January 2018

Developing A New Storage Format And A Warp-Based Spmv Kernel For Configuration Interaction Sparse Matrices On The Gpu

Mohammed Mahmoud

DEVELOPING A NEW STORAGE FORMAT AND A WARP-BASED SpMV KERNEL FOR CONFIGURATION INTERACTION SPARSE MATRICES ON THE GPU

By

Mohammed Mahmoud
Bachelor of Science, Helwan University, 2001
Master of Science, Helwan University, 2006

A Dissertation

Submitted to the Graduate Faculty

of the

University of North Dakota

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Grand Forks, North Dakota

December
2018

This dissertation, submitted by Mohammed Mahmoud in partial fulfillment of the requirements for the Degree of Doctor of Philosophy from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work has been done and is hereby approved.

Hassan Reza, Ph.D.

Mark Hoffmann, Ph.D.

Emanuel Grant. Ph.D.

Wenchen Hu, Ph.D.

Tim Young, Ph.D.

This dissertation is being submitted by the appointed advisory committee as having met all of the requirements of the School of Graduate Studies at the University of North Dakota and is hereby approved.

Grant McGimpsey
Dean of the School of Graduate Studies

December 5, 2018
Date

PERMISSION

Title Developing a New Storage Format and a Warp-Based SpMV Kernel for Configuration Interaction Sparse Matrices on the GPU.

Department Computer Science.

Degree Doctor of Philosophy in Scientific Computing with a Graduate Minor in Chemistry.

In presenting this dissertation in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professors who supervised my dissertation work or, in their absence, by the Chairperson of the department or the dean of the School of Graduate Studies. It is understood that any copying or publication or other use of this dissertation or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of North Dakota in any scholarly use which may be made of any material in my dissertation.

Mohammed Mahmoud
11/13/2018

TABLE OF CONTENTS

LIST OF FIGURES

.

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I would like to thank my graduate advisory committee for their help and support throughout my doctoral research. I would like to thank my advisors for their patience with me. Also, I would like to thank them for their helpful updates and feedback regarding my dissertation and my papers:

Dr. Hassan Reza

Dr. Mark Hoffmann

And the rest of my committee members:

Dr. Emanuel Grant

Dr. Wenchen Hu

Dr. Tim Young

I would like to thank Dr. Travis Desell for his continuous technical and academic support.

I would like to thank the Computer Science department at the University of North Dakota. I would also like to thank the Chemistry Department at the University of North Dakota.

I would like to thank Aaron Bergstrom from the University of North Dakota for his help and support, Aaron is the Advanced Cyberinfrastructure Manager at the University of North Dakota Computational Research Center.

I would like to thank Aliakbar Sepehri and Eric Timian, who are members of our research group (Dr. Hoffmann's group) for their help and support with regard to the Configuration Interaction theory.

I would like to thank the Graduate School at the University of North Dakota for financially supporting this doctoral research.

I would like to thank Dean Grant McGimpsey, the Vice President for Research & Economic Development and Dean of the School of Graduate Studies for his support.

ABSTRACT

Configuration interaction (CI) is a post Hartree–Fock method that is commonly used for solving the nonrelativistic Schrödinger equation for quantum many-electron systems of molecular scale. CI includes instantaneous electron correlation and it can deal with the ground state as well as multiple excited states.

The CI matrix is a sparse matrix, and the bigger the CI matrix, the more electron correlation can be captured. However, due to the large size of the CI sparse matrix that is involved in CI computations, a good amount of the time spent on the eigenvalue computations is associated with the multiplication of the CI sparse matrix by numerous dense vectors, which is basically known as Sparse matrix-vector multiplication (SpMV).

Sparse matrix-vector multiplication (SpMV) can be used to solve diverse-scaled linear systems and eigenvalue problems that exist in numerous and varying scientific applications. One of the scientific applications that SpMV is involved in is Configuration Interaction (CI).

In this work, we have developed a new hybrid approach to deal with CI sparse matrices. The proposed model includes a newly-developed hybrid format for storing CI sparse matrices on the Graphics Processing Unit (GPU). In addition to the new developed format, the proposed model includes the SpMV kernel for multiplying the CI matrix (proposed format) by a vector using the C language and the CUDA platform. The proposed SpMV kernel is a vector kernel that uses the warp approach. We have gauged the newly developed model in terms of two primary factors, memory usage and performance.

Our proposed kernel was compared to the cuSPARSE library and the CSR5 (Compressed Sparse Row 5) format and already outperformed both. Our proposed kernel outperformed the CSR5 format by 250.7% and the cuSPARSE library by 395.1%

Keywords— CI, SpMV, Linear System, GPU, Kernel, CUDA.

CHAPTER 1

INTRODUCTION

Configuration Interaction (CI) is a post Hartree–Fock method that is commonly used for solving the nonrelativistic Schrödinger equation for quantum chemical multi-electron systems. CI has the advantage of dealing with the ground state beside multiple excited states as opposed to other methods that deal only with the ground state.

Whether we perform a full CI or only a limited CI, we must be able to express the Hamiltonian in a matrix form so that we can diagonalize it and obtain the eigenvectors and eigenvalues of interest to us. Figure 1 is a simple representation of the CI sparse matrix that shows the two major components of the matrix, namely the Reference Region and the Expansion Space region.

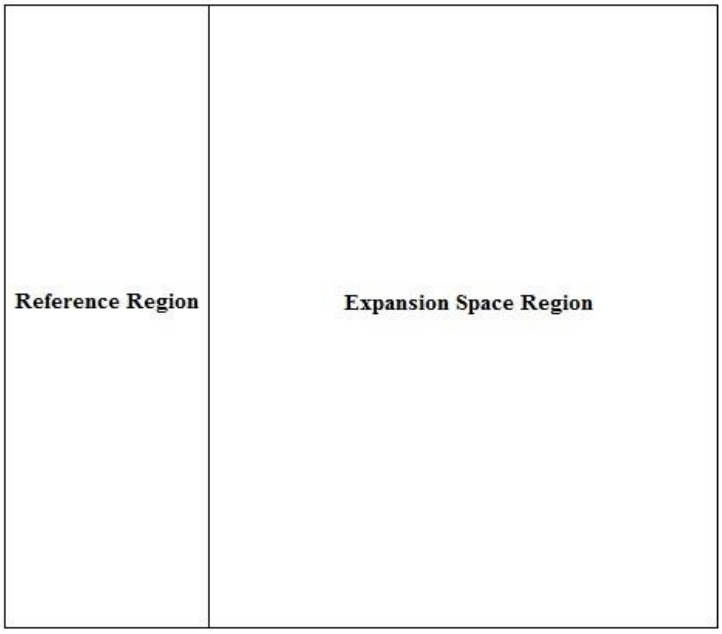

Fig. 1. The Configuration Interaction (CI) Matrix

The Hamiltonian or the CI matrix is a sparse matrix that can be very huge in size, as the CI matrix gets bigger, more electron correlation can be captured from it. Nonetheless, due to the large size of the CI sparse matrix that is involved in CI computations, a good amount of the time that is spent on the eigenvalue computations is already associated with the multiplication of the huge CI sparse matrix by numerous dense vectors, this process is commonly known as Sparse matrix-vector

multiplication (SpMV). Figure 2 is a simple representation of the SpMV operation for CI sparse matrices.

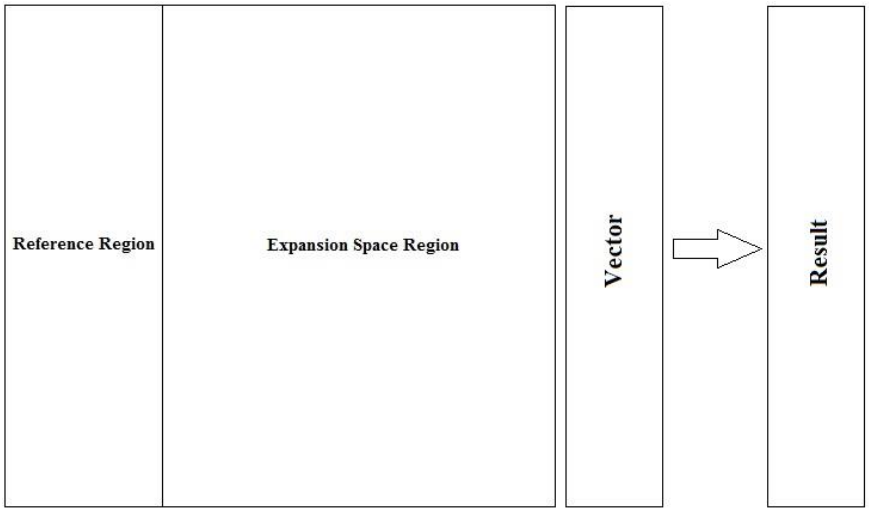

Fig. 2. Sparse matrix-vector multiplication (SpMV) for CI Matrices

Sparse matrix-vector multiplication (SpMV) is one of the most common operations in scientific and high-performance applications. Achieving a good and high SpMV performance is challenging because performance is heavily affected by the density of nonzero entries or their sparsity pattern. As processors are getting increasingly diverse and complex, optimizing SpMV becomes much harder.

CONTRIBUTIONS

- Before the proposed format and the proposed kernel were developed, we were working on designing and developing other algorithms that helped us design and implement the proposed model appropriately. We started out by designing the algorithms and also the C code for the following sparse matrix formats:
    1. The Compressed Sparse Row (CSR) or Compressed Row Storage (CRS) Format.
    2. The ELLPACK (ELL) Format.
    3. The ELLPACK-R (ELL-R) Format.
    4. The Sliced ELLPACK (Sliced ELL) Format.
    5. The Sliced ELLPACK-R (Sliced ELL-R) Format.

- We also worked on developing the algorithms for the GPU SpMV kernels for each of the previously mentioned SpMV formats. Then, we developed the GPU SpMV kernels for those algorithms.

- In this study, a new storage format for storing CI sparse matrices on the GPU was implemented. The proposed format compresses the sparse matrix in a way that saves a considerable amount of GPU memory. Besides, we have developed the SpMV kernel for the proposed format on the GPU. The proposed SpMV kernel is a single SpMV vector kernel that assigns a warp to each single row in the Reference region (ELLPACK format) and assigns another warp to each single row in the Expansion Space region (CSR format). We used the C language [16] and the CUDA platform [9] [10] for implementation. The C language is considered a fast high performance computing programming language as well as easy to use. Numerous programming languages as well as operating systems are built using the C language. The C language supports system calls more conveniently than FORTRAN. The two factors that we are interested in assessing and evaluating are the amount of used memory and performance [5].

- Generally, the use of double-precision is very common in quantum chemistry. Double precision generates more accurate results in addition to smaller errors [63]. The proposed CI

SpMV kernel was first developed using single precision as most of the SpMV kernels are single-precision kernels. Then, we converted the proposed CI SpMV kernel into double-precision for the sake of getting more accurate results.

- Our proposed kernel was compared to the cuSPARSE library and the CSR5 (Compressed Sparse Row 5) format and already outperformed both. Our proposed kernel outperformed the CSR5 format by 250.7% and the cuSPARSE library by 395.1%.

STRUCTURE OF THE DISSERTATION

Chapter 1 is the introductory part to our work. In **Chapter 2**, we will be reviewing some basic concepts, such as GPU, CUDA, and SpMV. In **Chapter 3**, we will be discussing different ways that are used to solve the Schrödinger equation, with an emphasis on Configuration Interaction (CI). In **Chapter 4**, we will be discussing some common/conventional storage formats that are used to store sparse matrices. In **Chapter 5**, we will be reviewing some related work to Sparse matrix-vector multiplication (SpMV). In **Chapter 6**, we will be discussing the proposed model, more specifically, we will be talking about the proposed storage format and the developed SpMV kernel. In **Chapter 7**, we will be discussing the experimental results of comparing the proposed model to the cuSPARSE library and the CSR5 (Compressed Sparse Row 5) format and show how the proposed model outperformed both of them. In **Chapter 8**, we will be discussing Quaternions and show how they relate to Quantum Chemistry in general and also relate to Configuration Interaction specifically. Quaternions can be integrated with Configuration Interaction (CI) since they are optimized to deal with objects that have more interior structure, i.e., CI sparse matrices. The structure of the CI sparse matrix can be more compact using quaternionic representation, thus memory access time will be less. The real challenge that we will deal with when using quaternions for CI matrices is performance. **Chapter 9** will include the conclusions and prospective future work, including the use of quaternions in lieu of real scalar coefficients; a detailed overview of features and widely-used functions.

TERMINOLOGIES

CI: Configuration Interaction.

SpMV: Sparse matrix-vector multiplication.

GPU: Graphics Processing Unit.

CUDA: Compute Unified Device Architecture.

CC: Coupled Cluster.

CSR: Compressed Sparse Row.

CRS: Compressed Row Storage.

ELL: ELLPACK.

ELL-R: ELLPACK-R.

Sliced ELL: Sliced ELLPACK.

Sliced ELL-R: Sliced ELLPACK-R.

CSB: Compressed Sparse Blocks.

THE PROBLEM STATEMENT

One of the main issues in CI computations is the immensely huge size of the CI sparse matrix [13]. The construction of the CI sparse matrix is itself expensive. Some of the elements of the CI matrix are hard to calculate or recreate and some aren't. Different parts of the CI sparse matrix can be calculated using different ways, therefore we have two approaches. One approach would be to pre-calculate the sparse matrix once at the beginning [13]. This option fits some parts of the CI sparse matrix that are hard to recreate. Adopting this approach will be limited by the GPU memory. The other approach would be to calculate the elements of the CI sparse matrix on the fly. This option fits some parts of the CI sparse matrix that are easy to recreate. It's worth mentioning that CPUs are faster than the GPUs when calculating the elements on the fly since CPUs have more complex chips than GPUs. GPUs do branch prediction in a slower fashion than CPUs. CPUs have better caching and more caches than GPUs, whereas GPUs have only global memory (slow), constant memory, local memory, shared memory, and registers. Modern CI calculations are often done on the fly, but this doesn't mean that the entire problem should be done on the fly. Based on the pre-mentioned information, we are going to develop a hybrid approach in order to deal with the CI sparse matrix elements.

We developed a new storage format for storing CI sparse matrices on the GPU. In addition to that, we have implemented on the GPU the SpMV kernel that is based on the newly-developed storage format. The newly-implemented SpMV kernel is a single SpMV vector kernel that dedicates a single warp to every single row in the CI matrix's Reference region and dedicates another warp to every single row in the CI matrix's Expansion Space region. The CSR format is very efficient with regard to using memory (storage space) since it does not require zero-padding like the ELLPACK format. Also, the CSR format provides high performance for SpMV operations on CPUs that have multiple cores [2]. On the other hand, the CSR format provides lower performance than the ELLPACK format on GPUs when it comes to SpMV because of the lack of coalesced access to global memory on the GPU. The ELLPACK format provides higher performance than the CSR format on the GPU [2].

The Reference region of the CI matrix represents most of the electronic structure and performance is an important factor in this part, consequently we are going to store the Reference region in the ELLPACK format. The Expansion Space region that occupies the rest of the CI matrix has a noticeable high sparsity, so storage space is crucial and critical in this part. The CSR format is very powerful regarding storage space, therefore the Expansion Space region will be stored in the CSR format. If we tried to store the Expansion Space region in the ELLPACK format, we will end up with a huge amount of zero-padding especially if one or more rows in the Expansion Space region was not/were not as sparse as the other rows in the same region.

The proposed model can be further extended to be a quaternion-based model. Quaternions (fully described in Chapter 8) are 4-dimensional objects that extend complex numbers and they can be looked at as complex numbers whose components are complex numbers. Quaternions can be integrated with CI since they are optimized to deal with objects that have more interior structure, i.e., CI sparse matrices; they can be used to display objects that have more structure than the point-like nuclei. The structure of the CI sparse matrix can be more compact using quaternionic representation, thus memory access time will be considerably less.

CHAPTER 2

BACKGROUND

GPU:

A Graphics Processing Unit (GPU) is an electronic chip that is designed for extremely fast parallel computations and processing of data. GPUs are more efficient and faster than CPUs at manipulating and processing data, especially computer graphics, because of their highly parallel structure and their capabilities to execute thousands of threads in parallel. A GPU is a specialized electronic chip that is designed to rapidly manipulate and alter memory to speed up the creation of images in a frame buffer intended for output to a display device. GPUs are used in many scientific and technical areas, i.e., embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics and image processing.

NVIDIA introduced the first GPU (GeForce 256) in August, 1999. A CPU includes a few cores that can handle a few threads at a time; on the other hand, a GPU is composed of hundreds of cores that can handle thousands of threads simultaneously in parallel, the matter that leads to faster and more efficient data computations.

Most GPUs are designed for a specific usage, real-time 3D graphics or other mass calculations:

1. **Gaming**
   - GeForce GTX
   - nVidia Titan X[disambiguation needed]
   - Radeon HD
   - Radeon RX
2. **Cloud Gaming**
   - nVidia Grid
   - Radeon Sky
3. **Workstation**
   - nVidia Quadro

- nVidia Titan X
- AMD FirePro
- Radeon Pro

4. **Cloud Workstation**
   - nVidia Tesla
   - AMD FireStream
5. **Artificial Intelligence Cloud**
   - nVidia Tesla
   - Radeon Instinct
6. **Automated/Driverless car**
   - nVidia Drive PX

The GPU architecture is different from the CPU architecture in terms of memory. The GPU has multiple memory types (or levels). The global memory is the slowest memory and it's a read/write memory. The global memory can be accessed by all the threads within the grid. The constant memory is a read-only memory and it's faster than global memory. The constant memory can be accessed by all the threads within the grid, just like the global memory. The shared memory is defined for each block. The shared memory can be accessed by all the threads within the block. Automatic variables are stored in registers. Registers are faster than the shared memory. Registers can be accessed only by the current thread. The compiler sometimes places automatic variables in the local memory; for example: an array created within the kernel is likely to be stored in the local memory. The local memory space resides in the global memory, however other threads can't access it. Local memory accesses have the same high latency and low bandwidth as global memory accesses, so local memory is rarely used. The local memory is only accessible from the current thread. Registers are faster than the shared memory. The shared memory is faster than the constant memory. The constant memory is faster than the global memory. Figure 3 illustrates the memory hierarchy of the GPU:

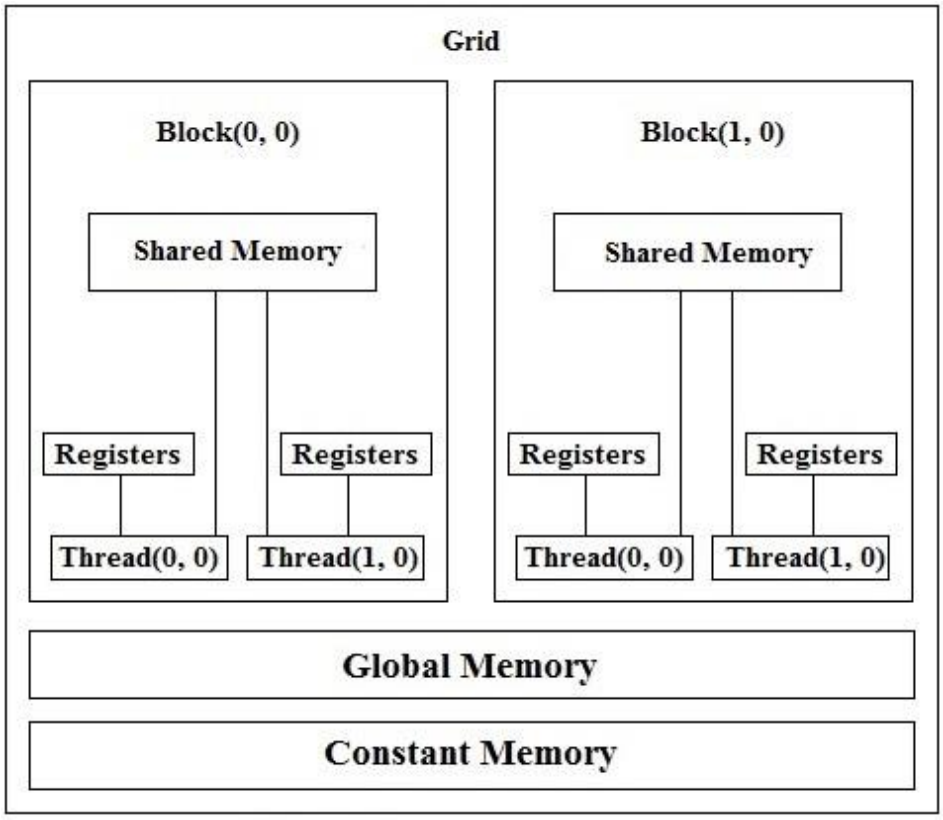

Fig. 3. The GPU Memory Hierarchy [21]

CUDA:

Compute Unified Device Architecture (CUDA) [9] [10] is often mistaken for a programming language, or it might be thought of as an Application Programming Interface (API). In fact, CUDA is more than that. CUDA is a general purpose parallel computing platform created by NVIDIA in 2006. It aids software developers by allowing the compute engine in NVIDIA GPUs to solve a variety of complex problems faster and more efficient than Central Processing Units (CPUs). CUDA allows the software developer to parallelize some portions of the program's code in order to make the program run faster [20]. A kernel is a function (serial program) that will run on the GPU. The CPU launches the kernel on parallel threads.

CUDA can be viewed as a layer that enables software developers to access the GPU for the sake of executing compute kernels. CUDA is designed to work with various programming languages: i.e., C, C++, C#, FORTRAN, Java, Python, etc. [20]. This makes it possible for software developers who are experts in diverse programming languages to use the CUDA platform comfortably.

CUDA has numerous advantages over traditional general-purpose computation on GPUs. Some of these advantages are:

- Faster downloads and readbacks to and from the GPU.
- Scattered reads.
- Unified memory (CUDA 6.0 and above).
- Full support for integer and bitwise operations.
- Unified virtual memory (CUDA 4.0 and above).
- Shared memory – CUDA exposes a fast shared memory region that can be shared among threads.

As already stated, CUDA is a general purpose parallel computing platform created by NVIDIA. One example of using CUDA kernels is given in the following steps:

1. The CPU allocates storage on the GPU (using cudaMalloc()).

2. The CPU copies input data from the CPU to the GPU (using cudaMemcpy()).
3. The CPU launches on the GPU multiple copies of the kernel on parallel threads to process the GPU data. A kernel is a Function (Serial program) that will run on the GPU. The CPU launches the kernel on parallel threads.
4. The CPU copies results back from the GPU to the CPU (using cudaMemcpy()).
5. Use or display the output.

CUDA GPU Support:

- CUDA SDK 6.5: Last Version with support for Compute Capability 1.x (Tesla)
- CUDA SDK 7.5 support for Compute Capability 2.0 – 5.x (Fermi, Kepler, Maxwell)
- CUDA SDK 8.0 support for Compute Capability 2.0 – 6.x (Fermi, Kepler, Maxwell, Pascal), last version with support for Compute Capability 2.x (Fermi)
- CUDA SDK 9.0 support for Compute Capability 3.0 – 7.x (Kepler, Maxwell, Pascal, Volta).

Code1 in Appendix 1 is a simple GPU kernel called AddArrays that is used to add two integer arrays. In this case, the CPU launches on the GPU multiple copies of the AddArrays kernel on parallel threads (one kernel per thread) in order to perform the addition operation. Code 2 in Appendix 1 illustrates how to call the AddArrays kernel that was mentioned before.

Code 3 in Appendix 1 is a host function (a function that runs on the CPU) that is called PrintDeviceProperties(). This function can be used in order to list the GPU properties.

Code 4 in Appendix 1 shows how to call the previous function. The for loop is used to call the PrintDeviceProperties() function for each device.

Although CUDA allows us to run millions of threads or more, programs that run on the GPU aren't million times faster than the CPU for multiple reasons:

- It takes time to copy data from the CPU to the GPU and vice versa.
- CUDA doesn't allow all the threads to run simultaneously on the GPU since it depends on the architecture of the GPU.
- Kernel threads are accessing global memory which is implemented in DRAM (slow and there is a lookup latency).

