: num Rows num Columns num_Nonzeroes

<Line 2>: length of each row separated by space

<Line 3>: column indices of non-zeroes in first row separated by space

<Line 4>: column indices of non-zeroes in second row separated by space

………………………….

………………………….

………………………….

so on

Example:

11 12 55

4 6 4 6 7 6 6 6 3 2 5

0 1 7 8

5 1 2 3 4 8 9

6 3 4 5 11

7 1 5 8 9 10 11

8 3 4 5 6 7 9 11

9 1 4 5 8 9 10

10 0 3 6 7 10 11

11 0 1 3 4 6 7

12 1 2 6

13 3 4

14 2 4 5 9 10

3. STEPS TO RUN THE APPLICATION

Step 1: Keep the matrix in file "A.txt" in the format mentioned above

Step 2: From "A.txt" generate file "Ab" using following

gcc −D FILE OFFSET BITS=64 dataconverter.c −*o* dataconverter ./dataconverter

It requires "A.txt" to be present in the current directory.

Step 3: Set number of GPUs to use in the header file "cudaBlocklanczos.h"

#define MAX_GPU_COUNT 4

This will use 4 GPUs

Step 4: Compile the CUDA code by running the following script file

. /run.sh

This will generate the executable file with name "blockLanczos.out"

Step 5: Run the executable ./blockLanczos.out

This will generate a file "dependencies.txt" which contains the dependency relations.

The code when run on the above mentioned sample files generated following "dependencies.txt".

8

0 1 2 3 4 5 10 11

(This means column 0, 1, 2, 3, 4, 5, 10 and 11 are linearly dependent)

(Both the entries repeated multiple times)

And (-1 denotes the end of file)

4. FILES AND ITS FUNCTIONALITY RELATED TO SOLVER DEVELOPED FOR THE MULTI-GPU SYSTEM

4.1 cudaBlocklanczos.h

This header file contains the definitions of the data structures used and defines some constants using #define primitive.

4.2 cudaBlocklanczos.cu

This file contains the main () function. It controls all the program flow and calls kernel and the host functions as required

4.3 cudaBlockLanczos_kernel.cu

This file contains all the functions that run on the GPU

4.4 cudaBlockLanczos_host.h

This file contains all the functions which run on the CPU. All the functions in this file can be divided into following two classes based on the purpose that they serve:
Class 1: Functions that are used in the algorithm but are run entirely on the CPU.
Class2: Serial CPU version of the kernel functions contained in "cudaBlockLanczos_kernel.cu". These functions were written so as to test and verify the output of the kernel functions. These functions are no-more needed by the current version of the algorithm but have been provided for the reference and better understanding.

4.5 mem.h

This file contains all the memory management and thread management routines which are used by other files.

4.6 texture.h

This file contains functions written to make use of the texture memory of the GPU. Since the performance while using the texture memory was not good its use has been removed in the current version of the solver.

4.7 Makefile

This contains the rules that compile the code. This file includes a commom makefile that comes along with the NVIDIA_CUDA_SDK. Please make sure that the architecture in that common file is changed to compute capability 1.3 so as to compile our code.

4.8 run.sh

This script file calls the make utility and compiles the code. By default the common make-file provided by NVIDIA_CUDA_SDK generates the executable and puts in its projects bin directory. This run.sh script also copies that executable to the current director. Make sure to put the directory paths in this file appropriately.

5. MISCELLANEOUS FILES RELATED TO SOLVER

eqngen.c: Code to generate the random matrix

dataconverter.c: Code to convert "A.txt" to "Ab"

checker.c: Code to check the validity of the dependency relations given by the solver. It requires the presence of "equations.txt" and "dependencies.txt" in the current directory.

Usage#   . /checker

REFERENCES

[1] Qi Wang, Xiubin Fan, Hongyan Zang, Yu Wang, The Space Complexity Analysis in the General Number Field Sieve Integer Factorization, Theoretical Computer Science, Volume 630, 2016, Pages 76-94, ISSN 0304-3975, https://doi.org/10.1016/j.tcs.2016.03.028.

[2] B. Sengupta, A. Das, Use of SIMD-based data parallelism to speed up sieving in integer-factoring algorithms, IACR Cryptology ePrint Archive (2015) 44.

[3] P. Giorgi, R. Lebreton, Online order basis algorithm and its impact on the block Wiedemann algorithm, in: Proc. 39th Int. Symp. Symbolic and Algebraic Computation (ISSAC14), ACM, 2014, pp. 202209.

[4] A.G. Huang, Parallel Block Wiedemann-based GNFS algorithm for integer factorization, Master thesis, St. Francis Xavier University, Canada (2010).

[5] T. Zhou, J. Jiang, Performance modeling of hyper-scale custom machine for the principal steps in block Wiedemann algorithm, The J. Supercomputing (2016) 123.

[6] Top500 list - November 2017, https://www.top500.org/list/ 2019/11/.

[7] Summit: Oak ridge national laboratory’s next high performance supercomputer, https://www.olcf.ornl.gov/ olcf-resources/compute-systems/summit/.

[8] I.Flesch, A new parallel approach to the Block Lanczos algorithm for finding nullspaces over GF(2), Master thesis, Utrecht University, the Netherlands (2006).

[9] E. Thome, A modified block Lanczos algorithm with fewer vectors, arXiv preprint arXiv: 1604.02277.

[10] http://en.wikipedia.org/wiki/Lanczosalgorithm (2009). Intel Corporation, Technical Report.