The following definitions summarize what we have described above:

Host: CPU. It runs the main program.

Device: GPU.

Kernel: Function (Serial program) that will run on the GPU. The CPU which launch the kernel on parallel threads.

GPU: Graphics Processing Unit.

CUDA: Compute Unified Device Architecture.

blockDim.x: The number of threads within the block in the x dimension.

gridDim.x: The number of blocks within the grid in the x dimension.

threadIdx.x: The thread index within the block in the x dimension (0 to (blockDim.x - 1)).

blockIdx.x: The block index within the grid in the x dimension (0 to (gridDim.x - 1)).

In order to compile a CUDA program, run the following command:

nvcc 1.cu -o 1.out (c and c++)

In order to run a CUDA program, run either of the following commands:

./1.out

cuda-memcheck ./1.out

In order to run a CUDA program that accepts 3 arguments, namely arg1, arg2, and arg3, run either of the following commands:

./1.out arg1 arg2 arg3

cuda-memcheck ./1.out arg1 arg2 arg3

If you want to compile and run a CUDA program in just a single step, run the following command:

nvcc 1.cu -run

If you want to compile and run a CUDA program that accepts 3 arguments, namely arg1, arg2, and arg3 in a single step, run the following command:

nvcc 1.cu -run -run-args arg1,arg2,arg3

SpMV:

Sparse matrix-vector multiplication (SpMV) is a substantial computational kernel that has numerous applications in a wide variety of scientific areas and fields. SpMV is the main step of some iterative solvers such as Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES) that can be used to solve sparse linear systems. SpMV has the form of Ax = y [8], where A is an m by n (m rows and n columns) sparse matrix, x is a dense vector of length n, y is a dense vector (The result of SpMV) of length m. A sparse matrix is a matrix in which most of its elements are zeros. A dense vector is a vector in which most of its elements are non-zeros. The sparsity of the matrix is the number of zero-valued elements divided by the total number of elements in the matrix (1 minus the density of the matrix, the sum of the sparsity and the density should equal 100%). SpMV itself isn't a complex algorithm [4], but it might take a huge amount of time especially when we deal with big matrices. When matrices are sparse, the algorithm wastes a great deal of time trying to multiply zero elements by vector elements and the more sparse the matrix is (increasing sparsity), the more time is wasted.

A linear system (system of linear equations) is a collection of two or more linear equations that include the same set of variables. SpMV can be effectively used to solve small-scale to large-scale linear systems [1] [3] (systems of linear equations) and eigenvalue problems. Eigenvalue problems in which only a subset of eigenvalues and eigenvectors are nearly ubiquitous targeting a subset of eigenspace in a big variety of scientific applications. Large eigenvalue problems are solved by iterative techniques which necessitates the need to efficiently improve the SpMV operation on the CPU and the GPU as well. As a matter of fact, improving the SpMV operation is extremely critical to the performance of a variety of scientific applications.

CHAPTER 3

CONFIGURATION INTERACTION (CI)

The Schrödinger Equation:

The Schrödinger equation is a mathematical equation that can be used to study non-relativistic (NR) quantum mechanical systems. It was named after Erwin Schrödinger, a scientist who deduced the equation in 1925. Although there are other formulations of Quantum Chemistry (e.g., Bohmian Mechanics), it is the case that most NR quantum mechanical system of interest to chemical physics and physical systems are most readily described and represented. Hence, the Schrödinger equation is considered the core of any quantum mechanical system of interest.

The Time-independent Schrödinger equation is illustrated below:

$$[-\frac{\overline{h}^2}{2m}\nabla^2 + V(r)]\ \psi(r) = E\ \psi(r)$$

Where r is a 3N-Dimensional vector of the coordinates of particles.

$\hat{H}\psi = E\psi$ (General Form)

Where:

h: Planck's constant ($6.62607004 \times 10^{-34}$ J.s).

$\overline{h} = h / 2\pi$.

m: Generalized Mass.

∇^2: Second Derivative (Laplacian).

V: Potential Energy.

ψ: Wavefunction.

There are multiple techniques and methods that have been used to solve the Schrödinger equation (i.e., figuring out the wavefunction and the energy) for a quantum system. Mean field methods, Hartree-Fock (HF) [17] and density function field (DFT) methods, aka the self-consistent field (SCF) methods are, approximation methods that are widely used to determine the wavefunction and the energy of a quantum system. In the HF method, there is an assumption that the wavefunction can be approximated using a single Slater determinant (A representation of the wavefunction). DFT also uses a single determinant, but is more properly classified as an effective potential method, and as such the HF solution is considered a starting point for many methods that deal with many-electron systems.

Hartree-Fock Method:

F C = S C E

Where:

F: Fock operator.

C: Matrix (a wavefunction) of expansion coefficients.

S: Overlap matrix.

E: Energy.

The electrons are repelled by mean fields of each other.

Post-Hartree-Fock methods are considered a group of methods that were created in order to improve the Hartree-Fock results. Electron correlation is the interaction among electrons in the electronic structure of a quantum system. Electron correlation is an accurate way of including the repulsions among electrons. Electron correlation is considered by post-Hartree-Fock methods as opposed to the Hartree-Fock method, where repulsions are averaged.

Consequently, it is convenient to define

$E_{corr} = \varepsilon_o - E_o$

$E_{corr:}$ Electron Correlation Energy.

ε_o: True (exact) ground state energy.

E_o: HF energy.

Coupled Cluster (CC) method [18] [60] is another numerical method that is used to describe many-body systems. The CC method is considered the most accurate post-Hartree-Fock method for solution to the Schrödinger equation for the ground electron state of molecules near their equilibrium geometries. Instead of the linear expansion of the wavefunction used by Configuration Interaction, CC uses an exponential expansion. CC takes the Hartree-Fock method and builds multi-electron wavefunctions by using the exponential cluster operator to account for electron correlation. One can think of CC as the product of excitations, while Configuration Interaction is the sum of excitations.

Perturbation Theory:

The Møller-Plesset (MP) treatment of electron correlation is based on an approach that is used to treat complex systems that are nearby, in a function space absolutely to a solvable simpler system, called perturbation theory [61]. In perturbation theory, we have:

$$\hat{H} = \hat{H}_o + \hat{V}$$

$$E = E_0 + E_1 + E_2 + \ldots$$

Where:

H: Perturbed Hamiltonian.

H_o: Unperturbed Hamiltonian.

Configuration Interaction:

Turning attention to Configuration Interaction (CI), which is a very useful method in Quantum chemistry has made important contributions to understanding environmental fates of pollutants. Quantum mechanical calculations of molecular electronic structure contribute to the greening of many chemical practices [51]. This could be done via replacing experiments with computation as a variety of complex chemical species and reactions, including the green alternative. Configuration Interaction [19] is a linear method for solving the nonrelativistic Schrödinger equation for quantum chemical multi-electron systems. Relativistic wave equations are applicable to massive particles at high energies and high velocities comparable to the speed of light. Unlike other techniques and methods that can only deal with the ground state, the CI method can deal with the ground state as well as multiple excited states. The term "Configuration" basically refers to the linear combination of Slater determinants that are used for the wavefunction. The term "Interaction" refers to mixing many-electron basis functions. CI uses a linear combination of configuration state functions (CSFs), of which Hartree-Fock method uses only one. The CI approach suffers the size-extensivity problem, which has significantly reduced the use of the CI approach. Although its unique capabilities for excited states, the bigger the CI matrix, the more electron correlation can be captured. However, due to the large size of the CI sparse matrix that is involved in CI computations, a good amount of the time spent on the eigenvalue computations is associated with the multiplication of the CI sparse matrix by numerous vectors which is basically known as SpMV.

The trial CI wave function is written as a linear combination of determinants with the expansion coefficients determined by requiring that the energy should be a minimum. The MOs (Molecular Orbitals) used for building the excited Slater determinants are often taken from a Hartree–Fock calculation. Determinants can be Singly, Doubly, Triply, etc., excited from relative to the HF configuration.

The CI Matrix Elements:

The CI matrix elements H_{ij} can be calculated by a strategy similar to that employed for calculating the energy of a single determinant used for deriving the Hartree–Fock equations [57]. In the CI case, this will involve expanding the determinants in a sum of products of Molecular Orbitals, consequently making it possible to express the CI matrix elements in terms of MO integrals. There are some general features that make many of the CI matrix elements equal to zero.

When the HF wave function is a singlet, this excited determinant often contains 2 more open shells than the reference. The corresponding CI matrix element can be written in terms of integrals over MOs, and the spin dependence can be separated out. If there is a different number of α and β spin-MOs, there will always be at least one integral $\langle a|b\rangle = 0$. That matrix elements between different spin states are zero may be fairly obvious. If we are interested in a singlet wave function, only singlet determinants can enter the expansion with non-zero coefficients. However, if the Hamiltonian operator includes for example the spin–orbit operator, matrix elements between singlet and triplet determinants are not necessarily zero, and the resulting CI wave function will be a mixture of singlet and triplet determinants.

The usual non-relativistic Hamiltonian operator does not contain spin, thus if two determinants have different total spin, the corresponding matrix element is zero. This situation occurs not only the obvious case if an electron is excited from an α spin-MO to a β spin-MO, but also when an excitation is reducible to a sum of functions of spin states.

If the system contains point group symmetry, there are additional CI matrix elements that become zero. The symmetry of a determinant is given as the direct product of the symmetries of the MOs. The Hamiltonian operator always belongs to the totally symmetric representation, thus if two determinants belong to different irreducible representations, the CI matrix element is zero. This is again fairly obvious if the interest is in a state of a specific symmetry, only those determinants that have the correct symmetry can contribute.

The excited Slater determinants are generated by removing electrons from occupied orbitals, and placing them in virtual orbitals. The number of excited Slater Determinants (SDs) is thus a

combinatorial problem, and therefore increases factorially with the number of electrons and basis functions. Consider for example a system such as H_2O with a 6-31G(d) basis. There are 10 electrons and 38 spin-MOs, of which 10 are occupied and 28 are empty.

$$\text{The number of SDs} = 38! / [10! (38 - 10)!]$$

The number of determinants (or CSFs) that can be generated grows wildly with the excitation level! Even if the C_2V symmetry of H_2O is employed, there is still a total of 7536400 singlet CSFs with A_1V symmetry.

Excitation Level (n)	**Total number of CSFs**
1	71
2	2556
3	42596
4	391126
5	2114666
6	7147876
7	15836556
8	24490201
9	29044751
10	30046752

TABLE 1: The number of singlet CSFs as a function of excitation level for H2O with a 6-31G(d) basis [57]

For the sake of developing a computationally tractable model [57], the number of excited determinants in the CI expansion has to be reduced. Truncating the excitation level at one (CI with Singles (CIS)) does not give any improvement over the HF result as all matrix elements between the HF wave function and singly excited determinants are zero. CIS is of the same accuracy as HF for the ground state energy, although higher roots from the secular equations may be used as approximations to excited states. Only doubly excited determinants have matrix elements with the HF wave function different from zero; thus the lowest CI level that gives an improvement over the HF result is to include only doubly excited states, yielding the CI with Doubles (CID) model. Compared with the number of doubly excited determinants, there are relatively few singly excited determinants, and including these gives the CISD method. Computationally, this is only a marginal increase in effort over CID. Although the singly excited determinants have zero matrix elements

with the HF reference, they enter the wave function indirectly as they have non-zero matrix elements with the doubly excited determinants.

Configuration State Functions (CSFs) are a linear combination of Slater Determinants [58]. Molecular Orbitals (MO) are one dimensional objects that are used to create N-dimensional Slater Determinants (or superposition of Slater Determinants) [59]. If we have a product of MOs and apply the anti-symmetrizer operation to it, you will create a single N-dimensional SD.

Small systems, but larger than the 6-31G(d) model problem used above, at the CISD level result in millions of CSFs. The variational problem is to extract one or possibly a few of the lowest eigenvalues and eigenvectors of a matrix the size of millions squared. This cannot be done by standard diagonalization methods where all the eigenvalues are found. There are iterative methods for extracting one, or a few, eigenvalues and eigenvectors of a large matrix.

The CI Matrix:

In this section, the CI matrix is considered in more depth [60]. We will take a linear combinations of Slater Determinants.

$$|\Phi_0> \;=\; |\psi_0> + \sum_{ar} C_a^r\,|\psi_a^r> + \sum_{arb} C_{ab}^{rs}\,|\psi_{ab}^{rs}> + \ldots$$

Where:

$|\Phi_o>$: CI wave function.

$|\Psi_o>$: Hartree-Fock wave function.

C: Some coefficient that describes amplitude of a specific Slater Determinant (Ψ).

All the possible single excitations. An electron is excited from orbital a to orbital r:

$$\sum_{ar} C_a^r\,|\psi_a^r>$$

All the possible double excitations:

$$\sum_{arb} C_{ab}^{rs} \mid \psi_{ab}^{rs} >$$

..: All the way up to n excitations (excite n electrons).

If we take that to N-electron excitations, then we will have what is called Full CI. In the case of Full CI, we have every single electron and every possible Slater determinant using all the orbitals available to us. Full CI is the exact solution to the nonrelativistic Schrödinger equation within the basis set. If we truncate at Single Excitations, then we will have what's called CIS, if we have Double Excitations, then we will have CISD, and this will continue until we reach Full CI.

The Slater determinant is one such basis function i.e., it represents no electron excitation from the reference. The Hamiltonian is expressed in the basis of these Slater determinants. So, we will diagonalize the Hamiltonian matrix and get the lowest eigenvalue from that to be our ground state.

H c = E c

The Hamiltonian matrix (CI matrix, H) will act on a vector of all of our coefficients (c) which will give the energy out (E) and the coefficients back. So, what does the Hamiltonian matrix look like? It looks like the following:

$$H = \begin{bmatrix} <\Psi o|H|\Psi o> & <\Psi o|H|S> & <\Psi o|H|D> \cdots \\ <S|H|\Psi o> & <S|H|S> & <S|H|D> \cdots \\ <D|H|\Psi o> & <S|H|D> & <D|H|D> \cdots \\ . & . & . \\ . & . & . \\ . & . & . \\ . & . & . \end{bmatrix}$$

And, if there is no truncation, this will continue all the way to N Full Excitations <N|H|N>

<Ψo|H|S> is the reference function interacting with single excitations.

<Ψo|H|D> is the reference function interacting with double excitations.

For single excitations, if we have N electrons and K basis functions, the number of single excitations we will have will be:

$$\binom{N}{1}\binom{2K-N}{1} = [N! / ((N - 1)! * 1!)] * [(2K - N)! / ((2K – N – 1)! * 1!)]$$

The previous value can be very large depending on the number of electrons you have (N) and the number of basis functions you have. The total number of Slater Determinants you will get is:

$$\binom{2K-N}{N}$$

For double excitations, if we have N electrons and K basis functions, the number of double excitations we will have will be:

$$\binom{N}{2}\binom{2K-N}{2}$$

So, our task is to calculate these matrix elements and then diagonalize this matrix [62] for the lowest eigenvalue which will be the ground state energy. We have to figure out what the energy of a linear combination of Slater determinants is.

If we have 2 Slater determinants which are exactly identical, for every pair of orbitals, the orbitals can be lined up exactly.

|…m n … >

|…m n … >

This matrix element will be Σ_i<i|h|i> + ½ Σ_{ij} <i j||i j>. This is the same for Hartree–Fock ground state.

If we have 2 Slater determinants that are different by 1 electron. One electron is in orbital m and one electron is in orbital p.

|…m n … >

|…p n … >

This matrix element will be <m|h|p> + Σ_i <m i||p i>

If we have 2 Slater determinants that are different by 2 electrons.

|…m n … >

|…p q … >

This matrix element will be <m n||p q> - one 2-electron composite integral made from 2 usual integrals.

If we have 3 or more electrons different => 0

Example:

Let take H_2 as an example. It has one S orbital on each of the hydrogen atoms. In this case, the smallest basis set will contains only 2 functions and a small number of electrons (2). There are only 6 possible Slater determinants. We will include these 6 Slater determinants in the CI matrix and in this case, the CI matrix will be a 6 by 6 matrix.

$|\psi_0>$ $|\psi_{\bar{1}}^{\bar{2}}>$ $|\psi_{\bar{1}}^{2}>$ $|\psi_{1}^{2}>$ $|\psi_{1}^{\bar{2}}>$ $|\psi_{1\bar{1}}^{2\bar{2}}>$

S S T S T S

(Eliminated) (Eliminated) (Eliminated) (Eliminated)

Before we do that, we can exclude some of them. Our ground state is a singlet. If we have a triplet, it is not going to mix with our ground state determinant. When we integrate out the spins, we will get 0 for that matrix element. So, anything that is a triplet is going to be eliminated since we are concerned with the ground state energy right now. In the ground state, we have 2 electrons that are in the σ_g orbital, g * g gives you the g ground state, so the g ground state is not going to mix with the excited state. The only thing that is left is the double excitation (the first one from the right). The CI matrix will be:

$$\mathrm{H} = \begin{bmatrix} <\Psi_0 | H | \Psi_0> & <\Psi_0 | H | \Psi_{1\bar{1}}^{2\bar{2}}> \\ <\Psi_{1\bar{1}}^{2\bar{2}} | H | \Psi_0> & <\Psi_{1\bar{1}}^{\bar{2}\bar{2}} | H | \Psi_{1\bar{1}}^{2\bar{2}}> \end{bmatrix}$$

Expressing the matrix elements in terms of integrals:

$$<\psi_0 | H | \psi_0> = <1|h|1> + <\bar{1}|h|\bar{1}> + <1\bar{1}||1\bar{1}> = 2h_{11} + j_{11}$$
$$(h_{11}) \qquad (h_{11})$$

$$<1\bar{1}||1\bar{1}> = <1\bar{1}||1\bar{1}> - <11||\bar{1}\bar{1}>$$
$$(j_{11}) \qquad (0)$$

$$<\psi_{1\bar{1}}^{\bar{2}\bar{2}} | H | \psi_{1\bar{1}}^{\bar{2}\bar{2}}> = 2h_{22} + J_{22}$$

$$<\psi_0 | H | \psi_{1\bar{1}}^{2\bar{2}}> = <1\bar{1}||2\bar{2}> = <1\bar{1}|2\bar{2}> - <12||\bar{1}\bar{2}> = K_{12} - 0 = K_{12}$$

The results are in the following 2 X 2 Hamiltonian matrix

$$\mathrm{H} = \begin{bmatrix} 2h_{11} + j_{11} & k_{12} \\ k_{12} & 2h_{22} + j_{22} \end{bmatrix}$$

Because this matrix is so small, the lowest eigenvalue of this matrix will be the Full CI energy. If we subtract out the energy of the Hartree-Fock determinant (2h11 + J11), then we will get the correlation energy.

$$\mathrm{H} - \mathrm{E_o}1 = \begin{bmatrix} 0 & k_{12} \\ k_{12} & 2\Delta \end{bmatrix}$$

$2\Delta = (2h_{22} + J_{22}) - (2h_{11} + J_{11})$

$$\begin{bmatrix} 0 - E_{corr} & k_{12} \\ k_{12} & 2\Delta - E_{corr} \end{bmatrix} = 0$$

$$-E_{corr}(2\Delta - E_{corr}) - k_{12}^2 = 0$$

$$E_{corr}{}^2 - 2\Delta E_{corr} - k_{12}^2 = 0$$

$$E_{corr} = \frac{2\Delta - \sqrt{4\Delta^2 + 4k_{12}^2}}{2}$$

$$E_{corr} = \Delta - \sqrt{\Delta^2 - k_{12}^2}$$

So E_{corr} is the correlation energy within the basis set.

In general, Full CI [55] [56] is very expensive. The scaling for Full CI is on the order of O(N!), where N is the number of electrons. Full CI is exponential with the number of electrons and also with the basis set size. Generally, both the number of electrons and the number of basis functions affect computational costs, but the number of basis functions is usually the limiting factor. The number of basis functions that we use depends on and scales with the number of atoms and the types of atoms. A small calculation would have 14 basis functions per carbon atom. A big calculation on Chromium might have 93 basis functions per atom.

CHAPTER 4

COMMON FORMATS

The format in which the matrix is stored in the CPU memory or the GPU memory affects both the performance of the SpMV operation and the amount of memory used. In this section, we will be discussing some of the features of various sparse matrix storage formats that are used for storing sparse matrices on the CPU or the GPU. We will also be discussing the SpMV kernel that is used along with each format. Besides, we will take a look at the pros and cons of each single storage format in terms of performance and the amount of used memory.

1. The Compressed Sparse Row (CSR) or Compressed Row Storage (CRS) Format [2][11]:

The CSR format compresses a sparse matrix into three vectors:

- The Value vector: contains all the non-zero entries.
- The Column vector: contains column index of each non-zero entry.
- The RowPtr vector: contains the index of the first non-zero entry of each row in the "Value" vector. We add the number of non-zero entries in the sparse matrix as the last element of the RowPtr vector.

In terms of memory, the CSR format is very powerful and efficient since no zero-padding [7] is needed.

In terms of performance, the CSR format is efficient for SpMV operations implemented on CPUs with multiple cores [2]. On the other hand, on the GPU, the CSR format isn't as efficient as the ELLPACK format and shows a worse throughput than the ELLPACK format when it comes to SpMV due to the lack of coalesced access to global memory on the GPU [2].

Example:

Consider the following 6 by 5 sparse matrix A, where:

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

The three vectors will be:

Value = 1, 2, 3, 4, 5, 6, 7, 8

Column = 0, 2, 4, 1, 4, 2, 3, 4

RowPtr = 0, 1, 3, 5, 6, 7, 8

Algorithm 1 describes the SpMV kernel for the CSR format.

Algorithm 1 is the SpMV kernel for the CSR format.

Input:

- **Value Vector:** All the non-zero entries.
- **Column Vector:** The column index of each non-zero entry.
- **Vector:** The vector that the matrix will be multiplied by.
- **RowPtr:** The index of the first non-zero entry of each row in the "Value" array.

Output:

- **Result:** The result of the SpMV process

```
SpMV_CSR(Value, Column, Vector, RowPtr, Result)
 for i ← 0 to ROWS - 1 do
   Start ← RowPtr[i]
   End  ← RowPtr[i + 1]
   for j ← Start to End - 1 do
     Temp ← Temp + (Value[j] * Vector[Column[j]])
   Result[i] ← Temp
   Temp      ← 0.00
```

Algorithm 1. The SpMV kernel for the CSR format

2. The ELLPACK (ELL) Format [2]:

In the ELLPACK (ELL) format, each single row will have the same number of elements. If a row contains fewer non-zero elements, then it will be padded with zeros in order to reach the length of the longest non-zero entry row. For example, consider a 10 by 10 diagonal matrix (or an identity

matrix) with the last row full of non-zero entries. In this case, each single row in the matrix except the last row will be padded with 9 zeros in order to reach the length of the longest non-zero entry row. Thus, instead of storing 19 elements in memory, we store 100 elements in memory.

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

A 10x10 Identity Matrix with the last row full of non-zero entries

The ELLPACK (ELL) format compresses a sparse matrix into two matrices:

- The NonZerosEntries matrix: All the non-zero entries.
- The Column matrix: The column index of each non-zero entry.

In terms of memory, the ELLPACK format introduces a noticeable redundancy since zero-padding [7] is needed in order to reach the length of the longest non-zero entry row. So, the ELLPACK format is less efficient than the CSR format from the memory standpoint.

In terms of performance, the ELLPACK format achieves better performance on the GPU than on the CPU due to the coalesced access to global memory on the GPU [2]. To ensure coalesced global memory access on the GPU, the number of rows has to be a multiple of the block size. This can be achieved by adding extra rows with zero entries [2].

Example:

Consider the following 6 by 5 sparse matrix A, where:

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

The two matrices will be:

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

$$\text{Column} = \begin{bmatrix} 0 & -1 \\ 2 & 4 \\ 1 & 4 \\ 2 & -1 \\ 3 & -1 \\ 4 & -1 \end{bmatrix}$$

Algorithm 2 describes the SpMV kernel for the ELLPACK format (thread per row) [6].

Algorithm 2 is the SpMV kernel for the ELLPACK format.

Input:

- **NonZerosEntries Matrix:** All the non-zero entries.
- **Column Matrix:** The column index of each non-zero entry.
- **Vector:** The vector that the matrix will be multiplied by.
- **MaxNonZeros:** The length of the longest non-zero entry row.

Output:

- **Result:** The result of the SpMV process.

```
SpMV_ELLPACK(NonZerosEntries, Column, Vector, Result, MaxNonZeros)
 for r1 ← 0 to ROWS – 1 do
  for r2 ← 0 to MaxNonZeros – 1 do
    if Column[r1][r2] = -1 then
    exit loop
    Temp = Temp + (NonZerosEntries[r1][r2] * Vector[Column[r1][r2]])
  Result[r1] ← Temp
  Temp      ← 0.00
```

Algorithm 2. The SpMV kernel for the ELLPACK format (thread per row)

3. The ELLPACK-R (ELL-R) Format:

The performance of the SpMV operation can be further improved by adding an extra vector that includes the number of non-zero entries in each row (NonZerosCount vector). In this case, No iterations will be wasted in the loop since only non-zero entries will be involved in calculations [7].

Example:

Consider the following 6 by 5 sparse matrix A, where:

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

The three matrices will be:

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

$$\text{Column} = \begin{bmatrix} 0 & -1 \\ 2 & 4 \\ 1 & 4 \\ 2 & -1 \\ 3 & -1 \\ 4 & -1 \end{bmatrix}$$

$$\text{RowLength} = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Algorithm 3 describes the SpMV kernel for the ELLPACK-R format (thread per row).

Algorithm 3 is the SpMV kernel for the ELLPACK-R format.

Input:

- **NonZerosEntries Matrix:** All the non-zero entries.
- **Column Matrix:** The column index of each non-zero entry.
- **Vector:** The vector that the matrix will be multiplied by.
- **NonZerosCount:** The count of the non-zero entries per row.

Output:

- **Result:** The result of the SpMV process.

```
SpMV_ELLPACK_R(NonZerosEntries, Column, Vector, NonZerosCount, Result)
 for r1 ← 0 to ROWS – 1 do
   End ← NonZerosCount[r1]
   for r2 ← 0 to End – 1 do
     Temp ← Temp + (NonZerosEntries[r1][r2] * Vector[Column[r1][r2]])
   Result[r1] ← Temp
   Temp       ← 0.00
```

Algorithm 3. The SpMV kernel for the ELLPACK-R format (thread per row)

4. The Sliced ELLPACK (Sliced ELL) Format:

The Sliced ELLPACK Format was introduced in order to reduce the redundancy that is inherent in the ELLPACK Format. In this format, the matrix has to be divided first into submatrices (slices) and then each slice is stored in the ELLPACK format. Therefore, the number of extra zero entries (zero-padding) will be determined by the length of the longest non-zero entry row in each slice, rather than in the whole sparse matrix, so we will have less zero-padding, which definitely saves memory.

Example:

Consider the following 6 by 5 sparse matrix A, where:

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

$$\text{Slice1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 3 \end{bmatrix}$$

$$\text{Slice2} = \begin{bmatrix} 0 & 4 & 0 & 0 & 5 \\ 0 & 0 & 6 & 0 & 0 \end{bmatrix}$$

$$\text{Slice3} = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

$$\text{NonZerosEntries1} = \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}$$

$$\text{Column1} = \begin{bmatrix} 0 & -1 \\ 2 & 4 \end{bmatrix}$$

$$\text{NonZerosEntries2} = \begin{bmatrix} 4 & 5 \\ 6 & 0 \end{bmatrix}$$

$$\text{Column2} = \begin{bmatrix} 1 & 4 \\ 2 & -1 \end{bmatrix}$$

NonZerosEntries3 = $\begin{bmatrix}7\\8\end{bmatrix}$

Column3 = $\begin{bmatrix}3\\4\end{bmatrix}$

Notice that we have one column in matrices, NonZerosEntries3 and Column3 instead of two, in other words, no zero-padding has occurred. Algorithm 4 describes the SpMV kernel for the Sliced ELLPACK format (thread per row).

Algorithm 4 is the SpMV kernel for the Sliced ELLPACK format.

Input:

- **NonZerosEntries Matrix:** All the non-zero entries.
- **Column Matrix:** The column index of each non-zero entry.
- **Vector:** The vector that the matrix will be multiplied by.
- **Rows:** The number of rows in the slice.
- **Cols:** The length of the longest non-zero entry row in the slice.

Output:

- **Result:** The result of the SpMV process.

```
SpMV_SlicedELLPACK(NonZerosEntries, Column, Vector, Result, Rows, Cols)
 for r1 ← 0 to Rows – 1 do
  for r2 ← 0 to Cols – 1 do
   if Column[r1][r2] = -1 then
   exit loop
   Temp ← Temp + (NonZerosEntries[r1][r2] * Vector[Column[r1][r2]])
  Result[r1] ← Temp
  Temp      ← 0.00
```

Algorithm 4. The SpMV kernel for the Sliced ELLPACK format (thread per row)

5. The Sliced ELLPACK-R (Sliced ELL-R) Format:

It's worth mentioning that the Sliced ELLPACK format can be even further extended to be Sliced ELLPACK-R. In this case, the performance of the SpMV operation will be improved by adding an extra vector that counts the number of non-zeros in each row (NonZerosCount vector), just like the ELLPACK-R format. Algorithm 5 describes the SpMV kernel for the Sliced ELLPACK-R format (thread per row).

Algorithm 5 is the SpMV kernel for the Sliced ELLPACK-R format.

Input:

- **NonZerosEntries Matrix:** All the non-zero entries.
- **Column Matrix:** The column index of each non-zero entry.
- **Vector:** The vector that the matrix will be multiplied by.
- **NonZerosCount:** The count of the non-zero entries per row in the slice.
- **Rows:** The number of rows in the slice.

Output:

- **Result:** The result of the SpMV process.

```
SpMV_SlicedELLPACK_R(NonZerosEntries, Column, Vector, NonZerosCount, Result, Rows)
 for r1 ← 0 to Rows – 1 do
   End ← NonZerosCount[r1]
   for r2 ← 0 to End – 1 do
     Temp ← Temp + (NonZerosEntries[r1][r2] * Vector[Column[r1][r2]])
   Result[r1] ← Temp
   Temp      ← 0.00
```

Algorithm 5. The SpMV kernel for the Sliced ELLPACK-R format (thread per row)

6. The Compressed Sparse Blocks (CSB) Format:

The Compressed Sparse Blocks (CSB) format [12] [13] [14] is used for storing sparse matrices. The CSB format partitions the n X n matrix into n^2/z^2 equal-sized z X z square blocks using a block size parameter z. The CSB format consists of the following:

- The **Value** vector: The Value vector is of length nnz. It stored all the non-zero elements of the sparse matrix.
- The **row_idx** and **col_idx** vectors: They track the row index and the column index of each non-zero entry inside the Value vector with regard to the block, not the whole entire matrix. So row_idx and col_idx range from 0 to z – 1.
- The **Block_ptr** vector: It stores the index of the first non-zero entry of each block inside the Value vector.

A comparison among the common formats in terms of memory and performance on the GPU is illustrated in the following table (Table 2). Note that the information in the following table might be different depending on the nature of the matrix and the sparsity pattern of it.

Format	Memory	Performance
CSR	Good	Bad
ELLPACK (ELL)	Bad (Zero Padding)	Good
ELLPACK-R (ELL-R)	Bad (Zero Padding)	Better than ELLPACK
Sliced ELLPACK	Bad (Zero Padding), but better than ELLPACK	Good
Sliced ELLPACK-R	Bad (Zero Padding), but better than ELLPACK	Better than ELLPACK

TABLE 2: Common Formats

In the proposed model, we stored the CI Reference region, which is more dense in the ELLPACK format instead of the ELLPACK-R format although the ELLPACK-R format is generally faster than the ELLPACK format. We applied the warp technique to the proposed SpMV kernel, we assigned a warp to each single row in the Reference region and in order to efficiently do that, each row has to be a multiple of 32, consequently we padded the rows in the Reference region with zeros in order to achieve that, therefore, we used the ELLPACK format for the Reference region instead on the ELLPACK-R format. We did not use the Sliced ELLPACK format or the Sliced ELLPACK-R format since there will be an added overhead that is caused by adding the sub-results of the submatrices (slices) in order to get the final result. The Expansion Space region that is extremely sparse, is stored in the CSR format that uses memory very efficiently.

CHAPTER 5

RELATED WORK

SpMV is noticeably a rich area of research that is heavily studied. A lot of work that has been done with regard to SpMV wasn't designed specifically for CI sparse matrices. CI sparse matrices are a special case of sparse matrices in terms of the structure of the sparse matrix as well as the sparsity pattern of the non-zero elements inside the matrix.

In this section, we will review some of the related work to SpMV, some of these studies generally apply to the majority of sparse matrices and others apply to special types of sparse matrices.

F. Vázquez and et al. [22] proposed a new format called ELLR-T. This format is an extension of the ELLPACK-R (ELL-R) format (The ELLPACK-R format was discussed in part 3 of the COMMON FORMATS section). In the ELLR-T format, there is a preprocessing step that has to be performed first [2]. The elements of the NonZerosEntries matrix and the Column matrix are permuted. Also, each row has to be a multiple of 16. In the ELLR-T format, multiple threads (T = 1, 2, 4, 8, 16, 32) operate on single row while executing the SpMV operation. The value T (number of threads per row) can change in order to obtain the best performance with different types of sparse matrices. The ELLR-T format achieves a higher overall performance due to the coalesced and aligned access to global memory.

Due to the fact that SpMV kernels are heavily used in scientific computing, M. M. Baskaran [23] tried to optimize SpMV kernels on GPUs. They tried to evaluate various issues and challenges that deal with and relate to developing robust and high performance kernels for the SpMV operation on NVIDIA GPUs using the CUDA platform. They have proposed a framework that includes both compile-time and run-time optimizations. The compile-time optimizer applies the following optimizations to SpMV kernels that execute on the GPU: exploiting synchronization-free parallelism, optimized thread mapping, optimized global memory access, and exploiting data reuse. There is an optional runtime optimizer. They have created a new blocked storage format for storing and accessing the elements of a sparse matrix in an optimized manner from the GPU

memories. They evaluated their optimizations over two classes of NVIDIA GPU chips, namely, GeForce 8800 GTX and GeForce GTX 280 using a large set of sparse matrices derived from real applications. They compared the performance of their approach with that of existing parallel SpMV implementations. Their optimization techniques resulted in significant performance improvements on both GPUs over existing parallel SpMV implementations by a factor of 2 to 4. Using this framework, they were able to achieve peak SpMV performance that is 70% of the performance observed for SpMV computations using dense matrices stored in sparse format.

M. R. Hugues and et al. [24] proposed an evaluation of several sparse matrix formats that are used by the GPU SpMV kernel. They analyzed the performance when having multiple sparse matrices with a strong distribution of non-zero elements. The sparse matrices that have been used in the study are from the University of Florida. These sparse matrices have different sizes as well as different number of non-zeros. The best results were obtained by using the ELL format for sparse matrices, however various formats that deal with sparse matrices that have a strong distribution of non-zero elements achieve approximately the same performance (2.0 GFLOPS) which is considered poor performance. This study concluded that the performance of SpMV computation depends on the format of the sparse matrix as well as the data structure of the sparse matrix.

B. Neelima and et al. [25] proposed a new format for storing square sparse matrices on the GPU. The newly-developed format can give 2x to 5x performance improvement compared to the CSR format. The CSR format doesn't benefit from global coalescing feature of the GPU. Also, the GPU is underutilized if the number of non-zero elements per row is less than 32 (warp size). The proposed format (CSPR format) reduces the SpMV operation to a constant time and it uses a single data structure (ind vector) as it embeds the row information into column information, hence it can also optimize the memory transfer between the CPU and the GPU. Generally, the CSPR format can be applied to any sparse matrix, but better performance can be achieved when using sparse matrices with a large number of rows with a minimum number of non-zero elements per row and few dense rows distributed in the center. The proposed format can get 2x to 54x performance improvement compared to other sparse matrix formats (CSR, COO, and CSR vector).

A. Monakov and et al. [26] proposed a new storage format that adapts to different sparse matrices in order to improve the performance of the SpMV operation on the GPU. The storage format is called sliced ELLPACK. This format takes parameter S which represents the slice size. The sparse matrix is partitioned into strips (slices or partitions) of length S (S adjacent rows) and each strip is stored in the known ELLPACK format. Sparse matrices that have big variations with regard to the number of non-zero elements per row have less padding (extra zeros) than the ELLPACK format and hence, use less memory. This format uses a single SpMV kernel as opposed to the hybrid ELLPACK/COO format [15], therefore it achieves more performance. The ability to allocate a variable number of threads per row helps to adapt to various different matrices. It's possible to allocate one thread per row in extremely sparse matrices or allocate multiple threads per row.

Some of the developed formats are actually built on one of the conventional (common) formats. For example, J. L. Greathouse and et al. [27] proposed a storage format called the CSR-Adaptive format that is based on the common CSR format (good at memory storage) that maps well to GPUs. The CSR format is frequently used in order to store sparse matrices on the GPU, nonetheless when it comes to the SpMV kernel on the GPU, this format has poor performance because of accessing memory in irregular patterns. The CSR-Adaptive format achieves an average speed up of 14.7x over the conventional CSR format.

F. Vázquez and et al. [28] and J. a Mart and et al. [30] proposes a new storage format for storing sparse matrices on NVIDIA GPUs. This format is called ELLPACK-R. They also implemented the SpMV kernel based on the newly-developed format. The ELLPACK-R format doesn't include conditional branches since it includes the rl (row length) array, in other words the ELLPACK-R format gets rid of useless computations since only non-zero elements of each row in the sparse matrix are considered. They compared the ELLPACK-R format to a variety of other sparse matrix storage formats using different test matrices. Although the performance depends on the pattern of the sparse matrix, the implementation based on ELLPACK-R format achieves noticeable higher performance. The ELLPACK-R format achieves the highest performance when it deals with

matrices of high dimensions, nonetheless the performance gets lower with sparse matrices of small dimensions.

X. Liu and et al. [29] proposed a new SpMV kernel on Intel Xeon Phi Coprocessor that is called Knights Corner (KNC). First, they tested the architecture with a CSR kernel. They got several performance bottlenecks. Then, they designed a new sparse matrix format called ELLPACK Sparse Block (ESB) that is tuned for KNC. The new format adds several features to the ELLPACK format, for example: 1. finite-window sorting in order to improve the SIMD efficiency of the kernel. 2. A bit array to encode nonzero locations to reduce the bandwidth requirement. 3. Column blocking to improve memory access locality. In addition to that, in order to deal with the load balancing problem, they proposed 3 load balancers for the SpMV kernel on KNC. The proposed kernel is 1.85x faster than other kernels that use the CSR format. Speaking of architecture, the SpMV kernel on KNC is 3.52x faster than on dual-socket Intel Xeon Processor E5-2680 and is 1.32x faster than on NVIDIA Tesla K20X.

J. W. Choi and et al. [31] proposed a performance model-driven framework for auto-tuning the SpMV kernel (more specifically, BELLPACK and BCSR kernels) on the GPU. They started out by implementing the blocked CSR (BCSR) format. The BCSR format led them to developing a new sparse matrix storage format called the blocked ELLPACK (BELLPACK) format. Using BELLPACK, they obtained up to 29.0 GFLOP/s in single-precision and 15.7 GFLOP/s in double-precision. Additionally, they worked on tuning the BELLPACK format. They developing a performance model-driven framework for auto-tuning the SpMV kernel for the BELLPACK format using some tuning parameters. This proposed framework is applicable only to matrices with small dense block sub-structures and to BELLPACK and BCSR kernels only.

P. Guo and et al. [32] proposed an auto-tuning framework that can calculate and select CUDA parameters for SpMV kernels in order to attain the optimal performance on the GPU. The proposed framework was tested on GeForce 9500 GTX and GeForce GTX 295 NVIDIA GPUs. The CUDA parameters that they used were the number of threads, the block size, and the warp size. They researched how these parameters can affect the performance of SpMV kernels. The auto-tuning

framework has 237% and 33% performance improvements on average for GeForce 9500 GTX and GeForce GTX 295 GPUs compared to SpMV kernels without the auto-tuning framework.

D. Grewe and et al. [33] proposed a representation language that is used for sparse matrix formats. The input will be a description of the sparse matrix format, then the compiler will automatically generate code for the SpMV kernel on the GPU. They used six sparse matrix formats: CSR, DIA, ELL, HYB, sliced ELLPACK and blocked ELLPACK. The automatically-generated code provides the same performance and occasionally better performance compared to the current hand-written kernel code for the SpMV operation. In addition to that, the automatically-generated SpMV code is automatically tuned in order to improve the performance of the SpMV operation. Besides, the description of the sparse matrix format can be used in order to automatically generate vectorized code.

A. Ashari and et al. [34] proposed a new algorithm for the SpMV kernel on the GPU. This algorithm is called Adaptive CSR (ACSR) and it's well-suited for graph processing applications and it uses the CSR format. It reduces thread divergence by putting rows that almost have the same number of non-zero elements into groups called bins. It overcomes the CSR problem when the deviation in the number of non-zero elements per row is high. This algorithm's preprocessing is limited to scanning the lengths of rows. The ACSR algorithm outperforms implementations from the NVIDIA CUSP and cuSPARSE libraries using a set of sparse matrices representing power-law graphs. They demonstrates the use of the ACSR algorithm for the analysis of dynamic graphs and showed great improvements over existing approaches. They demonstrated the benefits of the ACSR algorithm using several data analytics applications that utilize SpMV frequently.

X. Yang and et al. [35] developed optimizations for the SpMV kernel that runs on the GPU and they studied the effects of these optimizations with regard to graph mining. They tiled the matrix using texture cache. Their approach of using tiling uses the texture cache very efficiently. These optimizations consider the architecture of the GPU as well as the features of graph mining applications. Their work attained noticeable performance improvement on graphs by parameters tuning. They developed a performance model in order to automatically tune the developed tile-

composite kernel. They extended the use of their optimizations for the sake of dealing with web graphs on an MPI-based cluster.

The sparseness pattern of sparse matrices differs from one sparse matrix to another. Therefore, a storage format which is ideal to store one sparse matrix might not be ideal for another sparse matrix. K. K. Matam and et al. [36] proposed a technique in order to understand the pattern and nature of sparse matrices or what's called, preprocessing the sparse matrix and subsequently select the appropriate storage format for the sparse matrix in order for this storage format to be used in the SpMV process. They combined both the CSR format and the ELLPACK format into one storage format. If the number of non-zero elements in a row is more than a predefined threshold, the whole entire row will be stored in the CSR format. If the number of non-zero elements in a row is less than the same predefined threshold, the whole entire row will be stored in the ELLPACK format. After their implemented SpMV kernel used their storage format, the performance of the SpMV operation on NVidia Tesla GPU (C1060) was boosted up to 80% in some cases and 25% on average. They applied the proposed SpMV kernel to the conjugate gradient method and they obtained an average performance improvement of 20% when they applied the proposed kernel to the conjugate gradient method than applying the SpMV kernel using the HYB format to the conjugate gradient method.

The ELLPACK-R format is a common format for storing sparse matrices. A problem with the ELLPACK-R format is that when the maximum number of non-zero elements per row doesn't differ significantly from the average, the thread will suffer from load imbalance. W. Cao and et al. [37] proposed a new sparse matrix storage format called ELLPACK-RP. This format is a combination of two formats: the ELLPACL-R format and the JAD format [38]. They also implemented the SpMV kernel on the GPU using the proposed format. Firstly, they store the matrix in the ALLPACK-R format then they perform row permutation by sorting the rows of the matrix in a descending order based on the number of non-zero elements in each row. The RL array has to be permuted as well. The array (Permu) is used to keep track of the original positions of the rows. The proposed format obtains a better average performance than the ELLPACK-R format due to

the fact that the proposed format decreases the degree of load imbalance of the ELLPACK-R kernel.

X. Feng and et al. [39] proposed a new sparse matrix storage format called CSR with Segmented Interleave Combination (SIC). The proposed format combines certain amounts of the CSR rows to form a new SIC row. Segmented processing is considered in the proposed format. They also developed an automatic SIC-based kernel that applies to any matrix. The CSR storage format allows a variable number of nonzero entries per row, which is considered an advantage, nevertheless, this introduces thread divergence especially when we deal with a matrix with a high variable number of nonzero elements per row, in this case, many threads within the warp will remain idle while the thread that works on the longest row is still working. To lessen thread divergence, they assigned half of the warp to work on each row. To reduce imbalance between warps, Compressed Sparse Row is reordered and segmented. The proposed storage format outperforms the CSR vector kernel. Besides, it gives a comparable performance to the Hybrid format as well.

P. Guo and et al. [40] proposed a performance modeling and optimization analysis tool in order to predict as well as optimize the performance of the SpMV kernel on the GPU. Their model is platform-independent, since it doesn't depend on the programming language used or on the architecture of the GPU. They used analytical modeling in order to predict the execution times of the CSR, ELL, COO, and HYB SpMV kernels. They used NVIDIA Tesla C2050 GPU for their experiments. They reported that for 77 out of 82 cases, the differences between the measured and the predicted execution times in terms of performance were less than 9 percent, for the remaining cases, the differences were between 9 and 10 percent. For the CSR, ELL, COO, and HYB SpMV kernels, the average differences were 6.3, 4.4, 2.2, and 4.7 percent, respectively. They developed an auto-selection algorithm in order to automatically select the best solution in terms of storage format and execution time for the sparse matrix.

A. Ashari and et al. [41] developed a new sparse matrix storage format called blocked row-column (BRC). The proposed format improves load balancing by partitioning rows into blocks with the same number of non-zero elements. The proposed format reduces thread divergence by

reordering and grouping the rows of the sparse matrix that almost have the same number of non-zero entries in the same warp. They developed an auto-tuning technique in order to optimize the performance of the proposed format. The proposed format is adaptive to the matrix characteristics. Based on the sparsity features of the matrix, the size of the block is selected. The proposed format outperforms NVIDIA CUSP and cuSPARSE libraries, JDS, and other formats as well. They researched the two main formats that comprise the hybrid format (COO and ELLPACK). They proved that the proposed format can improve the COO format part that slows down the hybrid format.

F. Vázquez and et al. [42] focused on the ELLR-T kernel [22]. They proposed a model in order to auto-tune the ELLR-T kernel that is used to perform the SpMV operation on the GPU. The performance of the ELLR-T kernel mainly depends on 2 parameters: the number of threads per row and the block size (number of threads per block). The proposed model was developed on GeForce GTX285 and Tesla C2050 GPUs and was applied to a large set of test sparse matrices .When the proposed model is considered, the ELLR-T kernel attains 92% of the optimal performance on GeForce GTX285 and 94% of the optimal performance on Tesla C2050. Applying the proposed model to the ELLR-T kernel will have a superior performance compared to the other approaches that are developed so far. The average performance when applying the proposed model to the ELLR-T kernel will be close to optimum.

W. Liu and et al. [43] proposed a new sparse matrix storage format called CSR5 (Compressed Sparse Row 5). The proposed format is insensitive to the sparsity structure of the input matrix. The proposed format provides high performance for the SpMV operation on the CPU, the GPU, and Xeon Phi. Converting the CSR format to the proposed format has low overhead and also very fast. They implemented the proposed format on the CPU, NVidia GPU, AMD GPU, and Intel Xeon Phi. They evaluate the CSR5 format in both isolated SpMV tests and also iteration-based scenarios. The SpMV algorithm that is based on the proposed format was compared to other algorithms that are based on other storage formats using regular and irregular matrices. The SpMV based on the proposed storage format achieves better or comparable performance to the existing storage formats.