[11] Laurence T. Yang, Ying Huang, Jun Feng, Qiwen Pan, Chunsheng Zhu, An

improved parallel block Lanczos algorithm over GF(2) for integer factorization, Information Sciences, Volume 379, 2017, Pages 257-273, ISSN 0020-0255, https://doi.org/10.1016/j.ins.2016.09.052.

[12] T. L. Xu, Block Lanczos-based parallel GNFS algorithm for integer factorization, Master thesis, St. Francis Xavier University, Canada (2007).

[13] L. T. Yang, L. Xu, S.S. Yeo, S. Hussain, An integrated parallel GNFS algorithm for integer factorization based on Linbox Montgomery block Lanczos method over GF(2), Computers & Mathematics with Applications 60 (2) (2010) 338346.

[14] K. Koc and S. N. Arachchige, A fast algorithm for gaussian elimination over gf(2) and its implementation on the gapp. J. of Parallel and Distributed Computing. vol. 13, no. 1, pp. 118122, 1991.

[15] D. Parkinson and M. Wunderlich, A compact algorithm for gaussian elimination over gf(2) implemented on highly parallel computers, Parallel Computing, vol. 1, no. 1, pp. 6573, Aug. 1984.

[16] A. Bogdanov, M. C. Mertens, C. Paar, J. Pelzl, and A. Rupp, A parallel hardware architecture for fast gaussian elimination over gf(2), 14th IEEE Symposium on Field- Programmable Custom Computing Machines, pp. 237248, 2006.

[17] M. R. Albrecht, G. V. Bard, and C. Pernet, Efficient dense gaussian elimination over the finite field with two elements, CoRR, vol. abs/1111.6549, 2011.

[18] M4ri library, https://github.com/malb/m4ri.

[19] W. Bosma, J. Cannon, and C. Playoust, The magma algebra system i: The user language, Journal of Symbolic Computation, vol. 24, no. 3-4, pp. 235265, Oct. 1997.

[20] Aydin Bulu, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Proceedings of the twenty-

first annual symposium on Parallelism in algorithms and architectures (SPAA 09). Association for Computing Machinery, New York, NY, USA, 233244.DOI:https://doi.org/10.1145/1583991.1584053

[21] N. L. Zamarashkin and D. A. Zheltkov, Gpu based acceleration of parallel block lanczos solver, Lobachevski Journal of Mathematics, vol. 39, no. 4, pp. 596602, May 2018.

[22] Gpu acceleration of dense matrix and block operations for lanczos method for systems over large prime finite field, in Supercomputing. RuSCDays, ser. Communications in Computer and Information Science, vol. 793. Springer, 2017, pp. 1426.

[23] I.Gupta, P. Verma, V. Deshpande, N. Vydyanathan and B. Sharma, ”GPU-Accelerated Scalable Solver for Large Linear Systems over Finite Fields,” 2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC), Solan Himachal Pradesh, India, 2018, pp. 324-329, doi: 10.1109/PDGC.2018.8745743.

[24] B. Vastenhouw, R. H. Bisseling, A two-dimensional data distribution method for parallel sparse matrix-vector multiplication, SIAM Review 47 (1) (2004) 6795.

[25] An introduction to cuda-aware mpi, https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/.

[26] FAQ: Running cuda-aware mpi, https://www.openmpi.org/faq/?category=runcuda.

[27] J. Nickolls, I. Buck, M. Garland, and K. Skadron, Scalable parallel programming with cuda, Queue, vol. 6, no. 2, pp. 4053, Mar. 2008.

[28] M. P. Forum, Mpi: A message-passing interface standard, Knoxville, TN, USA, Tech. Rep., 1994.

[29] Nvidia GPUDirect, https://developer.nvidia.com/gpudirect.

[30] C. Reao and F. Silla, “Performance Evaluation of the NVIDIA Pascal GPU Architecture: Early Experiences,” 2016 IEEE 18th International Conference on

High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, NSW, 2016, pp. 1234-1235, doi: 10.1109/HPCC- SmartCity-DSS.2016.0173.

LIST OF PUBLICATION BY CANDIDATE

[1] **Verma. P., Sharma K.,** (2020) *GPU-accelerated Optimization of Block Lanczos Solver for Sparse Linear System*. **IJSTR (International Journal of Scientific & Technology Research) (Scopus Indexed)**
(Status: Published)

[2] **Verma. P., Sharma K.,** (2020) *A Study on Optimization of Sparse and Dense Linear System Solver over GF (2) on GPUs* **ICICSE-2020** (8th International conference on innovations in Computer science & engineering). **(Scopus Indexed)**
(Status: Paper Accepted)

[3] **Verma P., Sharma K., Walia G.S.**, (2020) *Highly Scalable Block Cipher Encryption in Map Reduce-Based Distribution System*. In: Dutta M., Krishna C., Kumar R., Kalra M.(eds) Proceedings of International Conference on IoT Inclusive Life (ICIIL 2019), ITTTR Chandigarh, India. **Lecture Notes in Networks and Systems, vol 116.** Springer, Singapore.
Status: (Presented and Published Online) (Scopus Indexed)

[4] **Verma P., Sharma K., Walia G.S.,** (2020) *Depression Detection Amongst Social Media Users using Machine Learning*. **ICICC-2020** (3rd International Conference on Innovative Computing and Communication.) **(Scopus Indexed)**
Status: (Presented)