M. Kreutzer and et al. [44] proposed a new matrix storage format called padded JDS (pJDS). The proposed format can be used for the SpMV kernel on the general purpose GPU (GPGPU). The proposed format might allow us to save a significant amount of memory space. The performance of the proposed format is comparable to or better than the ELLPACK-R format. They came up with a condition for the average number of non-zero elements in each matrix row that guarantees a beneficial performance benefit of GPGPU-based spMVM in comparison to standard server nodes. The proposed format takes the overall spMVM memory footprint down on the GPGPU by up to 70%. The proposed format attains 91% to 130% of the ELLPACK-R format performance. They extended previous work on distributed-memory parallel spMVM in order to present a scalable hybrid MPI-GPGPU code.

J. Godwin and et al. [45] proposed a new sparse matrix storage format that takes advantage of the diagonal structure of the sparse matrix for stencil operations on structured grids. The storage format is optimized for block-diagonal sparse matrices that come from structured grid computations with multiple degrees of freedom. They also developed the SpMV kernel of the proposed format on the GPU using CUDA. They implemented their kernel on NIVIDIA GTX 280, Quadro Plex S2200 S4, Tesla C2050, and GTX580. They dealt with sparse matrices that come from structured grid problems with high degrees of freedom at each grid node. They focus on sparse matrices that have a block structure. Thus, other storage formats such as the CSR format or the DIA format won't be beneficial and efficient for this type of sparse matrices. They optimized for the case of higher degrees of freedom, where other formats (i.e., DIA) are forced to include many zero entries in the matrix. The performance of the developed kernel that is based on the proposed storage format exceeds the performance of other kernels that are based on other formats (CSR and DIA) for more than one degree of freedom.

W. Xu and et al. [46] focused on tuning the performance of the SpMV operation on the GPU. In this paper, they proposed a cache blocking technique (storage format) in order to improve the performance of the SpMV kernel on the GPU. In the proposed format, the sparse matrix is divided into sub-blocks and each sub-block is stored in the CSR format. Each element of the vector will be stored in the cache, so it can be reused by different blocks, thus, reduced the global memory

access time. The developed kernel of the proposed format was implemented on GeForce GTX 480 that is a Fermi GPU. It consists of a 16K/48K configurable cache on each SM. The developed kernel is 5x faster than the unblocked CSR kernel.

Z. Wang and et al. [47] were working on optimizing the SpMV operation/kernel on NIVIDIA GPUs. They tried to optimize the sparse matrix vector multiplication kernel on NIVIDIA GPUs using the CUDA parallel computing platform. They tried to research the basic challenges with regard to performance in SpMV kernels. They focused on 3 main optimization factors that considered the application and the architectural characteristics: optimizing the CSR sparse matrix storage format, optimizing thread mapping, and avoiding divergence judgment. They implemented the developed optimizations on GeForce 9600 GTX GPU. They compared their approach to NVIDIA's SpMV library and NVIDIA's CUDDP library. Optimizing the SpMV kernel on NIVIDIA GPUs gained noticeable higher performance over other SpMV implementations.

Accessing the memory in SpMV applications happens very frequently, that is why improving the performance of these applications is a challenging task. B. Neelima and et al. [48] proposed a model that can be used in order to detect and choose the best storage format for a sparse matrix. The predicted format by the model is the best high performing format when considering the pre-processing time, CPU to GPU communication time and SpMV computation time on the GPU. The proposed model can predict the best format for any sparse matrix based on the input data that is available. The overhead added to the application is very small. The execution time difference varies from 7.5% to 159.9% when a random format is compared against a selected optimal format, for a few benchmark input matrices.

P. Guo and et al. [49] proposed a performance model that can be used to predict the execution time of the SpMV kernel. They have developed a framework that can be used to partition a sparse matrix into multiple partitions and then store each partition in the appropriate storage format based on different storage characteristics. They integrated the partitioning framework with their previous auto-tuning framework in order to adjust some CUDA parameters so as to improve the performance of the SpMV kernels on the GPU. The partitioning approach was evaluated by using 14 matrices on NVIDIA’s GeForce GTX 295. The developed approach has an average

performance improvement of 222%, 197%, and 33% for the CSR vector kernel, ELL kernel, and HYB kernel. They plan to improve their currently-developed performance model in order to predict matrix partitioning more accurately as well as improving and extending the current framework by including more SpMV kernels.

S. Yan and et al. [50] developed a storage format for storing sparse matrices on the GPU in addition to updating the SpMV kernel that is based on the proposed format on the GPU using CUDA. The proposed format is called blocked compressed common coordinate (BCCOO). They extended the COO with blocking. They partitioned the sparse matrix vertically prior to using the proposed format in order to improve the locality for accesses to the vector. They also used a bit flag array instead of using the row index array. They developed a powerful matrix-based segmented sum/scan for the SpMV operation. They evaluated the proposed approach on 20 sparse matrices. The proposed approach outperforms other modern SpMV algorithms. It outperforms the vendor tuned CUSPARSE by up to 150% and 42% on average on GTX480 GPUs, by up to 229% and 65% on average on GTX680 GPUs.

The Compressed Sparse Blocks (CSB) format [12] [13] [14] is used for storing sparse matrices. The CSB format partitions the n X n matrix into n2/z2 equal-sized z X z square blocks using a block size parameter z. The CSB format consists of the following: Value vector of length nnz. It stored all the non-zero elements of the sparse matrix. row_idx and col_idx vectors track the row index and the column index of each non-zero entry inside the Value vector with regard to the block, not the whole entire matrix. So row_idx and col_idx range from 0 to z – 1. Block_ptr vector stores the index of the first non-zero entry of each block inside the Value vector.

CHAPTER 6

THE PROPOSED MODEL

Before we started working on the proposed model, we have started out by implementing the most common and knows formats for storing sparse matrices on the CPU as well as on the GPU. These formats that have been developed are:

- The Compressed Sparse Row (CSR) or Compressed Row Storage (CRS) Format.
- The ELLPACK (ELL) Format.
- The ELLPACK-R (ELL-R) Format.
- The Sliced ELLPACK (Sliced ELL) Format.
- The Sliced ELLPACK-R (Sliced ELL-R) Format.

Also, we have developed the SpMV kernels for the previously mentioned formats on the CPU and on the GPU as well.

The CI matrix is a sparse matrix that has a greatly-varying dimension length. It can be up to 10^7 by 10^7 or even more. Generally speaking, the CI matrix includes two main regions, namely, the Reference region and the Expansion Space region. The Reference region occupies almost 10% of the whole CI matrix with sparsity ranging from 70% to 80%. The Reference region starts out from the left side of the CI matrix. The Expansion Space region occupies the remaining space of the CI matrix with extremely high sparsity, which is around 98% to 99%.

The Proposed Storage Format:

The proposed format is a combination of two formats: the ELLPACK format and the CSR format. Each single row in the matrix is divided up into two sections based on the value of BOUNDARY. For each single row, columns with column indices ranging from 0 to (BOUNDARY – 1) will be stored in the ELLPACK format and the rest of the row will be stored in the CSR format. The proposed format is illustrated in Figure 4.

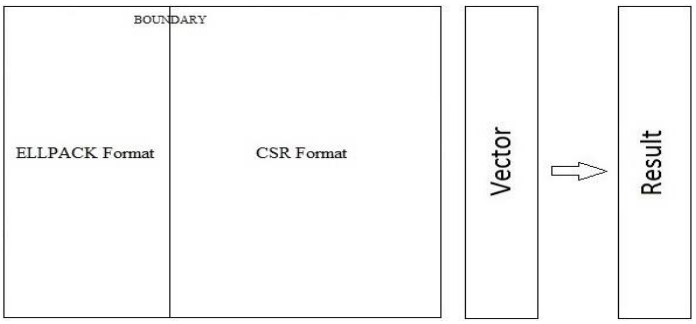

Fig. 4. The Proposed Format

The CSR format is very efficient with regard to using memory (storage space) since it does not require zero-padding like the ELLPACK format. Also, the CSR format provides high performance for SpMV operations on CPUs that have multiple cores [2]. On the other hand, the CSR format provides lower performance than the ELLPACK format on GPUs when it comes to SpMV because of the lack of coalesced access to global memory on the GPU. The ELLPACK format provides higher performance than the CSR format on the GPU [2].

The Reference region of the CI matrix represents most of the electronic structure and performance is an important factor in this part, consequently we are going to store the Reference region in the ELLPACK format. The Expansion Space region that occupies the rest of the CI matrix has a noticeable high sparsity, so storage space is crucial and critical in this part. The CSR format is very powerful regarding storage space, therefore the Expansion Space region will be stored in the CSR format. If we tried to store the Expansion Space region in the ELLPACK format, we will end up with a huge amount of zero-padding especially if one or more rows in the Expansion Space region was not/were not as sparse as the other rows in the same region.

As we saw in the previous figure (Fig. 2), the value of BOUNDARY marks the end of the Reference region which is stored in the ELLPACK format and the beginning of the Expansion Space region which is stored in the CSR format. Algorithm 6 describes the proposed format that we have created.

Algorithm 6 is about creating the proposed format.

Input:

- **Mat:** The original matrix.
- **AllNonZerosCount:** The count of all the non-zero entries per row.

Output:

- **NonZerosEntries Matrix:** All the non-zero entries that are stored in the ELLPACK format.
- **ELLPACK_Column Matrix:** The column index of each non-zero entry that is stored in the ELLPACK format.
- **Value Vector:** All the non-zero entries that are stored in the CSR format.
- **CSR_Column Vector:** The column index of each non-zero entry that is stored in the CSR format.
- **StartArr:** The index of the first non-zero entry of each row in the "value" array.
- **EndArr:** The (index + 1) of the last non-zero entry of each row in the "Value" array.

```
CreateFormat(Mat, NonZerosEntries, ELLPACK_Column, Value, CSR_Column,
AllNonZerosCount, StartArr, EndArr)
for i ← 0 to ROWS – 1 do
  c = 0
  GotIt = 0
  for j ← 0 to COLS – 1 do
    if Mat[i][j] != 0.00 then
      if c < BOUNDARY
        NonZerosEntries[r][c] = Mat[i][j]
        ELLPACK_Column[r][c]  = j
        c++
      else
        Value[Index] = Mat[i][j]
        CSR_Column[Index] = j
        if GotIt = 0
          GotIt = 1
          Start = Index
        Index++
  r++
  if AllNonZerosCount[i] > BOUNDARY
    End = Index
    StartArr[i] = Start
    EndArr[i] = End
```

Algorithm 6. The proposed format.

The proposed model can calculate the amount of allocated memory that the proposed format uses. The space allocated to the proposed storage format can be calculated through the following equation:

Storage Space = (3 * ROWS * sizeof(unsigned int)) + (ROWS * BOUNDARY * sizeof(double)) + ROWS * BOUNDARY * sizeof(int)) + (CSRNonZeros * sizeof(double)) + (CSRNonZeros * sizeof(unsigned int)).

Where:

ROWS: The number of rows in the matrix.

BOUNDARY: A divider between the Reference region (ELLPACK format) and the Expansion Space region (CSR format). It marks the end of the Reference region and the start of the Expansion Space region.

CSRNonZeros: The number of non-zero entries in the Expansion Space region.

The space allocated to the CSR storage format can be calculated through the following equation:

Storage Space = (AllNonZeros * sizeof(double)) + (AllNonZeros * sizeof(unsigned int)) + ((ROWS + 1) * sizeof(unsigned int)).

Where:

AllNonZeros: The number of non-zero entries in the matrix.

ROWS: The number of rows in the matrix.

The space allocated to the ELLPACK storage format can be calculated through the following equation:

Storage Space = (ROWS * MaxNonZeros * sizeof(double)) + (ROWS * MaxNonZeros * sizeof(int)).

Where:

ROWS: The number of rows in the matrix.

MaxNonZeros: The length of the longest non-zero entry row in the matrix.

The Sliced ELLPACK Format was created in order to deal with the redundancy problem that exists in the ELLPACK Format. In this format, the matrix will be divided into submatrices (sometimes called slices) and then each submatrix will be stored in the ELLPACK format. The storage spaced that is allocated to the Sliced ELLPACK format will be the sum of the storage spaces that are allocated to the submatrices that make up the Sliced ELLPACK format. The sliced ELLPACK format takes less storage space than the ELLPACK format.

The Developed SpMV Kernel:

The proposed storage format is used by our developed Warp-Based SpMV Kernel. The proposed SpMV kernel is a single SpMV vector kernel that assigns a warp to each single row in the Reference region and assigns another warp to each single row in the Expansion Space region. Although scalar kernels (one thread per row) are relatively straightforward and provide reasonable performance, there is a big disadvantage that comes with them, when a thread accesses the elements of the storage format (by the storage format, we mean the vector that stores the values and the vector that stores the column indices), it accesses them in a sequential way. Every thread does not access these vectors simultaneously although they are stored in a contiguous fashion in the format, the matter that could have a negative effect on performance.

We overcame this problem when we developed the proposed SpMV kernel. The proposed SpMV kernel is a vector kernel that uses the warp approach, each row in each of the two regions of the storage format (Reference and Expansion Space regions) is accessed by a warp (32 threads), so in total, we have 2 warps that access each matrix row. The vector kernel eliminates the problem that is inherent in the scalar kernel (the threads' sequential access to the values in the storage format). The order that Warps access the memory is difficult to determine and also does not affect performance. Also, all warps execute independently in vector kernels, therefore thread divergence is less pronounced.

Each warp in the proposed SpMV kernel mandates coordination among the 32 threads that are within it, we achieved this using the atomicAdd() function. The atomicAdd() function provides a level of synchronization to the kernel, it reads a memory location and updates it and then stores the result back into the same location. Every thread has to wait on accessing that memory location until the atomicAdd() function is finished.

The proposed format uses a single SpMV kernel although it's a combination of two different formats (The ELLPACK format and the CSR format). Algorithm 7 describes the SpMV kernel for the proposed format.

Algorithm 7 is the SpMV kernel for the proposed format.

Input:

- **NonZerosEntries Matrix:** All the non-zero entries that are stored in the ELLPACK format.
- **ELLPACK_Column Matrix:** The column index of each non-zero entry that is stored in the ELLPACK format.
- **Value Vector:** All the non-zero entries that are stored in the CSR format.
- **CSR_Column Vector:** The column index of each non-zero entry that is stored in the CSR format.
- **StartArr:** The index of the first non-zero entry of each row in the "value" array.
- **EndArr:** The (index + 1) of the last non-zero entry of each row in the "Value" array.
- **AllNonZerosCount:** The count of all the non-zero entries per row.
- **Vector:** The vector that the matrix will be multiplied by.

Output:

- **Result:** The result of the SpMV process.

```
SpMV_Hybrid_ELLPACKandCSR(NonZerosEntries, ELLPACK_Column, Value, CSR_Column,
StartArr, EndArr, AllNonZerosCount, Vector, Result)
Thread_id ← (blockDim.x * blockIdx.x) + threadIdx.x
Warp_id ← Thread_id / 32
Th_Wa_Id ← Thread_id % 32
define ShMem1[32] // Shared Memory
define ShMem2[32] // Shared Memory
initialize ShMem1
initialize ShMem2
if Warp_id < ROWS then
  ShMem1[threadIdx.x] = 0.00
  for r2 ← 0 + Th_Wa_Id  to BOUNDARY – 1 STEP=32 do
    ShMem1[threadIdx.x] ← ShMem1[threadIdx.x] + (NonZerosEntries[Warp_id][r2] *
Vector[ELLPACK_Column[Warp_id][r2]])

  //Warp-level reduction for the ELLPACK format:
if Th_Wa_Id  < 16
    ShMem1[threadIdx.x] ←(SynchronousAdd) ShMem1[threadIdx.x] + ShMem1[threadIdx.x + 16]
if Th_Wa_Id  < 8
  ShMem1[threadIdx.x] ←(SynchronousAdd) ShMem1[threadIdx.x] + ShMem1[threadIdx.x + 8]
if Th_Wa_Id  < 4
  ShMem1[threadIdx.x] ←(SynchronousAdd) ShMem1[threadIdx.x] + ShMem1[threadIdx.x + 4]
if Th_Wa_Id  < 2
  ShMem1[threadIdx.x] ←(SynchronousAdd) ShMem1[threadIdx.x] + ShMem1[threadIdx.x + 2]
if Th_Wa_Id  < 1
  ShMem1[threadIdx.x] ←(SynchronousAdd) ShMem1[threadIdx.x] + ShMem1[threadIdx.x + 1]

if AllNonZerosCount[Warp_id] > BOUNDARY then
  Start ← StartArr[Warp_id]
  End ← EndArr[Warp_id]
  ShMem2[threadIdx.x] = 0.00
  for j ← Start + Th_Wa_Id  to End – 1 STEP=32 do
    ShMem2[threadIdx.x] ← ShMem2[threadIdx.x] + (Value[j] * Vector[CSR_Column[j]])

  //Warp-level reduction for the CSR format:
  if Th_Wa_Id  < 16
    ShMem2[threadIdx.x] ←(SynchronousAdd) ShMem2[threadIdx.x] + ShMem2[threadIdx.x + 16]
  if Th_Wa_Id  < 8
    ShMem2[threadIdx.x] ←(SynchronousAdd) ShMem2[threadIdx.x] + ShMem2[threadIdx.x + 8]
  if Th_Wa_Id  < 4
    ShMem2[threadIdx.x] ←(SynchronousAdd) ShMem2[threadIdx.x] + ShMem2[threadIdx.x + 4]
  if Th_Wa_Id  < 2
    ShMem2[threadIdx.x] ←(SynchronousAdd) ShMem2[threadIdx.x] + ShMem2[threadIdx.x + 2]
  if Th_Wa_Id  < 1
    ShMem2[threadIdx.x] ←(SynchronousAdd) ShMem2[threadIdx.x] + ShMem2[threadIdx.x + 1]
  if Th_Wa_Id  = 0
    // Writing the results
    Result[Warp_id] ← Result[Warp_id] + ShMem1[threadIdx.x] + ShMem2[threadIdx.x]
```

Algorithm 7. The SpMV kernel for the proposed format.

Thread_id is the index of each thread within the whole entire grid, Warp_id is the index of each warp within the grid, and Th_Wa_Id is the index of each thread inside the warp.

The overall memory is efficiently used throughout the use of shared memory. The shared memory (so much faster than global memory) is useful if you access the data transferred to it more than once (which happens in the proposed model), using good access patterns, to have it help. If you access the data once, then the shared memory is not going to be useful. We computed the running sum for each thread in the Reference region by copying intermediate results to the shared memory (SharedMem1). We did the same thing with the Expansion Space region by copying intermediate results to the shared memory (SharedMem2). For the Reference region and the Expansion Space region, we had to perform parallel reduction in SharedMem1 and SharedMem2. The global memory is not accessed by every thread within the warp, instead we let the first thread in the warp (Th_Wa_Id = 0) access the global memory and write the results back to it by copying them from the shared memory (SharedMem1 and SharedMem2) to the global memory.

The proposed model generates some information about the CI matrix itself, i.e., the number of non-zero elements in the whole entire matrix, the number of non-zero elements in the Reference region (ELLPACK format), the number of non-zero elements in the Expansion Space region (CSR format), the number of non-zero elements in each row of the matrix, the number of non-zero elements in each row of the Reference region, the number of non-zero elements in each row of the Expansion Space region. It also outputs the length of the longest non-zero entry row in the whole entire matrix and its order as well. In addition to that, the proposed model calculates the amount of allocated memory that is used by the proposed format and the execution time that is consumed by the SpMV kernel. In addition to the results of the SpMV operation, the proposed model can produce the components of each of the storage formats (The ELLPACK format and the CSR format) that compose the hybrid format, if the user wanted to.

CUDA offers a relatively light-weight alternative to CPU timers via the CUDA event API. The CUDA event API includes calls to create and destroy events, record events, and compute the elapsed time in milliseconds between two recorded events. CUDA events are of type cudaEvent_t and are created and destroyed with cudaEventCreate() and cudaEventDestroy(). In the following

example, cudaEventRecord() places the start and stop events into the default stream, stream 0. The device will record a time stamp for the event when it reaches that event in the stream. The function cudaEventSynchronize() blocks CPU execution until the specified event is recorded. The cudaEventElapsedTime() function returns in the first argument the number of milliseconds time elapsed between the recording of start and stop. This value has a resolution of approximately one half microsecond.

Example:

```
cudaEvent_t Start, End;

float ExecutionTime = 0.00f;

cudaEventCreate(&Start);

cudaEventCreate(&End);

cudaEventRecord(Start);

(void) SpMV_Hybrid_ELLPACKandCSR<<<GridDim, BlockDim>>>(d_NonZerosEntries, d_ELLPACK_Column, d_Value, d_CSR_Column, d_StartArr, d_EndArr, d_AllNonZerosCount, d_Vector, d_Result);

cudaEventRecord(End);

cudaEventSynchronize(End);

cudaEventElapsedTime(&ExecutionTime, Start, End);
```

The Sliced ELLPACK Format was created in order to deal with the redundancy problem that exists in the ELLPACK Format. In this format, the matrix will be divided into submatrices (sometimes called slices) and then each submatrix will be stored in the ELLPACK format. The storage spaced that is allocated to the Sliced ELLPACK format will be the sum of the storage spaces that are allocated to the submatrices that make up the Sliced ELLPACK format. The sliced ELLPACK format takes less storage space than the ELLPACK format.

There is a variation of the proposed format that we have already developed. This variation is similar to the proposed format except the fact that instead of using the ELLPACK format, we used the Sliced ELLPACK format. This variation is a combination of two formats: the Sliced ELLPACK format and the CSR format [3]. The matrix has to be divided first into submatrices (slices). Each single row in slice_n is divided into two sections based on the value of BOUNDARY_n. For each single row in slice_n, columns with column indices ranging from 0 to (BOUNDARY_n – 1) will be stored in the Sliced ELLPACK format and the rest of the row will be stored in the CSR format. This format is illustrated in Figure 5.

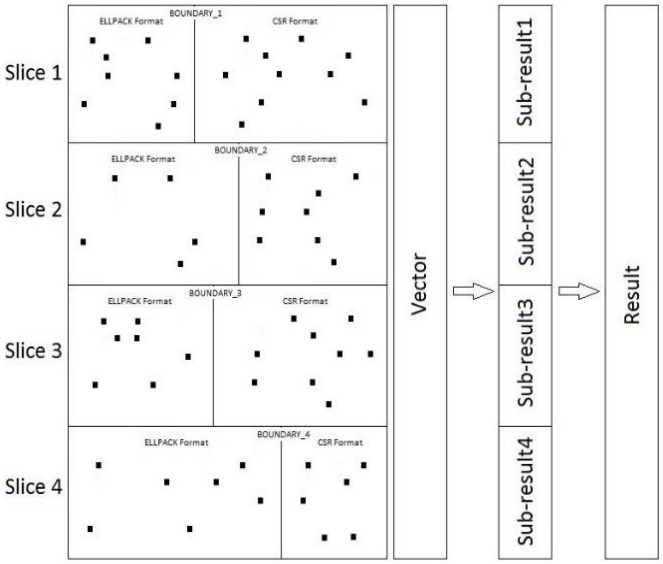

Fig. 5. A Variation of the Proposed Model

Algorithm 8 describes the SpMV kernel for a variation of the proposed format that we just discussed.

Algorithm 8 is the SpMV kernel for a variation of the proposed format.

Input:

- **NonZerosEntries Matrix:** All the non-zero entries that are stored in the ELLPACK format in the slice.
- **ELLPACK_Column Matrix:** The column index of each non-zero entry that is stored in the ELLPACK format in the slice.
- **Value Vector:** All the non-zero entries that are stored in the CSR format in the slice.
- **CSR_Column Vector:** The column index of each non-zero entry that is stored in the CSR format in the slice.
- **StartArr:** The index of the first non-zero entry of each row in the "value" array in the slice.
- **EndArr:** The (index + 1) of the last non-zero entry of each row in the "Value" array in the slice.
- **AllNonZerosCount:** The count of all the non-zero entries per row in the slice.
- **Vector:** The vector that the matrix will be multiplied by.
- **Rows:** The number of rows in the slice.
- **Boundary:** A divider between the ELLPACK format and the CSR format in the slice. It marks the end of the ELLPACK format and the start of the CSR format in the slice.

Output:

- **Result:** The result of the SpMV process for the slice.

```
SpMV_Hybrid_SlicedELLPACKandCSR(NonZerosEntries, ELLPACK_Column, Value,
CSR_Column, StartArr, EndArr, AllNonZerosCount, Vector, Result, Rows, Boundary)
for r1 ← 0 to Rows – 1 do
  for r2 ← 0 to Boundary – 1 do
    if ELLPACK_Column [r1][r2] = -1 then
    exit for loop
    Temp ← Temp + (NonZerosEntries[r1][r2] * Vector[ELLPACK_Column[r1][r2]])
  if AllNonZerosCount[r1] > Boundary then
    Start ← StartArr[r1]
    End   ← EndArr[r1]
    for j ← Start to End – 1 do
      Temp ← Temp + (Value[j] * Vector[CSR_Column[j]])
  Result[r1] ← Temp
  Temp       ← 0.00
```

Algorithm 8. The SpMV kernel for a variation of the proposed format.

CHAPTER 7

THE EXPERIMENTAL RESULTS

System Configuration:

The proposed format and the developed kernel were both developed on Hodor supercomputer. Hodor has 33 compute nodes in total. 1 compute node is the head node. 16 out of the 33 compute nodes have NIVIDIA K20m GPUs. The remaining 16 compute nodes have Intel® Xeon Phi™ co-processor. The 16 compute nodes that have NIVIDIA K20m GPUs can be used to process CUDA code. Each compute node has the following configurations:

- 2x 3.3 GHZ Sandy Bridge CPU. Each CPU has 4 cores.
- NIVIDIA K20m GPU card or Intel® Xeon Phi™ co-processor card.
- 64 GB of memory.
- 150 GB RAID 1 7.2 K RPM storage hard drives.
- Linux Red Hat operating system RHEL 7.0.

Hodor supercomputer has an external 10 Gbps Ethernet communications network. In addition to that, it has an internal 56 Gbps Internal FDR InfiniBand network between the compute nodes.

CUDA can be used in order to list the GPU properties. Code 3 and Code 4 in Appendix 2 list some of the GPU properties on Hodor. The output is illustrated in table 3 below:

Name	Tesla K20m
Major revision number	3
Minor revision number	5
Maximum memory pitch	2147483647
Clock rate	705500
Texture alignment	512
Concurrent copy and execution	Yes
Kernel execution timeout enabled	No
Number of multiprocessors	13
The maximum number of threads per multiprocessor	2048
The maximum number of threads per block	1024
The maximum sizes of each dimension of a block (x, y, z)	1024, 1024, 64
The maximum sizes of each dimension of a grid (x, y, z)	2147483647, 65535, 65535
The total number of registers available per block	65536
The total amount of shared memory per block (Bytes)	49152
The total amount of constant memory (Bytes)	65536
The total amount of global memory (Bytes), (Gigabytes)	4972937216, 4.631409
Warp size (Threads)	32

TABLE 3: GPU Properties

The following figure (figure 6) is an illustration of the Hodor supercomputer. It shows the total 33 nodes (1 head nodes and 32 compute nodes) in addition to the internal as well as the external networks:

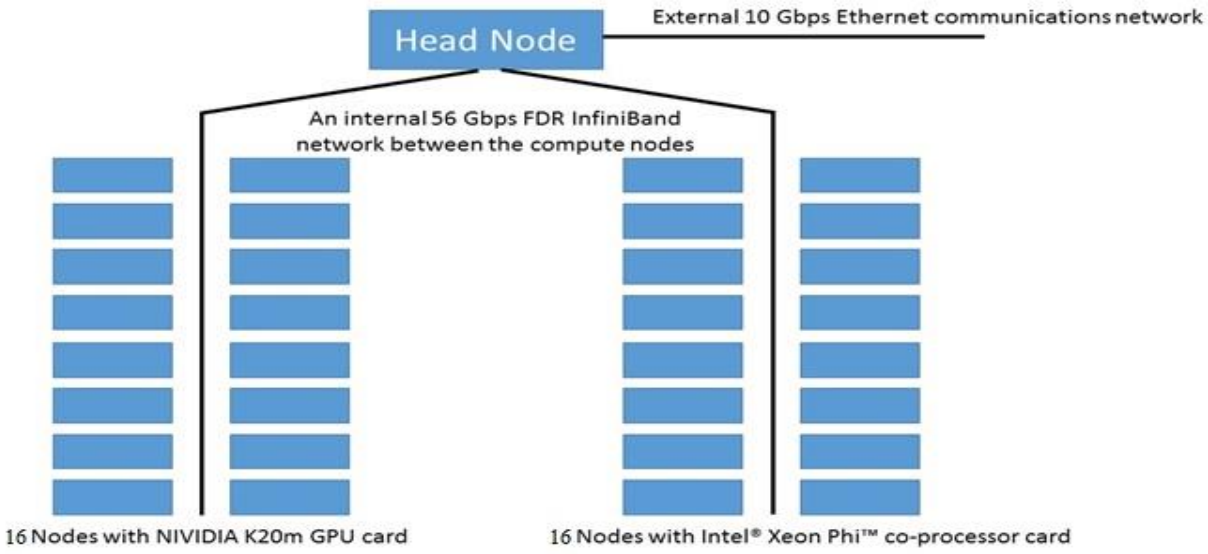

Fig. 6. Hodor Supercomputer

The Results:

We have tested the proposed model with some simple and relatively large sparse matrices. We have compared the proposed format to the other common format using different-sized sparse matrices. Besides, we have compared the proposed kernel to the cuSPARSE library and the CSR5 (Compressed Sparse Row 5) format. The comparison was based on two main factors: the amount of used memory and performance.

The output of the proposed model is very inclusive and comprehensive. It gives a lot of details about the format that is being used to store the sparse matrix on the GPU as well as the results of the SpMV operation. The information presented in the output is very intensive and elaborate.

The sparse matrices that we used for testing are generated from one of the modules that is a part of the UNDMOL package. The dimensions (the number of rows and the number of columns) of the CI sparse matrix are the primary parameters to the program for the sake of generating CI sparse matrices. The program will ask for the sparsity of the reference region and the sparsity of the expansion space region to fully generate CI sparse matrices. The CI sparse matrices that we used for testing were generated with different sparsities that led to different number of non-zero elements.

We started out by using 10 CI sparse matrices for testing. Each sparse matrix is a 32768 by 32768 matrix (32768 rows and 32768 columns). Each single matrix out of the 10 matrices was stored in different storage formats, namely the CSR format, the ELLPACK format, the Sliced ELLPACK format, and finally, the proposed format. We compared the proposed format to the other mentioned formats in terms of memory usage. We also compared the execution time of the proposed SpMV kernel to the execution times of the cuSPARSE library and the CSR5 (Compressed Sparse Row 5) format.

Table 4 demonstrates some general information about the 10 testing sparse matrices that will be stored in the proposed storage format. This information is about the number of non-zero entries in the Reference region, the number of non-zero entries in the Expansion Space region, and the total number of non-zero entries in the whole entire sparse matrix.

The last column of table 4 identifies the length of the longest non-zero entry row, in another way, the length of the row that has the largest number of non-zero elements in the whole entire sparse matrix. It also identifies the Row_Number of that row. Figure 7 shows the total number of non-zero entries in each of the 10 CI sparse matrices in a graphical way.

Rows:

Row n: An individual CI sparse matrix.

Cols:

Ref. Non-Zeros: The number of non-zero elements in the Reference Region.

Exp. Space Non-Zeros: The number of non-zero elements in the Expansion Space Region.

Total Non-Zeros: The total number of non-zero elements in the whole entire sparse matrix.

Length of the Longest Non-Zero Row - Row No.: The length of the longest non-zero entry row – The row number.

	Ref. Non-Zeros	Exp. Space Non-Zeros	Total Non-Zeros	Length of the Longest Non-Zero Row - Row No.
Matrix 1	21,463,040	9,678,327	31,141,367	1074 - 5314
Matrix 2	21,463,040	9,673,031	31,136,071	1069 - 6144
Matrix 3	21,463,040	9,666,827	31,129,867	1078 - 22358
Matrix 4	21,463,040	9,670,529	31,133,569	1068 - 21549
Matrix 5	21,463,040	9,669,733	31,132,773	1060 - 4233
Matrix 6	21,463,040	9,673,530	31,136,570	1069 - 28394
Matrix 7	21,463,040	9,666,857	31,129,897	1079 - 21836
Matrix 8	21,463,040	9,670,234	31,133,274	1077 - 20148
Matrix 9	21,463,040	9,661,419	31,124,459	1067 - 12787
Matrix 10	21,463,040	9,670,316	31,133,356	1064 - 25678

TABLE 4: The Proposed Storage Format Information

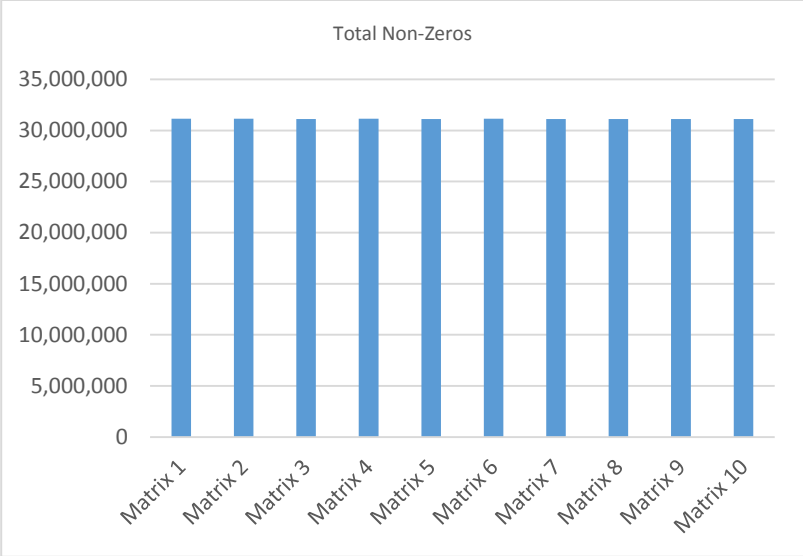

Fig. 7. Total Non-Zeros

Table 5 has some information about the sparsities of the 10 testing sparse matrices that will be stored in the proposed storage format. For each sparse matrix, it presents the sparsity of the Reference region, the sparsity of the Expansion Space region, and the overall total sparsity of the sparse matrix. Figure 8 shows the total sparsity of each of the 10 CI sparse matrices in a graphical way.

Rows:

Row n: An individual CI sparse matrix.

Cols:

Ref. Region Spar. (%): The sparsity of the Reference region.

Exp. Space Region Spar. (%): The sparsity of the Expansion Space region.

Total Spar. (%): The overall total sparsity of the sparse matrix.

	Ref. Region Spar. (%)	Exp. Space Region Spar. (%)	Total Spar. (%)
Matrix 1	80.0110	98.9985	97.0997
Matrix 2	80.0110	98.9990	97.1002
Matrix 3	80.0110	98.9997	97.1008
Matrix 4	80.0110	98.9993	97.1005
Matrix 5	80.0110	98.9994	97.1005
Matrix 6	80.0110	98.9990	97.1002
Matrix 7	80.0110	98.9997	97.1008
Matrix 8	80.0110	98.9993	97.1005
Matrix 9	80.0110	99.0002	97.1013
Matrix 10	80.0110	98.9993	97.1005

TABLE 5: Matrices Sparsities Information

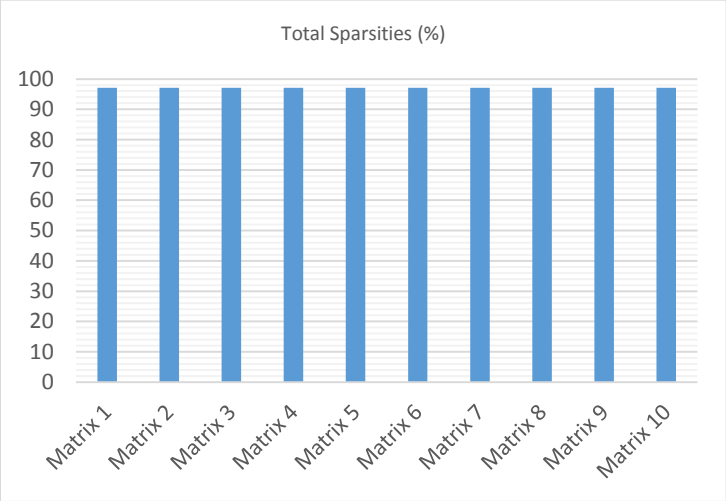

Fig. 8. Matrices Total Sparsities Information

Table 6 compares the amount of memory used (in MB) by the proposed format to the amount of memory used by the CSR format, the ELLPACK format, and the Sliced ELLPACK formats. Figure 9 presents the same previous information (memory usage) in a graphical way.

Rows:

Row n: An individual CI sparse matrix.

Cols:

CSR: The amount of memory used by the CSR format.

ELL: The amount of memory used by the ELLPACK format.

Sliced ELLPACK: The amount of memory used by the Sliced ELLPACK format.

Pro. Model: The amount of memory used (in MB) by the proposed format.

	CSR	ELL	Sliced ELLPACK	Pro. Model
Matrix 1	356.439	407.250	397.125	356.759
Matrix 2	356.460	398.25	395.062	356.699
Matrix 3	356.387	405.75	404.437	356.628
Matrix 4	356.488	403.875	397.031	356.670
Matrix 5	356.394	399.375	395.062	356.661
Matrix 6	356.381	399.75	397.593	356.704
Matrix 7	356.446	400.875	396.937	356.628
Matrix 8	356.510	401.25	396.562	356.667
Matrix 9	356.433	402.75	398.25	356.566
Matrix 10	356.437	405	403.125	356.667
Average	*356.438*	*402.413*	*398.118*	*356.665*

TABLE 6: Memory Usage

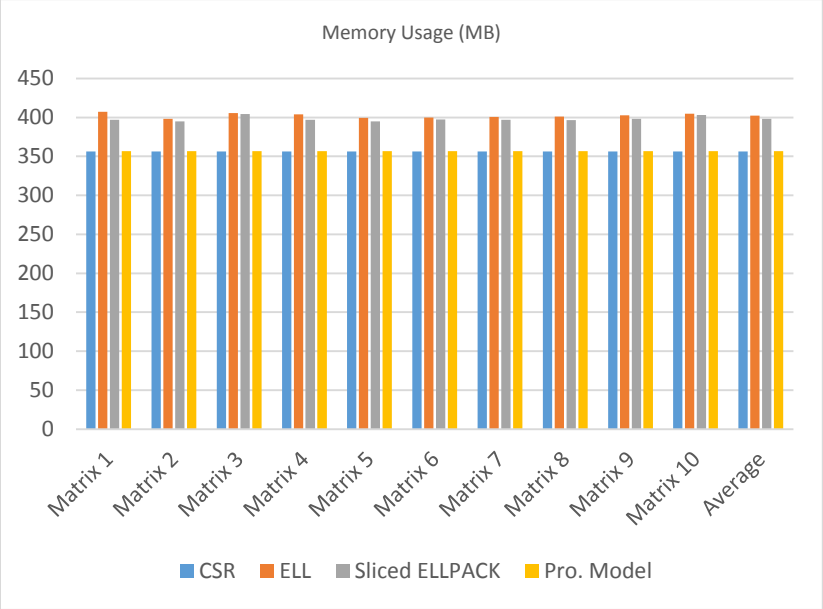

Fig. 9. Memory Usage

In the following sections, performance will be the main point. We will discuss and analyze the performance of the proposed kernel and compare it to the other kernels.

The number of threads per block (block size) you choose can and does effect the performance of the code that is running on the hardware. How each code behaves will be different and the only right way to quantify it, is by careful benchmarking and profiling. You should be aware that the block size you choose can and does have an impact on how fast your code will run, but it depends on the hardware you have and the code you are running. The number of threads per block should be a multiple of the warp size, which is 32 on all current hardware. In CUDA, in order to get the warp size for device number DevNo, you can use the following pseudocode:

cudaDeviceProp *DeviceProp_var*

cudaGetDeviceProperties(*DeviceProp_var*, DevNo)

Print *DeviceProp_var*.warpSize

We are planning on using different values for the size of the block and then run our proposed CUDA SpMV kernel. Next, we will try to pick the block size that has the highest performance on the GPU. We tried 3 different block sizes for testing: 16, 32, and 64. Finally, we will compare the proposed SpMV kernel with the best block size to the cuSPARSE library and the CSR5 (Compressed Sparse Row 5) format.

We used nvprof [54] which is a visual profiler developed by NIVIDIA, in order to analyze and view some performance data from the command line. We ran the proposed SpMV kernel with a block size (blockDim) of 16 and analyzed the performance using the nvprof tool. We used the nvprof tool while running the SpMV kernel using block sizes of 32 and 64 as well.

A subset of the output from the nvprof tool when we ran the proposed SpMV kernel using a block size of 16 is illustrated in table 7. We had run the nvprof visual profiler 5 times.

Rows:

Each row presents a single test case.

Cols:

min: The min time (in ms) that the profiler uses for running the kernel using a block size of 16.

max: The max time (in ms) that the profiler uses for running the kernel using a block size of 16.

avg: The avg time (in ms) that the profiler uses for running the kernel using a block size of 16.

Block Size	min	max	avg
16	4.8877 ms	4.8877 ms	4.8877 ms
16	4.8884 ms	4.8884 ms	4.8884 ms
16	4.8891 ms	4.8891 ms	4.8891 ms
16	4.8882 ms	4.8882 ms	4.8882 ms
16	4.8870 ms	4.8870 ms	4.8870 ms
Average	*4.8881 ms*		

TABLE 7: Running the nvprof profiler with a Block Size of 16

Table 8 illustrates a subset of the output from the nvprof tool when we ran the proposed SpMV kernel using a block size of 32. We had run the nvprof visual profiler 5 times.

Rows:

Each row presents a single test case.

Cols:

min: The min time (in ms) that the profiler uses for running the kernel using a block size of 32.

max: The max time (in ms) that the profiler uses for running the kernel using a block size of 32.

avg: The avg time (in ms) that the profiler uses for running the kernel using a block size of 32.

Block Size	min	Max	avg
32	3.2847 ms	3.2847 ms	3.2847 ms
32	3.2838 ms	3.2838 ms	3.2838 ms
32	3.2843 ms	3.2843 ms	3.2843 ms
32	3.2834 ms	3.2834 ms	3.2834 ms
32	3.2859 ms	3.2859 ms	3.2859 ms
Average	*3.2844 ms*		

TABLE 8: Running the nvprof profiler with a Block Size of 32

Table 9 illustrates a subset of the output from the nvprof tool when we ran the proposed SpMV kernel using a block size of 64. We had run the nvprof visual profiler 5 times.

Rows:

Each row presents a single test case.

Cols:

min: The min time (in ms) that the profiler uses for running the kernel using a block size of 64.

max: The max time (in ms) that the profiler uses for running the kernel using a block size of 64.

avg: The avg time (in ms) that the profiler uses for running the kernel using a block size of 64.

Block Size	min	max	avg
64	3.5451 ms	3.5451 ms	3.5451 ms
64	3.5424 ms	3.5424 ms	3.5424 ms
64	3.5437 ms	3.5437 ms	3.5437 ms
64	3.5410 ms	3.5410 ms	3.5410 ms
64	3.5448 ms	3.5448 ms	3.5448 ms
Average	*3.5434 ms*		

TABLE 9: Running the nvprof profiler with a Block Size of 64

In the previous experiments, we used the nvprof visual profiler while executing the proposed SpMV kernel and we considered different block sizes or different number of threads in each block. We tried 3 different block sizes, 16, 32, and 64. When the block size was set to 16, the average performance of the proposed SpMV kernel using the nvprof visual profiler was 4.8881 ms. When the block size was set to 32, the average performance of the proposed SpMV kernel using the nvprof visual profiler was 3.2844 ms. When the block size was set to 64, the average performance

of the proposed SpMV kernel using the nvprof visual profiler was 3.5434 ms. The information presented in Table 7, Table 8, and Table 9, show that using the nvprof visual profiler while running the proposed SpMV kernel using a block size of 32 gives us the best performance, so we will consider this block size for testing.

In the next section, we tried to compare our proposed kernel to the cuSPARSE [52] library and the CSR5 (Compressed Sparse Row 5) format [53]. The cuSPARSE library was developed by NIVIDIA in order to perform linear algebra operations on matrices and vectors. It has already implemented subroutines that deal with the SpMV operation and it is easy to use. The cuSPARSE library supports multiple data types, i.e., float, double, cuComplex, and cuDoubleComplex. The cuSPARSE library supports multiple matrix data formats, i.e., the CSR format, the CSC format, the COO format, and the hybrid format which is a combination of 2 formats, the COO and the ELLPACK formats.

We used the cusparseDcsrmv() subroutine of the cuSPARSE library for the SpMV operation. The "D" portion in the subroutine name refers to the data type of the sparse matrix's elements and the vector's elements, which is double "double precision", the "mv" portion refers to the fact that we multiply a matrix by a vector. If the elements of the sparse matrix and the vector are of type float "single precision", then we can use the cusparseScsrmv() subroutine. The subroutines cusparseCcsrmv() and cusparseZcsrmv() deal with single precision and double precision complex numbers. The cuSPARSE library also supports matrix-matrix multiplication throughout the cusparseDcsrmm() subroutine that multiplies a sparse matrix (commonly a flat sparse matrix) by a dense matrix (commonly a tall dense matrix).

The CSR5 format [53] is insensitive to the sparsity structure of the sparse matrix. The SpMV kernel of the CSR5 format has high throughput on various platforms (CPU, the GPU, and Xeon Phi).

Table 10 compares the performance (in millisecond "ms") of the proposed SpMV kernel to the performance of the cuSPARSE library and the CSR5 format. In this case, the block size (blockDim) is set to 32 for our proposed SpMV kernel, which means that each block will contain

32 threads. We ran the 3 SpMV kernels on the GPU and listed the results in the following table (Table 10). Figure 10 presents the same previous information (performance with a block size of 32) in a graphical way.

Rows:

Row n: An individual CI sparse matrix.

Cols:

cuSPARSE: The performance (in millisecond "ms") of the cuSPARSE library.

CSR5: The performance (in millisecond "ms") of the CSR5 format.

The Prop. Model: The performance (in millisecond "ms") of the proposed SpMV kernel.

	cuSPARSE	CSR5	The Prop. Model
Matrix 1	12.6567	7.3672	3.3661
Matrix 2	12.8334	8.1694	3.3573
Matrix 3	12.1227	7.8954	3.4127
Matrix 4	13.1043	8.2654	3.5434
Matrix 5	13.4352	8.1489	3.8167
Matrix 6	12.3672	8.2336	3.1884
Matrix 7	13.2559	8.3962	3.3655
Matrix 8	13.1655	7.3672	3.3784
Matrix 9	12.8985	8.5934	4.1836
Matrix 10	13.3568	7.8349	3.3471
Average	*12.9196*	*8.0272*	*3.3661*

TABLE 10: Performance with a Block Size of 32

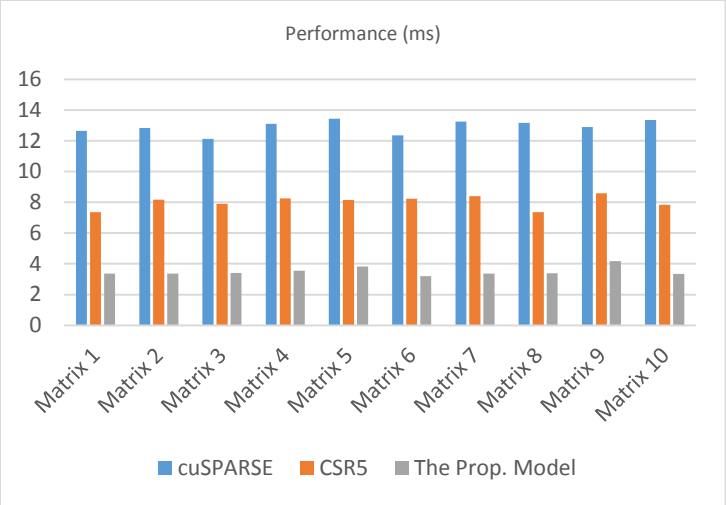

Fig. 10. Performance with a Block Size of 32

Since block size of 32 showed better performance, we tried to use it with bigger matrices. Subsequently, we tried to execute the proposed SpMV kernel using bigger matrices (1048576 by 1048576). We used the same block size which is 32. Table 10 compares the performance (in second "s") of the proposed SpMV kernel to the cuSPARSE library and the CSR5 format using 10 different 1048576 by 1048576 CI sparse matrices. We ran the proposed SpMV kernel, the cuSPARSE library, and the CSR5 format on the GPU and listed the results in table 11. Figure 11 presents the same previous information (performance in seconds with a block size of 32 using 10 different 1048576 by 1048576 CI sparse matrices) in a graphical way.

Rows:

Row n: An individual CI sparse matrix.

Cols:

cuSPARSE: The performance (in millisecond "ms") of the cuSPARSE library.

CSR5: The performance (in millisecond "ms") of the CSR5 format.

The Prop. Model: The performance (in millisecond "ms") of the proposed SpMV kernel.

	cuSPARSE	CSR5	The Prop. Model
Matrix 1	11.9564	7.8736	3.1583
Matrix 2	12.3787	8.2376	2.9672
Matrix 3	12.4513	9.5462	3.1987
Matrix 4	13.5645	7.9456	2.7761
Matrix 5	12.6587	7.9348	3.3762
Matrix 6	12.7457	7.1764	2.9376
Matrix 7	12.4874	7.3672	2.8729
Matrix 8	12.4138	7.6327	3.6534
Matrix 9	12.0845	8.3432	3.6249
Matrix 10	13.0198	7.7482	3.2653
Average	*12.5761*	*7.9806*	*3.1831*

TABLE 11: Performance with a Block Size of 32 Using 10 Different 1048576 by 1048576 CI Sparse Matrices

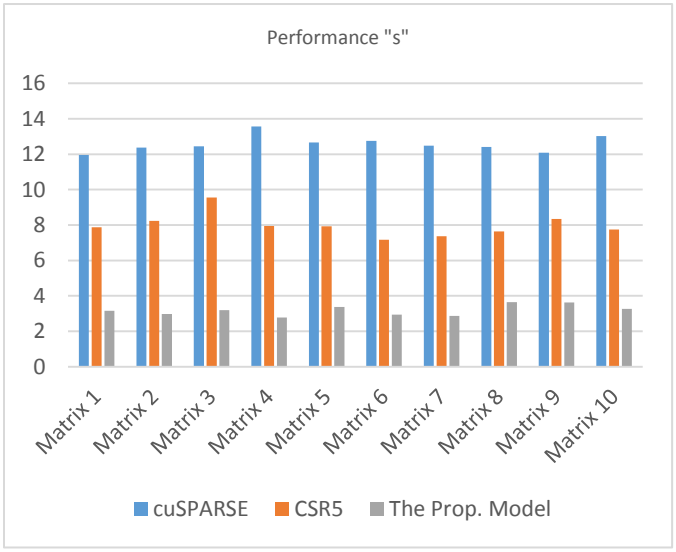

Fig. 11. Performance with a Block Size of 32 Using 10 Different 1048576 by 1048576 CI Sparse Matrices

In general, with regard to memory usage, the proposed format used less memory than the ELLPACK format and the Sliced ELLPACK format and was too close to the CSR format. In terms of performance, the proposed kernel gained better performance than the cuSPARSE library and the CSR5 (Compressed Sparse Row 5) format when we used a block size of 32. The proposed SpMV kernel outperform the other kernels and we already showed that. If we tried to use the scalar ELLPACK kernel for our proposed model, the performance would have been very low since the

number of non-zero entries in each row of the sparse matrix is not the same, the matter that leads to slowing the performance down significantly on the GPU. In the proposed model, the number of non-zero entries in each row of the ELLPACK format is guaranteed to be close, since each row starts out with the ELLPACK format's (Reference region) elements followed by the CSR format's elements, hence this guarantees high performance.

The CI sparse matrices are a special type of square sparse matrices that have a greatly-varying dimension length, they can be up to 10^7 by 10^7 or even more. CI sparse matrices have the same structure in terms of having 2 regions (the Reference Region and the Expansion Space Region) with 2 different sparsities. We used 20 matrices for testing, the first 10 matrices were 32768 by 32768, and the other 10 matrices were 1048576 by 1048576. For the first 10 matrices, they have different number of non-zero elements as illustrated in table 3. The sparsities in table 4 look close because these matrices have a huge number of elements (1073741824) and when dividing the number of zero elements by that number in order to get the total sparsity of each CI sparse matrix, the resulting percentages will look very close which holds true for most CI matrices.

In the previous part, we compared the performance of the proposed kernel to the performance of the cuSPARSE library and the CSR5 format. Generally speaking, the value of the generated error when we deal with different applications might not be the same even if all the applications use double precision. This applies to our case since we are comparing our proposed kernel to 2 different kernels (or applications) that use double precision, however the SpMV problem is considered an eigenvalue problem and the error in energy will always be ε^2 not ε.

CHAPTER 8

QUATERNIONS

Quaternions were created by William Rowan Hamilton, an Irish mathematician in 1843 [64]. He was trying to answer the question: What is the result of dividing a three-dimensional vector by another three dimensional-vector?

Quaternions are 4-dimensional objects that extend complex numbers. Generally, we go from scalars to complex numbers and from complex numbers to quaternions. We can look at quaternions as complex numbers whose components are complex numbers. When we move from complex numbers to quaternions, objects are no longer fields, whereas they algebraically form a semi-ring [66].

We can describe molecular symmetry in terms of quaternions since quaternions can be used to rotate any object, e.g. cartesian coordinates of nuclei, to a different frame. In particular, quaternions can be used to orient any molecule into a preferential, or "standard", orientation in which chemists are accustomed to describing symmetry [65].

Quaternions can be used to display objects that have more structure than the point-like nuclei. They can provide a more compact representation of 3-dimensional objects (such as the basis functions that are attached to nuclei in typical quantum chemistry calculations), and it is in this context that quaternions have seen use in different computational areas, such as image processing [67]. One consequence is that memory access times can be less than by less compact representations. The real challenge that will be faced when dealing with quaternions is performance.

Quaternions as an algebraic system can be expected to be an efficacious description of objects in quantum physics. Specifically, since quaternions are non-commutative (i.e., the result of multiplying two quaternions depends on their order), they naturally capture the non-commutative nature of many operators in quantum mechanics.

Moreover, one can suspect that quaternions will also be useful in describing rotations of multidimensional basis functions, such as those found in describing electron correlation, e.g. by Configuration Interaction (CI). In an extrapolation from the relatively well understood rotations of simple objects in 3-dimensional space, such as points or vectors, the interior structure of the CI sparse matrix should allow for more compact representation when quaternions are used rather than real scalars. However, this is a complex question that involves nontrivial developments in both quantum chemistry and scientific computing. The critical issue addressed in this dissertation work is whether a robust and efficient library that allows manipulation of quaternions as straightforwardly and efficiently as do the libraries that support operations on scalars can be obtained.

In the remainder of the chapter, we describe the desiderata of a library for quaternions for use in quantum chemical calculations. Specific features of quaternions and aspects of the algorithms used for implementation are described. Furthermore, examples of the use the quaternion library to rotate the orientation of molecules (i.e., the positions of the nuclei) are provided to help illustrate.

Quaternions have the following properties for addition:

- **Closure:** If $P, Q \in q$, then $P+Q \in q$.
- **Commutativity:** $P + Q = Q + P$ for all $P, Q \in q$.
- **Associativity:** $(P + Q) + R = P + (Q + R)$ for all $P, Q, R \in q$
- **Identity:** There is a $0 \in q$ such that $0 + P = P + 0 = P$.
- **Inverse:** For any $P \in q$, there exists a $(-P) \in q$ such that $P + (-P) = (-P) + P = 0$.

Quaternions have the following properties for multiplication:

- **Closure:** If $P, Q \in q$, then $PQ \in q$.
- **Associativity:** $(PQ)R = P(QR)$ for all $P, Q, R \in q$.
- **Identity:** There is a $1 \in q$ such that $1P = P1 = P$.
- **Inverse:** If $P \neq 0$, then there is a P^{-1} such that $PP^{-1} = P^{-1}P = 1$.

We can define a quaternion mathematically using different ways. The section below shows how to define a quaternion mathematically:

$Q = [s, v], s \in R, v \in R^3$

OR

$Q = a + bi + cj + dk, a, b, c, d \in R$

Where:

a is the scalar (real) Part

*b*i + *c*j + *d*k is the vector (imaginary) part

i (1, 0, 0), j (0, 1, 0), and k (0, 0, 1) are unit vectors in the x, y, and z direction.

Moreover, quaternions can be represented using a 2X2 complex matrix.

i, j, and k are imaginary numbers.

i

k j

If we multiply clockwise, the result will be positive, on the other hand, if we multiply counter clockwise, the result will be negative, for example, i X j = k, while j X i = -k.

We used quaternions for rotating molecules. To rotate a point we should have 3 inputs:

1. The unit vector of the rotation axis (axis of spin) we will rotate around
2. The angle we will rotate by
3. The point we will rotate

Steps:

1. Create the Rotation Quaternion. The Rotation Quaternion for rotating θ degrees around the rotation axis which has a unit vector V = $<v_x, v_y, v_z>$ is $\cos(\theta / 2) + (V.v_x * \sin(\theta / 2))i + (V.v_y * \sin(\theta / 2))j + (V.v_z * \sin(\theta / 2))k$

2. Create the Point Quaternion which is the point we want to rotate in a quaternion form. The Point Quaternion of a point P = $<p_x, p_y, p_z>$ is $0 + (P.p_x)i + (P.p_y)j + (P.p_z)k$

3. Multiply the Rotation Quaternion by the Point Quaternion in order to get a Halfway Rotation Quaternion.

4. Get the Conjugate of the Rotation Quaternion. The Conjugate of a Quaternion Q = $a + bi + cj + dk$ is $a - bi - cj - dk$

5. Multiply the Halfway Rotation Quaternion by the Conjugate of the Rotation Quaternion in order to get the rotated point.

Rotated Point = Rotation Quaternion * Point * Rotation Quaternion Conjugate

Example:

Rotate the point (1, 0, 0) by 90 degrees around the z axis

1. The unit vector of the rotation axis we will rotate around is 0x + 0y + 1z

2. The angle we will rotate by is 90

3. The point we will rotate is 1x + 0y + 0z

Solution:

Rotation Quaternion.a = cos(90 / 2) = cos(45) = 1 / sqrt(2) = 0.707106781186

Rotation Quaternion.i = 0 * sin(90 / 2) = 0

Rotation Quaternion.j = 0 * sin(90 / 2) = 0

Rotation Quaternion.k = 1 * sin(90 / 2) = 1 / sqrt(2) = 0.707106781186

Rotation Quaternion = *1 / sqrt(2)* + *0*i + *0*j + *(1 / sqrt(2))*k

Point Quaternion = 0 + *1*i + *0*j + *0*k

Rotation Quaternion					
1 / sqrt(2)	0i	0j	(1 / sqrt(2))k		*Point Quaternion*
0	0	0	0	0	
(1 / sqrt(2))i	0	0	(1 / sqrt(2))j	1i	
0	0	0	0	0j	
0	0	0	0	0k	

Halfway Rotation Quaternion = 0 + (1 / sqrt(2))i + (1 / sqrt(2))j + 0k

Rotation Quaternion Conjugate = 1 / sqrt(2) - 0i - 0j - (1 / sqrt(2))k

Halfway Rotation Quaternion					
0	(1 / sqrt(2))i	(1 / sqrt(2))j	0k		*Rotation Quaternion Conjugate*
0	(1.0/2.0)i	(1.0/2.0)j	0	1 / sqrt(2)	
0	0	0	0	0i	
0	0	0	0	0j	
0	(1.0/2.0)j	- (1.0/2.0)i	0	- (1 / sqrt(2))k	

The Rotated Point Quaternion = *0* + *0*i + *1*j + *0*k

The Rotated Point = *0*x + *1*y + *0*z

Another way using Rotation Matrices:

$$R_x(\theta) = \begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{vmatrix}$$

$$R_y(\theta) = \begin{vmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{vmatrix}$$

$$R_z(\theta) = \begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

$$R_z(\theta) \begin{vmatrix} \cos 90 & -\sin 90 & 0 \\ \sin 90 & \cos 90 & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix} = \begin{vmatrix} 0 \\ 1 \\ 0 \end{vmatrix}$$

The Rotated Point = *0*x + *1*y + *0*z

Quaternion Invariants:

Commutative							
Yes	a * a ∈ R	a * i ∈ i	i * a ∈ i	a * j ∈ j	j * a ∈ j	a * k ∈ k	k * a ∈ k
No	i X j = k	j X i = -k	j X k = i	k X j = -i	k X i = j	i X k = -j	
Yes	i . j = 0	j . i = 0	j . k = 0	k . j = 0	k . i = 0	i . k = 0	
	i X i = 0		j X j = 0		k X k = 0		
	i . i = 1		j . j = 1		k . k = 1		
	i * i = -1	$i^2 = -1$	j * j = -1	$j^2 = -1$	k * k = -1	$k^2 = -1$	i * j * k = -1

Real Quaternion

A Quaternion with a vector part of zero 0

Q = [s, 0]

Q = $a + 0i + 0j + 0k$

Pure Quaternion

A Quaternion with a scalar part of zero 0

Q = [0, v]

Q = $0 + bi + cj + dk$

Addition Identity Quaternion

Q = [0, 0]

Q = $0 + 0i + 0j + 0k$

Quaternion + Addition Identity Quaternion = Quaternion

Multiplication Identity Quaternion

The Multiplication Identity Quaternion is considered a Real Quaternion

Q = [1, 0]

$Q = 1 + 0i + 0j + 0k$

Quaternion * Multiplication Identity Quaternion = Quaternion (No Rotation)

Quaternion Magnitude, Length, or Norm ||Q||:

$Q = a + bi + cj + dk$

$||Q|| = \sqrt{(a)^2 + (b)^2 + (c)^2 + (d)^2}$

$||Q|| = \sqrt{Q \cdot Q} = \sqrt{Q * Conjugate(Q)} = \sqrt{Conjugate(Q) * Q}$

$(||Q||)^2 = Q \cdot Q = Q * Conjugate(Q) = Conjugate(Q) * Q$

$||Q1 * Q2|| = ||Q1|| * ||Q2||$

Unit Quaternion (Quaternion Normalization):

A Normalized Quaternion is a Unit Quaternion of length 1

$Q = a + bi + cj + dk$

Normalized Quaternion = Q / Magnitude(Q) = Q.a / Magnitude(Q) + (Q.i / Magnitude(Q))i + (Q.j / Magnitude(Q))j + (Q.k / Magnitude(Q))k

Normalized Quaternion.a = a / Magnitude(Q)

Normalized Quaternion.i = b / Magnitude(Q)

Normalized Quaternion.j = c / Magnitude(Q)

Normalized Quaternion.k = d / Magnitude(Q)

Conjugate(Unit Quaternion) = Inverse(Unit Quaternion)

Rotation Quaternion:

A Rotation Quaternion is a Unit Quaternion of length 1

Rotation Quaternion * Rotation Quaternion = Unit Quaternion of length 1

Rotation Quaternion = cos(Rotation Angle in Radians / 2) + Rotation Axis Unit Vector * sin(Rotation Angle in Radians / 2)

Assume:

Rotation Axis Unit Vector, V = $<v_x, v_y, v_z>$

Rotation Angle in Radians = Θ

Then:

Rotation Quaternion.a = cos(Θ / 2.0)

Rotation Quaternion.i = V.v_x * sin(Θ / 2.0)

Rotation Quaternion.j = V.v_y * sin(Θ / 2.0)

Rotation Quaternion.k = V.v_z * sin(Θ / 2.0)

If the Rotation Axis Unit Vector = <0, 0, 0> and the Rotation Angle = 0.0 then:

The Rotation Quaternion = 1 + *0*i + *0*j + *0*k, (Multiplication Identity Quaternion, No Rotation)

Get Rotation Axis and Rotation Angle from Rotation Quaternion:

Q = *a* + *b*i + *c*j + *d*k

Rotation angle in Radians (Θ) = $\cos^{-1}(a)$ * 2.0

Rotation Axis Unit Vector.v_x = *b* / sin(Θ / 2.0)

Rotation Axis Unit Vector.v_y = *c* / sin(Θ / 2.0)

Rotation Axis Unit Vector.v_z = *d* / sin(Θ / 2.0)

$\cos^2(\Theta) + \sin^2(\Theta) = 1$

$\cos^2(\Theta / 2.0) + \sin^2(\Theta / 2.0) = 1$

$\sin(\Theta / 2.0) = \text{sqrt}(1.0 - \cos^2(\Theta / 2.0)) = \text{sqrt}(1.0 - a^2)$

Point Quaternion:

A Point Quaternion is considered a Pure Quaternion. It's a point in a Quaternion form.

Point, P = $<p_x, p_y, p_z>$

Point Quaternion = $0 + (P.p_x)i + (P.p_y)j + (P.p_z)k$

Point Quaternion.a = *0*

Point Quaternion.i = $P.p_x$

Point Quaternion.j = $P.p_y$

Point Quaternion.k = $P.p_z$

Quaternion Conjugate:

$Q = a + bi + cj + dk$

$\text{Conjugate}(Q) = a - bi - cj - dk$

Conjugate(Q).a = *a*

Conjugate(Q).i = - *b*

Conjugate(Q).j = - *c*

Conjugate(Q).k = - *d*

We can get the conjugate of a rotation quaternion by rotating in the opposite direction, so we use -Θ (360 - Θ) instead of Θ. Cos(Θ) = cos(-Θ) and sin(-Θ) = -sin(Θ). That is why the scalar part didn't change and the vector (Imaginary part) was negated.

Quaternion Inverse (Reciprocal):

$Q = a + bi + cj + dk$

$Inverse(Q) = Q^{-1} = 1 / Q$ *"Multiply the nominator and the denominator by Conjugate(Q)"*

$= Conjugate(Q) / (Q * Conjugate(Q))$

$= Conjugate(Q) / (Magnitude(Q))^2$

$= (a - bi - cj - dk) / (a)^2 + (b)^2 + (c)^2 + (d)^2$

$Inverse(Q).a = a / (a)^2 + (b)^2 + (c)^2 + (d)^2$

$Inverse(Q).i = - b / (a)^2 + (b)^2 + (c)^2 + (d)^2$

$Inverse(Q).j = - c / (a)^2 + (b)^2 + (c)^2 + (d)^2$

$Inverse(Q).k = - d / (a)^2 + (b)^2 + (c)^2 + (d)^2$

$Q * Q^{-1} = 1$

$Q_1 * Q_2 * Q_2^{-1} = Q_1$

$Q_2 * Q_1 * Q_2^{-1} != Q_1$

Adding Quaternions:

$Q_1 = a_1 + b_1i + c_1j + d_1k$

$Q_2 = a_2 + b_2i + c_2j + d_2k$

$Q_1 + Q_2 = [s_1, v_1] + [s_2, v_2] = [s_1 + s_2, v_1 + v_2]$

$Q_1 + Q_2 = (a_1 + a_2) + (b_1 + b_2)i + (c_1 + c_2)j + (d_1 + d_2)k$

$Resulting\ Quaternion.a = a_1 + a_2$

Resulting Quaternion.i = $b_1 + b_2$

Resulting Quaternion.j = $c_1 + c_2$

Resulting Quaternion.k = $d_1 + d_2$

Subtracting Quaternions:

$Q_1 = a_1 + b_1i + c_1j + d_1k$

$Q_2 = a_2 + b_2i + c_2j + d_2k$

$Q_1 - Q_2 = [s_1, v_1] - [s_2, v_2] = [s_1 - s_2, v_1 - v_2]$

$Q_1 - Q_2 = (a_1 - a_2) + (b_1 - b_2)i + (c_1 - c_2)j + (d_1 - d_2)k$

Resulting Quaternion.a = $a_1 - a_2$

Resulting Quaternion.i = $b_1 - b_2$

Resulting Quaternion.j = $c_1 - c_2$

Resulting Quaternion.k = $d_1 - d_2$

Multiplying Quaternions:

When you combine Vectors, you have to add them.

When you combine Quaternions, you have to multiply them.

$Q_1 = a_1 + b_1i + c_1j + d_1k$

$Q_2 = a_2 + b_2i + c_2j + d_2k$

$Q_1 * Q_2 = [s_1, v_1] * [s_2, v_2] = [(s_1 * s_2) – (v_1 . v_2), (s_1 * v_2) + (s_2 * v_1) + (v_1 \times v_2)]$

Proof:

$Q_1 * Q_2 = [a_1 + b_1i + c_1j + d_1k] * [a_2 + b_2i + c_2j + d_2k]$

$= a_1a_2 + a_1b_2i + a_1c_2j + a_1d_2k$

$+ b_1ia_2 + b_1ib_2i + b_1ic_2j + b_1id_2k$

$+ c_1ja_2 + c_1jb_2i + c_1jc_2j + c_1jd_2k$

$+ d_1ka_2 + d_1kb_2i + d_1kc_2j + d_1kd_2k$

$= ((a_1a_2) - (b_1b_2) - (c_1c_2) - (d_1d_2))$

$+ ((a_1b_2) + (b_1a_2) + (c_1d_2) - (d_1c_2))i$

$+ ((a_1c_2) + (c_1a_2) + (d_1b_2) - (b_1d_2))j$

$+ ((a_1d_2) + (d_1a_2) + (b_1c_2) - (c_1b_2))k$

$= ((a_1a_2) - (b_1b_2) - (c_1c_2) - (d_1d_2))$ => $(s_1 * s_2) - (v_1 . v_2)$

$+ (a_1b_2)i + (a_1c_2)j + (a_1d_2)k$ => $s_1 * v_2$

$+ (a_2b_1)i + (a_2c_1)j + (a_2d_1)k$ => $s_2 * v_1$

$+ ((c_1d_2) - (d_1c_2))i + (-(b_1d_2) + (d_1b_2))j + ((b_1c_2) - (c_1b_2))k$ => $v_1 \times v_2$

$= ((a_1a_2) - (b_1b_2) - (c_1c_2) - (d_1d_2))$ => $(s_1 * s_2) - (v_1 . v_2)$

$+ a_1(b_2i + c_2j + d_2k)$ => $s_1 * v_2$

$+ a_2(b_1i + c_1j + d_1k)$ => $s_2 * v_1$

$+ ((c_1d_2) – (d_1c_2))i + (– (b_1d_2) + (d_1b_2))j + ((b_1c_2) – (c_1b_2))k \quad => \quad v_1 \times v_2$

Resulting Quaternion.a = $(a_1a_2) – (b_1b_2) – (c_1c_2) – (d_1d_2)$

Resulting Quaternion.i = $(a_1b_2) + (b_1a_2) + (c_1d_2) – (d_1c_2)$

Resulting Quaternion.j = $(a_1c_2) + (c_1a_2) + (d_1b_2) – (b_1d_2)$

Resulting Quaternion.k = $(a_1d_2) + (d_1a_2) + (b_1c_2) – (c_1b_2)$

Dividing Quaternions:

$Q_1 = a_1 + b_1i + c_1j + d_1k$

$Q_2 = a_2 + b_2i + c_2j + d_2k$

$Q_1 / Q_2 = Q_1 * \text{Inverse}(Q_2)$

$= Q1 * (\text{Conjugate}(Q2) / (\text{Magnitude}(Q_2))^2)$

$= Q1 * ((a_2 - b_2i - c_2j - d_2k) / ((a_2)^2 + (b_2)^2 + (c_2)^2 + (d_2)^2))$

Multiplying a Quaternion by a Scalar

$Q = a + bi + cj + dk$

Scalar Value = s

$Q * s = a * s + (b * s)i + (c * s)j + (d * s)k$

Resulting Quaternion.a = $a * s$

Resulting Quaternion.i = $b * s$

Resulting Quaternion.j = $c * s$

Resulting Quaternion.k = $d * s$

Dividing a Quaternion by a Scalar

$Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$

Scalar Value = s

$Q / s = a / s + (b / s)\mathrm{i} + (c / s)\mathrm{j} + (d / s)\mathrm{k}$

Resulting Quaternion.a = a / s

Resulting Quaternion.i = b / s

Resulting Quaternion.j = c / s

Resulting Quaternion.k = d / s

Multiplying a Quaternion by itself

$Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$

$Q_1 * Q_2 = [s_1, v_1] * [s_2, v_2] = [(s_1 * s_2) - (v_1 . v_2), (s_1 * v_2) + (s_2 * v_1) + (v_1 \times v_2)]$

$Q * Q = (a^2 - b^2 - c^2 - d^2) + (2 * a * b)\mathrm{i} + (2 * a * c)\mathrm{j} + (2 * a * d)\mathrm{k}$

Resulting Quaternion.a = $a^2 - b^2 - c^2 - d^2$

Resulting Quaternion.i = $2 * a * b$

Resulting Quaternion.j = $2 * a * c$

Resulting Quaternion.k = $2 * a * d$

Multiplying a Quaternion by its Conjugate

$Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$

$Q_1 * Q_2 = [s_1, v_1] * [s_2, v_2] = [(s_1 * s_2) - (v_1 . v_2), (s_1 * v_2) + (s_2 * v_1) + (v_1 \times v_2)]$

$Q * \text{Conjugate}(Q) = (a^2 + b^2 + c^2 + d^2) + 0\mathrm{i} + 0\mathrm{j} + 0\mathrm{k}$

Resulting Quaternion.a = $(a^2 + b^2 + c^2 + d^2)$

Resulting Quaternion.i = 0

Resulting Quaternion.j = 0

Resulting Quaternion.k = 0

Quaternions Dot Product:

$Q_1 = a_1 + b_1 i + c_1 j + d_1 k$

$Q_2 = a_2 + b_2 i + c_2 j + d_2 k$

$Q_1 . Q_2 = (a_1 * a_2) + (b_1 * b_2) + (c_1 * c_2) + (d_1 * d_2)$

Point Quaternions Cross Product:

$Q_1 = 0 + b_1 i + c_1 j + d_1 k$

$Q_2 = 0 + b_2 i + c_2 j + d_2 k$

1. Multiply the two point quaternions to get a resulting quaternion.
2. The scalar part of the resulting quaternion is negative of the dot product of the two "vector parts" of the two "point quaternions".
3. The vector part of the resulting quaternion is the cross product of the two "vector parts" of the two "point quaternions".

Scalar Part($Q_1 * Q_2$) = Vector Part(Q_1) . Vector Part(Q_2) * -1

Vector Part($Q_1 * Q_2$) = Vector Part(Q_1) x Vector Part(Q_2)

$Q_1 * Q_2$ = [Vector Part(Q_1) . Vector Part(Q_2) * -1, Vector Part(Q_1) x Vector Part(Q_2)]

Proof:

$Q_1 * Q_2 = [s_1, v_1] * [s_2, v_2] = [(s_1 * s_2) – (v_1 . v_2), (s_1 * v_2) + (s_2 * v_1) + (v_1 x v_2)]$

$= [0 - v_1 . v_2, (0, 0, 0) + (0, 0, 0) + v_1 \times v_2] = [-(v_1 . v_2), v_1 \times v_2]$

The Angle Between two Quaternions:

$Q_1 = a_1 + b_1 i + c_1 j + d_1 k$

$Q_2 = a_2 + b_2 i + c_2 j + d_2 k$

Q_1 and Q_2 should be Unit Quaternions "Normalized"

$Q_1 . Q_2 = || Q_1|| * || Q_2|| * \cos(\text{Angle} / 2)$

$\cos(\text{Angle} / 2) = Q_1 . Q_2 / || Q_1|| * || Q_2||$. "$Q_1$ and Q_2 are Unit Quaternions"

$\cos(\text{Angle} / 2) = Q_1 . Q_2$

Angle between Q_1 and Q_2 in Radians = $\cos^{-1}(Q_1 . Q_2) * 2.0$

The Distance Between two Quaternions:

$Q_1 = a_1 + b_1 i + c_1 j + d_1 k$

$Q_2 = a_2 + b_2 i + c_2 j + d_2 k$

Q_1 and Q_2 should be Unit Quaternions "Normalized"

Distance between Q_1 and $Q_2 = 1.0 - (Q_1 . Q_2)^2$

Distance between Q_1 and Q_2 = (1.0 – cos(Angle Between Quaternions)) / 2

Converting a Quaternion to a Matrix:

$Q = a + bi + cj + dk$

Resulting Matrix = $|1 - 2c^2 - 2d^2 \quad 2bc - 2ad \quad 2bd + 2ac \quad 0|$

$$\begin{vmatrix} 2bc + 2ad & 1 - 2b^2 - 2d^2 & 2cd - 2ab & 0 \\ 2bd - 2ac & 2cd + 2ab & 1 - 2b^2 - 2c^2 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Slerp (Spherical Linear Interpolation):

$Q_1 = a_1 + b_1 i + c_1 j + d_1 k$

$Q_2 = a_2 + b2i + c2j + d2k$

Q_1 and Q_2 should be Unit Quaternions "Normalized".

Θ -> Half the angle in radians between Q_1 and Q_2

t -> Percent

Slerp Quaternion = $[\sin((1.0 - t) * \Theta) / \sin(\Theta)] * Q_1 + [\sin(t * \Theta) / \sin(\Theta)] * Q_2$

Another Way:

$Q * Q_1 = Q_2$ Multiply both sides by Q_1^{-1}

$Q * Q_1 * Q_1^{-1} = Q_2 * Q_1^{-1}$ $Q_1 * Q_1^{-1} = 1$

$Q = Q_2 * Q_1^{-1}$

$Q = \cos(\Theta / 2) + V * \sin(\Theta / 2)$

Then we can get the Rotation Axis Unit Vector, *V* and Rotation Angle in Radians, *Θ*

$Q^t = \cos((t * \Theta) / 2) + V * \sin((t * \Theta) / 2)$

Slerp Quaternion = $Q^t * Q_1 = (Q_2 * Q_1^{-1})^t * Q_1$

Chapter Summary:

Quaternions are 4-dimensional objects that extend complex numbers. They were created by William Rowan Hamilton in 1843. Quaternions can be used to display objects that have more sophisticated structure than the point-like nuclei.

There are some tangible advantages that come along with using quaternions. Quaternions introduce no gimbal lock, as opposed to rotation matrices and Euler angles; gimbal lock happens when two axes effectively line up, resulting in a temporary loss of a degree of freedom. With regard to rotation, quaternions are allocated less memory (4 scalars) than a 3x3 rotation matrix (9 scalars). Regarding performance, quaternions multiplication is much faster than a 3x3 matrix multiplication, rotating matrices require an evaluation of sin() and cos(). Besides, multiplying a quaternion by a vector (dense or sparse) is going to be much faster than a full matrix-vector multiplication.

Quaternions prove useful for coordinate transformations. The quaternion method is much better if one of the coordinate systems keeps moving, which is the usual case with navigation and animated 3-D graphics.

When the scalars that make up the CI sparse matrix get replaced with quaternions, the interior structure of the CI sparse matrix will be represented in a more compact structure due to the nature of quaternions [68], hence memory reduction will be tangible and pronounced.

When representing the CI sparse matrix using quaternions, multiplying the CI sparse matrix by a vector will be noticeably faster due to the fact that quaternions multiplication is much faster than normal matrix multiplication.

Quaternions can also be used when multiplying a sparse matrix by a sparse vector, rather than a dense vector (SpMV), in this case quaternions will be used to represent both the sparse matrix and the sparse vector in a more compact way rather than using scalars.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

In this study, we are proposing a new model for storing CI sparse matrices on the GPU. We have also implemented the kernel of the SpMV operation for the proposed model. We have started out by creating the storage format for the newly-developed proposed model. This storage format will be used to store CI matrices on the GPU. We have used this proposed format in order to develop the SpMV kernel for the proposed model. The previously-mentioned SpMV kernel will be used to multiply the compact representation of the CI sparse matrix by a vector. The proposed SpMV kernel that we have developed is a vector kernel that uses the warp technique. The proposed kernel was compared to the cuSPARSE library and the CSR5 (Compressed Sparse Row 5) format using different input sparse matrices and already outperformed both of them.

We have compared the proposed format to the other common formats using different input sparse matrices. The proposed format used less memory than the ELLPACK format and the Sliced ELLPACK format.

The proposed storage format and the developed SpMV kernel work efficiently with CI sparse matrices, but this does not mean that the proposed model can be used efficiently with any sparse matrix. The proposed model including the proposed storage format and the developed SpMV kernel, was created and developed specifically for CI sparse matrices, it is not a general purpose model.

The CI sparse matrices have a specific structure and the algorithm that we have developed is well-suited for them. The developed algorithm includes the use of 2 warps (one for each row in the Reference Region and another warp for each row in the Expansion Space Region) with an efficient level of synchronization. The proposed algorithm didn't use tiles as in CSR5, which is not a practical solution for CI matrices due to their irregular sparsity pattern through the whole entire matrix, rather used 2 warps that efficiently serve the 2 main regions of the CI matrix. Although the Expansion space region has a very small sparsity, it's huge especially when we deal

with big CI matrices, we stored the Expansion space region in the CSR format that is simple and uses very minimal space on the GPU compared to the CSR5 format.

I have learnt some lessons from my work on SpMV problems in the context of high performance computing (HPC). One lesson is that using high performance computing as a part of solving a problem will not always be necessary, in some cases it might add overhead to the algorithm, hence using the CPU will be more efficient. Another lesson that I learnt is that an efficient algorithm that works for one type of problems will not necessarily work for other problems. For example, an SpMV kernel that is very efficient for a specific type of sparse matrices will not necessarily be efficient and successful for other types of sparse matrices.

The proposed model is expected to be extendable to a wider class of algebraic objects than real scalars. In particular, it is expected that it is extendable all the way to quaternions, but implementation is outside the scope of this dissertation. Quaternions are 4-dimensional objects, where $Q = [s, v]$, $s \in R$, $v \in R^3$, $Q = a + bi + cj + dk$, $a, b, c, d \in R$. $i^2 = -1$, $j^2 = -1$, $k^2 = -1$, $ijk = -1$. They extend complex numbers. When we move from complex numbers to quaternions, objects are no longer fields, whereas they algebraically form a semi-ring. Specifically, we lose commutativity which is not necessarily a disadvantage, since many operations in quaternions are not commutative. For example, $ij = k$ whilst $ji = -k$. Subsequently, the proposed model (quaternions version) should be compared with the quaternions versions of the already developed models. Of course, the comparison will be based on the two crucial key factors: the amount of used memory and performance.

In Chapter 8, we have discussed quaternions by giving a handy introduction about Quaternions features and functions. Subsequently in Appendix 1, I will be talking about the Quaternions package that I have developed in order to widely deal with Quaternions functions.

Also, I am planning to continue working on SpMV, but instead of dealing with dense vectors (my case), I might consider working on problems that involve sparse vectors; in this case, in addition to creating a storage format for the sparse matrix, I will have to create a separate storage format for the sparse vector as well.

APPENDIX 1 - THE QUATERNIONS PACKAGE

In this section, I will list all the functions that are contained in the Quaternions package that we have developed. They almost covert all the Quaternions functionality.

---

struct Quaternion CreateQuaternionByComponents(double a, double i, double j, double k)

```
{
 struct Quaternion Q;

 Q.a = a;
 Q.i = i;
 Q.j = j;
 Q.k = k;

 return Q;
}
```

---

struct Quaternion CreateQuaternionByScalarAndVector(double Scalar, struct Vector V)

```
{
struct  Quaternion Q;

Q.a = Scalar;
Q.i = V.Vx;
Q.j = V.Vy;
Q.k = V.Vz;

return Q;
}
```

---

struct Quaternion CreateQuaternionByArray(double *QuaternionComponents)

```
{
 struct Quaternion Q;

 Q.a = *QuaternionComponents;
 Q.i = *(QuaternionComponents + 1);
 Q.j = *(QuaternionComponents + 2);
 Q.k = *(QuaternionComponents + 3);

 return Q;
}
```

void PrintQuaternion(const char *QuaternionName, struct Quaternion Q, int Precision)

```
{
 char *ComponentFormat = NULL;

 ComponentFormat = (char*) calloc(BufferSize + 1, sizeof(char));

 sprintf(ComponentFormat, "%%.%df", Precision);
 printf("%s = ", QuaternionName);
 printf(ComponentFormat, Q.a);
 if(Q.i >= 0.0) {  printf(" + "); printf(ComponentFormat, Q.i); } else if(Q.i < 0.0) { printf(" - ");
printf(ComponentFormat, Q.i * -1); } printf("i");
 if(Q.j >= 0.0) {  printf(" + "); printf(ComponentFormat, Q.j); } else if(Q.j < 0.0) { printf(" - ");
printf(ComponentFormat, Q.j * -1); } printf("j");
 if(Q.k >= 0.0) {  printf(" + "); printf(ComponentFormat, Q.k); } else if(Q.k < 0.0) { printf(" - ");
printf(ComponentFormat, Q.k * -1); } printf("k\n");

 free(ComponentFormat);
}
```

double GetQuaternionScalarPart(struct Quaternion Q)

```
{
 double ScalarPart = 0.0;

 ScalarPart = Q.a;

 return ScalarPart;
}
```

struct Vector GetQuaternionVectorPart(struct Quaternion Q)

```
{
 struct Vector VectorPart;

 VectorPart.Vx = Q.i;
 VectorPart.Vy = Q.j;
 VectorPart.Vz = Q.k;

 return VectorPart;
}
```

```
double* GetQuaternionComponents(struct Quaternion Q)
{
 double *QuaternionComponents = NULL;

 QuaternionComponents = (double*) calloc(4 + 1, sizeof(double));

 *QuaternionComponents       = Q.a;
 *(QuaternionComponents + 1) = Q.i;
 *(QuaternionComponents + 2) = Q.j;
 *(QuaternionComponents + 3) = Q.k;

 return QuaternionComponents;
}
```

```
double GetQuaternionMagnitude(struct Quaternion Q)
{
 double QuaternionMagnitude = 0.0; /**** Quaternion Magnitude = Quaternion Length =
Quaternion Norm = Sqrt(Quaternion . Quaternion) = Sqrt(Quaternion * Conjugate(Quaternion))
= Sqrt(Conjugate(Quaternion) * Quaternion) ****/

 QuaternionMagnitude = sqrt(pow(Q.a, 2) + pow(Q.i, 2) + pow(Q.j, 2) + pow(Q.k, 2));

 return QuaternionMagnitude;
}
```

```
double GetQuaternionMagnitudeSquared(struct Quaternion Q)
{
 double QuaternionMagnitudeSquared = 0.0;

 QuaternionMagnitudeSquared = pow(Q.a, 2) + pow(Q.i, 2) + pow(Q.j, 2) + pow(Q.k, 2);

 return QuaternionMagnitudeSquared;
}
```

```
struct Quaternion NormalizeQuaternion(struct Quaternion Q)
{
 double QuaternionMagnitude = 0.0;
 struct Quaternion NormalizedQuaternion; /**** A Normalized Quaternion is a Unit Quaternion
of length 1 ****/
```

```
 QuaternionMagnitude = GetQuaternionMagnitude(Q);

 if(QuaternionMagnitude == 0.0)
 {
  return CreateQuaternionByComponents(0.0, 0.0, 0.0, 0.0);
 }

 NormalizedQuaternion.a = Q.a / QuaternionMagnitude;
 NormalizedQuaternion.i = Q.i / QuaternionMagnitude;
 NormalizedQuaternion.j = Q.j / QuaternionMagnitude;
 NormalizedQuaternion.k = Q.k / QuaternionMagnitude;

 return NormalizedQuaternion;
}
```

---

int EqualQuaternions(struct Quaternion Q1, struct Quaternion Q2, double Tolerance)

```
{
 int EqualQuaternions = 0;

 if(fabs(Q1.a - Q2.a) <= Tolerance && fabs(Q1.i - Q2.i) <= Tolerance && fabs(Q1.j - Q2.j) <=
Tolerance && fabs(Q1.k - Q2.k) <= Tolerance)
 {
  EqualQuaternions = 1;
 }

 return EqualQuaternions;
}
```

---

double GetQuaternionsDotProduct(struct Quaternion Q1, struct Quaternion Q2)

```
{
 double DotProduct = 0.0;

 DotProduct = (Q1.a * Q2.a) + (Q1.i * Q2.i) + (Q1.j * Q2.j) + (Q1.k * Q2.k);

 return DotProduct;
}
```

---

struct Vector GetQuaternionsCrossProduct(struct Quaternion Q1, struct Quaternion Q2, double *DotProduct)

```
{
```

```
 /**** The Quaternions cross product is the cross product of the two "vector parts" of the two
"Point Quaternions" ****/

 struct Quaternion MultiplicationQuaternions;
 struct Vector CrossProduct;

 Q1.a = 0;
 Q2.a = 0;
 MultiplicationQuaternions = MultiplyQuaternions(Q1, Q2);
 CrossProduct            = GetQuaternionVectorPart(MultiplicationQuaternions);
 *DotProduct             = GetQuaternionScalarPart(MultiplicationQuaternions) * -1;  /****
DotProduct is the dot product of the two "Point Quaternions" ****/


 /**** Another Way ****/
 /*
 struct Vector Q1VectorPart;
 struct Vector Q2VectorPart;
 struct Vector CrossProduct;

 Q1VectorPart = GetQuaternionVectorPart(Q1);
 Q2VectorPart = GetQuaternionVectorPart(Q2);
 CrossProduct = GetVectorsCrossProduct(Q1VectorPart, Q2VectorPart);
 */
 return CrossProduct;
}
```

---

double GetAngleBetweenTwoQuaternions(struct Quaternion Q1, struct Quaternion Q2)

```
{
 double DotProduct        = 0.0;
 double AngleInRadians    = 0.0;
 double AngleInDegrees    = 0.0;

 DotProduct = GetQuaternionsDotProduct(Q1, Q2);

 /**** The parameter of acos() function is in the interval [-1, 1]. The return value of acos is in
the interval [0, pi] radians. We have to do the following check ****/
 if(DotProduct < -1)
 {
  DotProduct = -1;
 }
 else if(DotProduct > 1)
 {
  DotProduct = 1;
 }
```

```
 AngleInRadians = 2.0 * acos(DotProduct);
 AngleInDegrees = AngleInRadians * RadiansToDegrees;

 return AngleInDegrees;
}
```

---

double GetAngleBetweenTwoQuaternions_2(struct Quaternion Q1, struct Quaternion Q2)

```
{
 double DotProduct     = 0.0;
 double Value          = 0.0;
 double AngleInRadians = 0.0;
 double AngleInDegrees = 0.0;

 DotProduct = GetQuaternionsDotProduct(Q1, Q2);
 Value      = (2.0 * pow(DotProduct, 2)) - 1.0;

 /**** The parameter of acos() function is in the interval [-1, 1]. The return value of acos is in
the interval [0, pi] radians. We have to do the following check ****/
 if(Value < -1)
 {
  Value = -1;
 }
 else if(Value > 1)
 {
  Value = 1;
 }

 AngleInRadians = acos(Value);
 AngleInDegrees = AngleInRadians * RadiansToDegrees;

 return AngleInDegrees;
}
```

---

double GetDistanceBetweenTwoQuaternions(struct Quaternion Q1, struct Quaternion Q2)

```
{
 double DotProduct = 0.0;
 double Distance   = 0.0;

 DotProduct = GetQuaternionsDotProduct(Q1, Q2);
 Distance   = 1.0 - pow(DotProduct, 2);

 return Distance;
```

```
}
```

---

double GetDistanceBetweenTwoQuaternions_2(struct Quaternion Q1, struct Quaternion Q2)

```
{
 double AngleInDegrees = 0.0;
 double AngleInRadians = 0.0;
 double Distance       = 0.0;

 AngleInDegrees = GetAngleBetweenTwoQuaternions(Q1, Q2);
 AngleInRadians = AngleInDegrees * DegreesToRadians;
 Distance = (1.0 - cos(AngleInRadians)) / 2.0;

 return Distance;
}
```

---

struct Quaternion CreateRotationQuaternion(struct Vector RotationAxisUnitVector, double RotationAngleInDegrees)

```
{
 /****
  RotationAxisUnitVector -> The axis to rotate around
  RotationAngleInDegrees -> The angle to rotate by
  RotationQuaternion     -> The Rotation Quaternion that results from rotating
"RotationAngleInDegrees" degrees around the rotation axis which has
"RotationAxisUnitVector" unit vector
  ****/

 double RotationAngleInRadians = 0.0;
 struct Quaternion RotationQuaternion;

 /**** When rotating about the Y-Axis, Quaternions rotate clockwise, that is why we have to do
the following operation ****/
 if(RotationAxisUnitVector.Vx == 0 && RotationAxisUnitVector.Vy == 1 &&
RotationAxisUnitVector.Vz == 0)
 {
   RotationAngleInDegrees = 360 - RotationAngleInDegrees;
 }

 RotationAngleInRadians = RotationAngleInDegrees * DegreesToRadians;
 RotationQuaternion.a   = cos(RotationAngleInRadians / 2.0);
 RotationQuaternion.i   = RotationAxisUnitVector.Vx * sin(RotationAngleInRadians / 2.0);
 RotationQuaternion.j   = RotationAxisUnitVector.Vy * sin(RotationAngleInRadians / 2.0);
 RotationQuaternion.k   = RotationAxisUnitVector.Vz * sin(RotationAngleInRadians / 2.0);
```

```
 return RotationQuaternion;
}
```

---

struct Vector GetAxisAndAngleFromRotationQuaternion(struct Quaternion RotationQuaternion, double *RotationAngleInDegrees)

```
{
 double RotationAngleInRadians = 0.0;
 struct Vector RotationAxisUnitVector;

 RotationAngleInRadians    = acos(RotationQuaternion.a) * 2.0;
 *RotationAngleInDegrees   = RotationAngleInRadians * RadiansToDegrees;
 RotationAxisUnitVector.Vx = (RotationQuaternion.i) / sin(RotationAngleInRadians / 2.0);
 RotationAxisUnitVector.Vy = (RotationQuaternion.j) / sin(RotationAngleInRadians / 2.0);
 RotationAxisUnitVector.Vz = (RotationQuaternion.k) / sin(RotationAngleInRadians / 2.0);

 /**** pow(cos(Angle), 2) + pow(sin(Angle), 2) = 1 ****/
 /**** sin(RotationAngleInRadians / 2.0) = sqrt(1.0 - pow(RotationQuaternion.a, 2)); ****/

 return RotationAxisUnitVector;
}
```

---

struct Quaternion CreatePointQuaternion(struct Vector Point)

```
{
 struct Quaternion PointQuaternion;

 PointQuaternion.a = 0;
 PointQuaternion.i = Point.Vx;
 PointQuaternion.j = Point.Vy;
 PointQuaternion.k = Point.Vz;

 return PointQuaternion;
}
```

---

struct Quaternion GetQuaternionConjugate(struct Quaternion Q)

```
{
 struct Quaternion QuaternionConjugate;

 QuaternionConjugate.a = Q.a;
 QuaternionConjugate.i = Q.i * -1;
 QuaternionConjugate.j = Q.j * -1;
 QuaternionConjugate.k = Q.k * -1;
```

```
 return QuaternionConjugate;
}
```

---

struct Quaternion GetQuaternionInverse(struct Quaternion Q)

```
{
 /**** Quaternion Inverse = Quaternion Reciprocal ****/
 /**** Inverse(Q) = 1 / Q ****/

 struct Quaternion QuaternionConjugate;
 double QuaternionMagnitudeSquared = 0.0;
 struct Quaternion QuaternionInverse;

 QuaternionConjugate       = GetQuaternionConjugate(Q);
 QuaternionMagnitudeSquared = GetQuaternionMagnitudeSquared(Q);

 if(QuaternionMagnitudeSquared == 0.0)
 {
  return CreateQuaternionByComponents(0.0, 0.0, 0.0, 0.0);
 }

 QuaternionInverse.a = QuaternionConjugate.a / QuaternionMagnitudeSquared;
 QuaternionInverse.i = QuaternionConjugate.i / QuaternionMagnitudeSquared;
 QuaternionInverse.j = QuaternionConjugate.j / QuaternionMagnitudeSquared;
 QuaternionInverse.k = QuaternionConjugate.k / QuaternionMagnitudeSquared;

 return QuaternionInverse;
}
```

---

struct Quaternion AddQuaternions(struct Quaternion Q1, struct Quaternion Q2)

```
{
 struct Quaternion AdditionQuaternion;

 AdditionQuaternion.a = Q1.a + Q2.a;
 AdditionQuaternion.i = Q1.i + Q2.i;
 AdditionQuaternion.j = Q1.j + Q2.j;
 AdditionQuaternion.k = Q1.k + Q2.k;

 return AdditionQuaternion;
}
```

---

struct Quaternion SubtractQuaternions(struct Quaternion Q1, struct Quaternion Q2)

```
{
 struct Quaternion SubtractionQuaternion;

 SubtractionQuaternion.a = Q1.a - Q2.a;
 SubtractionQuaternion.i = Q1.i - Q2.i;
 SubtractionQuaternion.j = Q1.j - Q2.j;
 SubtractionQuaternion.k = Q1.k - Q2.k;

 return SubtractionQuaternion;
}
```

---

struct Quaternion MultiplyQuaternions(struct Quaternion Q1, struct Quaternion Q2)

```
{
 struct Quaternion MultiplicationQuaternion;

 MultiplicationQuaternion.a = (Q1.a * Q2.a) + (Q1.i * Q2.i * -1) + (Q1.j * Q2.j * -1) + (Q1.k *
Q2.k * -1);
 MultiplicationQuaternion.i = (Q1.a * Q2.i) + (Q1.i * Q2.a)      + (Q1.j * Q2.k)      + (Q1.k * Q2.j
* -1);
 MultiplicationQuaternion.j = (Q1.a * Q2.j) + (Q1.j * Q2.a)      + (Q1.k * Q2.i)      + (Q1.i * Q2.k
* -1);
 MultiplicationQuaternion.k = (Q1.a * Q2.k) + (Q1.k * Q2.a)      + (Q1.i * Q2.j)      + (Q1.j *
Q2.i * -1);

 return MultiplicationQuaternion;
}
```

---

struct Quaternion MultiplyQuaternions_2(struct Quaternion Q1, struct Quaternion Q2)

```
{
 double Q1R       = 0.0;
 double Q2R       = 0.0;
 double ScalarPart = 0.0;
 struct Vector Q1V;
 struct Vector Q2V;
 struct Vector VectorPart;
 struct Quaternion MultiplicationQuaternion;

 Q1R = GetQuaternionScalarPart(Q1);
 Q2R = GetQuaternionScalarPart(Q2);
 Q1V = GetQuaternionVectorPart(Q1);
 Q2V = GetQuaternionVectorPart(Q2);

 ScalarPart             = (Q1R * Q2R) - GetVectorsDotProduct(Q1V, Q2V);
```

```
 VectorPart             = AddVectors(MultiplyVectorByScalar(Q1V, Q2R),
MultiplyVectorByScalar(Q2V, Q1R));
 VectorPart             = AddVectors(VectorPart, GetVectorsCrossProduct(Q1V, Q2V));
 MultiplicationQuaternion = CreateQuaternionByScalarAndVector(ScalarPart, VectorPart);

 return MultiplicationQuaternion;
}
```

```
struct Quaternion DivideQuaternions(struct Quaternion Q1, struct Quaternion Q2)
{
 /**** Q1 / Q2 = Q1 * Inverse(Q2) ****/

 struct Quaternion Q2Inverse;
 struct Quaternion DivisionQuaternion;

 Q2Inverse          = GetQuaternionInverse(Q2);
 DivisionQuaternion = MultiplyQuaternions(Q1, Q2Inverse);

 return DivisionQuaternion;
}
```

```
struct Quaternion AddScalarToQuaternion(struct Quaternion Q, double Scalar)
{
 struct Quaternion AddScalarQuaternion;

 AddScalarQuaternion.a = Q.a + Scalar;
 AddScalarQuaternion.i = Q.i;
 AddScalarQuaternion.j = Q.j;
 AddScalarQuaternion.k = Q.k;

 return AddScalarQuaternion;
}
```

```
struct Quaternion AddVectorToQuaternion(struct Quaternion Q, struct Vector V)
{
 struct Quaternion AddVectorQuaternion;

 AddVectorQuaternion.a = Q.a;
 AddVectorQuaternion.i = Q.i + V.Vx;
 AddVectorQuaternion.j = Q.j + V.Vy;
 AddVectorQuaternion.k = Q.k + V.Vz;
```

```
 return AddVectorQuaternion;
}
```

---

struct Quaternion SubtractScalarFromQuaternion(struct Quaternion Q, double Scalar)

```
{
 struct Quaternion SubtractScalarQuaternion;

 SubtractScalarQuaternion.a = Q.a - Scalar;
 SubtractScalarQuaternion.i = Q.i;
 SubtractScalarQuaternion.j = Q.j;
 SubtractScalarQuaternion.k = Q.k;

 return SubtractScalarQuaternion;
}
```

---

struct Quaternion SubtractVectorFromQuaternion(struct Quaternion Q, struct Vector V)

```
{
 struct Quaternion SubtractVectorQuaternion;

 SubtractVectorQuaternion.a = Q.a;
 SubtractVectorQuaternion.i = Q.i - V.Vx;
 SubtractVectorQuaternion.j = Q.j - V.Vy;
 SubtractVectorQuaternion.k = Q.k - V.Vz;

 return SubtractVectorQuaternion;
}
```

---

struct Quaternion MultiplyQuaternionByScalar(struct Quaternion Q, double ScalarValue)

```
{
 struct Quaternion MultiplyByScalarQuaternion;

 MultiplyByScalarQuaternion.a = Q.a * ScalarValue;
 MultiplyByScalarQuaternion.i = Q.i * ScalarValue;
 MultiplyByScalarQuaternion.j = Q.j * ScalarValue;
 MultiplyByScalarQuaternion.k = Q.k * ScalarValue;

 return MultiplyByScalarQuaternion;
}
```

---

struct Quaternion DivideQuaternionByScalar(struct Quaternion Q, double ScalarValue)

```
{
 struct Quaternion DivideByScalarQuaternion;

 DivideByScalarQuaternion.a = Q.a / ScalarValue;
 DivideByScalarQuaternion.i = Q.i / ScalarValue;
 DivideByScalarQuaternion.j = Q.j / ScalarValue;
 DivideByScalarQuaternion.k = Q.k / ScalarValue;

 return DivideByScalarQuaternion;
}
```

```
struct Quaternion NegateQuaternion(struct Quaternion Q)
{
 struct Quaternion NegatedQuaternion;

 NegatedQuaternion.a = Q.a * -1;
 NegatedQuaternion.i = Q.i * -1;
 NegatedQuaternion.j = Q.j * -1;
 NegatedQuaternion.k = Q.k * -1;

 return NegatedQuaternion;
}
```

```
struct Quaternion CopyQuaternion(struct Quaternion Q)
{
 struct Quaternion CopiedQuaternion;

 CopiedQuaternion.a = Q.a;
 CopiedQuaternion.i = Q.i;
 CopiedQuaternion.j = Q.j;
 CopiedQuaternion.k = Q.k;

 return CopiedQuaternion;
}
```

```
double** ConvertQuaternionToMatrix(struct Quaternion Q)
{
 double **QuaternionMatrix = NULL;
 const int Rows          = 4;
 const int Cols          = 4;
 int i                   = 0;
```

```
 QuaternionMatrix = (double**) calloc(Rows + 1, sizeof(double*));
 for(i = 0; i < Rows; i++)
 {
  *(QuaternionMatrix  + i) = (double*) calloc(Cols + 1, sizeof(double));
 }

 *(*(QuaternionMatrix + 0) + 0) = 1.0 - (2.0 * pow(Q.j, 2)) - (2.0 * pow(Q.k, 2));
 *(*(QuaternionMatrix + 0) + 1) = (2.0 * Q.i * Q.j) - (2.0 * Q.a * Q.k);
 *(*(QuaternionMatrix + 0) + 2) = (2.0 * Q.i * Q.k) + (2.0 * Q.a * Q.j);
 *(*(QuaternionMatrix + 0) + 3) = 0;

 *(*(QuaternionMatrix + 1) + 0) = (2.0 * Q.i * Q.j) + (2.0 * Q.a * Q.k);
 *(*(QuaternionMatrix + 1) + 1) = 1.0 - (2.0 * pow(Q.i, 2)) - (2.0 * pow(Q.k, 2));
 *(*(QuaternionMatrix + 1) + 2) = (2.0 * Q.j * Q.k) - (2.0 * Q.a * Q.i);
 *(*(QuaternionMatrix + 1) + 3) = 0;

 *(*(QuaternionMatrix + 2) + 0) = (2.0 * Q.i * Q.k) - (2.0 * Q.a * Q.j);
 *(*(QuaternionMatrix + 2) + 1) = (2.0 * Q.j * Q.k) + (2.0 * Q.a * Q.i);
 *(*(QuaternionMatrix + 2) + 2) = 1.0 - (2.0 * pow(Q.i, 2)) - (2.0 * pow(Q.j, 2));
 *(*(QuaternionMatrix + 2) + 3) = 0;

 *(*(QuaternionMatrix + 3) + 0) = 0;
 *(*(QuaternionMatrix + 3) + 1) = 0;
 *(*(QuaternionMatrix + 3) + 2) = 0;
 *(*(QuaternionMatrix + 3) + 3) = 1;

 return QuaternionMatrix;
}
```

---

struct Quaternion LerpQuaternion(struct Quaternion Q1, struct Quaternion Q2, double Percent)

```
{
 /****
     Lerp stands for Linear Interpolation. Interpolates between two Quaternions linearly
     "Percent" indicates how far to interpolate between the two Quaternions
     Lerp Quaternion = Q1 + (Q2 - Q1) * Percent = Q1 + (Q2 * Percent) - (Q1 * Percent) =
Q1(1.0 - Percent) + (Q2 * Percent)
 ****/

 struct Quaternion LerpQ;

 // LerpQ = AddQuaternions(Q1, MultiplyQuaternionByScalar(SubtractQuaternions(Q2, Q1),
Percent));
   LerpQ = AddQuaternions(MultiplyQuaternionByScalar(Q1, (1.0 - Percent)),
MultiplyQuaternionByScalar(Q2, Percent));
```

```
 return LerpQ;
}
```

---

struct Quaternion SlerpQuaternion(struct Quaternion Q1, struct Quaternion Q2, double Percent)

```
{
 /****
     Slerp stands for Spherical Linear Interpolation. Interpolates between two Quaternions using spherical linear interpolation
     "Percent" indicates how far to interpolate between the two Quaternions
     Slerp Quaternion = Q1 * (sin((1.0 - Percent) * HalfAngleInRadians) / sin(HalfAngleInRadians))
                 + Q2 * (sin(Percent * HalfAngleInRadians) / sin(HalfAngleInRadians))
 ****/

 double AngleInDegrees     = 0.0; /**** Angle Between the two Quaternions ****/
 double HalfAngleInDegrees = 0.0;
 double HalfAngleInRadians = 0.0;
 double Value1             = 0.0;
 double Value2             = 0.0;
 struct Quaternion SlerpQ;

 AngleInDegrees     = GetAngleBetweenTwoQuaternions(Q1, Q2);
 HalfAngleInDegrees = AngleInDegrees / 2.0;
 HalfAngleInRadians = HalfAngleInDegrees * DegreesToRadians;

 Value1 = sin((1.0 - Percent) * HalfAngleInRadians) / sin(HalfAngleInRadians);
 Value2 = sin(Percent * HalfAngleInRadians) / sin(HalfAngleInRadians);

 SlerpQ = AddQuaternions(MultiplyQuaternionByScalar(Q1, Value1), MultiplyQuaternionByScalar(Q2, Value2));

 return SlerpQ;
}
```

---

struct Quaternion SlerpQuaternion_2(struct Quaternion Q1, struct Quaternion Q2, double Percent)

```
{
 struct Quaternion Q;
 struct Quaternion QPercent;
 struct Quaternion SlerpQ;
 double RotationAngleInDegrees = 0.0;
```

```
 struct Vector RotationAxisUnitVector;

 Q = MultiplyQuaternions(Q2, GetQuaternionInverse(Q1));
 /****
     PrintQuaternion("Q", Q, 6);
     PrintQuaternion("Q * Q1", MultiplyQuaternions(Q, Q1), 6); // Q * Q1 = Q2
     PrintQuaternion("Q2", Q2, 6);
 ****/

 RotationAxisUnitVector = GetAxisAndAngleFromRotationQuaternion(Q,
&RotationAngleInDegrees);
 QPercent              = CreateRotationQuaternion(RotationAxisUnitVector, Percent *
RotationAngleInDegrees);
 SlerpQ                = MultiplyQuaternions(QPercent, Q1);

 return SlerpQ;
}
```

APPENDIX 2 – GPU CODE

```
__global__ void AddArrays(int *d_Arr1, int *d_Arr2, int *d_sum, unsigned int Length)
{
 unsigned int Index = (blockDim.x * blockIdx.x) + threadIdx.x; /**** If LENGTH = 65,
then Index will be from 0 to 95 ****/

 if(Index < Length)
 {
  d_sum[Index] = d_Arr1[Index] + d_Arr2[Index];
 }
}
```

Code 1. The AddArrays Kernel

```
int main(int argc, char **argv)
{
 const unsigned int LENGTH = 64U;
 const unsigned int SIZE   = LENGTH * sizeof(int);
 unsigned int i            = 0U;
 dim3 BlockDim(32, 1, 1); /**** x = 32, y = 1, z = 1 ****/
 dim3 GridDim(ceil((float) LENGTH / BlockDim.x), 1, 1);

 (void) GetMemoryInfo("Before Allocation");

 int *h_Arr1 = NULL;
 int *h_Arr2 = NULL;
 int *h_sum  = NULL;

 h_Arr1 = (int*) malloc(SIZE);
 h_Arr2 = (int*) malloc(SIZE);
 h_sum  = (int*) malloc(SIZE);
 memset(h_Arr1, 0, SIZE);
 memset(h_Arr2, 0, SIZE);
 memset(h_sum, 0, SIZE);

 for(i = 0; i < LENGTH; i++)
 {
  h_Arr1[i] = i;
  h_Arr2[i] = i * 2;
 }

 (void) PrintArray("h_Arr1", h_Arr1, LENGTH);
 (void) PrintArray("h_Arr2", h_Arr2, LENGTH);

 int *d_Arr1 = NULL;
 int *d_Arr2 = NULL;
 int *d_sum  = NULL;

 cudaMalloc((void**) &d_Arr1, SIZE);
 cudaMalloc((void**) &d_Arr2, SIZE);
 cudaMalloc((void**) &d_sum, SIZE);
 cudaMemset(d_Arr1, 0, SIZE);
 cudaMemset(d_Arr2, 0, SIZE);
 cudaMemset(d_sum, 0, SIZE);

 (void) GetMemoryInfo("After Allocation");

 cudaMemcpy(d_Arr1, h_Arr1, SIZE, cudaMemcpyHostToDevice);
 cudaMemcpy(d_Arr2, h_Arr2, SIZE, cudaMemcpyHostToDevice);

 (void) AddArrays<<<GridDim, BlockDim>>>(d_Arr1, d_Arr2, d_sum, LENGTH);
 // (void) AddArrays<<<ceil((float) LENGTH / 32.00f), 32>>>(d_Arr1, d_Arr2, d_sum, LENGTH);

 cudaMemcpy(h_sum, d_sum, SIZE, cudaMemcpyDeviceToHost);
 (void) PrintArray("h_sum", h_sum, LENGTH);
 cudaFree(d_Arr1);
 cudaFree(d_Arr2);
 cudaFree(d_sum);
 free(h_Arr1);
 free(h_Arr2);
 free(h_sum);
(void) GetMemoryInfo("After Deallocation");

 return 0;
}
```

Code 2. Calling the AddArrays Kernel

```
__host__ void PrintDeviceProperties(cudaDeviceProp DeviceProp)
{
 printf("Name                                   : %s\n", DeviceProp.name);
 printf("Major revision number                  : %d\n", DeviceProp.major);
 printf("Minor revision number                  : %d\n", DeviceProp.minor);
 printf("Maximum memory pitch                   : %u\n", DeviceProp.memPitch);
 printf("Clock rate                             : %d\n", DeviceProp.clockRate);
 printf("Texture alignment                      : %u\n", DeviceProp.textureAlignment);
 printf("Concurrent copy and execution          : %d\n",
DeviceProp.deviceOverlap);
 printf("Concurrent copy and execution          : %s\n", (DeviceProp.deviceOverlap
? "Yes" : "No"));
 printf("Kernel execution timeout enabled       : %s\n",
(DeviceProp.kernelExecTimeoutEnabled ? "Yes" : "No"));
 printf("Number of multiprocessors              : %d\n",
DeviceProp.multiProcessorCount);
 printf("The maximum number of threads per multiprocessor  : %d\n",
DeviceProp.maxThreadsPerMultiProcessor);
 printf("The maximum number of threads per block           : %d\n",
DeviceProp.maxThreadsPerBlock);
 printf("The maximum sizes of each dimension of a block (x, y, z) : %d, %d, %d\n",
DeviceProp.maxThreadsDim[0], DeviceProp.maxThreadsDim[1],
DeviceProp.maxThreadsDim[2]);
 printf("The maximum sizes of each dimension of a grid (x, y, z)  : %d, %d, %d\n",
DeviceProp.maxGridSize[0], DeviceProp.maxGridSize[1], DeviceProp.maxGridSize[2]);
 printf("The total number of registers available per block  : %d\n",
DeviceProp.regsPerBlock);
 printf("The total amount of shared memory per block (Bytes)      : %u\n",
DeviceProp.sharedMemPerBlock);
 printf("The total amount of constant memory (Bytes)              : %u\n",
DeviceProp.totalConstMem);
 printf("The total amount of global memory (Bytes), (Gigabytes)   : %llu, %f\n", (unsigned
long long int) DeviceProp.totalGlobalMem, (float) DeviceProp.totalGlobalMem /
1073741824.00f);
 printf("Warp size (Threads)                              : %d\n", DeviceProp.warpSize);  /****
We should set blockDim as a multiple of 32 or whatever the warp size happens to be. ****/
 printf("\n");
}
```

Code 4. List the GPU Properties

```
int main(int argc, char **argv)
{
 unsigned int i  = 0U;
 int DeviceCount = 0;
 cudaDeviceProp DeviceProp;

 cudaGetDeviceCount(&DeviceCount);
 printf("\nDevice Count = %d\n", DeviceCount);

 cudaGetDeviceProperties(&DeviceProp, 0);
 printf("Device 0 Name is %s\n\n", DeviceProp.name);

 for(i = 0; i < DeviceCount; i++)
 {
  printf("Device Number %d:\n", i);
  cudaGetDeviceProperties(&DeviceProp, i);
  PrintDeviceProperties(DeviceProp);
 }

 return 0;
}
```

Code 4. Calling the PrintDeviceProperties() Function

APPENDIX 3 – SpMV FORMATS AND KERNELS FOR THE CPU

```
void CreateFormat(double **Mat, double *Value, unsigned int *Column,
unsigned int *RowPtr, unsigned char *Flag, unsigned int *AllNonZeros)
{
 unsigned int i      = 0U, j = 0U;
 unsigned int Index  = 0U;
 unsigned char GotIt = 0U;

 for(i = 0U; i < ROWS; i++)
 {
  GotIt = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(Mat[i][j] != 0.00)
   {
    Value[Index]  = Mat[i][j];
    Column[Index] = j;

    if(GotIt == 0U)
    {
     GotIt       = 1U;
     RowPtr[i]   = Index;
     Flag[Index] = 1U;
    }
    Index++;
   }
  }
 }
 RowPtr[i] = *AllNonZeros;
}
```

The CSR Format

```
void SpMV_CSR(double *Value, unsigned int *Column, double *Vector,
unsigned int *RowPtr, double *Result)
{
 unsigned int i     = 0U, j   = 0U;
 unsigned int Start = 0U, End = 0U;
 double Temp        = 0.00;

 for(i = 0U; i < ROWS; i++)
 {
  Start           = RowPtr[i];
  End             = RowPtr[i + 1];
  for(j = Start; j < End; j++)
  {
   Temp += Value[j] * Vector[Column[j]];
  }
  Result[i] = Temp;
  Temp      = 0.00;
 }
}
```

The CSR Kernel

```
void CreateFormat(double **Mat, double **NonZerosEntries, int **Column)
{
 unsigned int i = 0U, j = 0U;
 unsigned int r = 0U, c = 0U;

 for(i = 0U; i < ROWS; i++)
 {
  c = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(Mat[i][j] != 0.00)
   {
    NonZerosEntries[r][c] = Mat[i][j];
    Column[r][c]          = j;
    c++;
   }
  }
  r++;
 }
}
```

The ELLPACK Format

```
void SpMV_ELLPACK(double **NonZerosEntries, int **Column, double
*Vector, double *Result, unsigned int *MaxNonZeros)
{
 unsigned int r1 = 0U, r2 = 0U;
 double Temp     = 0.00;

 for(r1 = 0U; r1 < ROWS; r1++)
 {
  for(r2 = 0U; r2 < *MaxNonZeros; r2++)
  {
   if(Column[r1][r2] == -1)
   {
    break;
   }
   Temp = Temp + (NonZerosEntries[r1][r2] * Vector[Column[r1][r2]]);
  }
  Result[r1] = Temp;
  Temp       = 0.00;
 }
}
```

The ELLPACK Kernel

```
void SpMV_ELLPACK_R(double **NonZerosEntries, int **Column, double
*Vector, unsigned int *NonZerosCount, double *Result)
{
 unsigned int r1  = 0U, r2 = 0U;
 unsigned int End = 0U;
 double Temp      = 0.00;

 for(r1 = 0U; r1 < ROWS; r1++)
 {
  End = NonZerosCount[r1];
  for(r2 = 0U; r2 < End; r2++)
  {
   Temp = Temp + (NonZerosEntries[r1][r2] * Vector[Column[r1][r2]]);
  }
  Result[r1] = Temp;
  Temp       = 0.00;
 }
}
```

The ELLPACK-R Kernel

```
void CreateFormat(double **Mat, double **NonZerosEntries, int **Column,
unsigned int Rows)
{
 unsigned int i = 0U, j = 0U;
 unsigned int r = 0U, c = 0U;

 for(i = 0U; i < Rows; i++)
 {
  c = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(Mat[i][j] != 0.00)
   {
    NonZerosEntries[r][c] = Mat[i][j];
    Column[r][c]          = j;
    c++;
   }
  }
```

The Sliced ELLPACK Format

```
void SpMV_SlicedELLPACK(double **NonZerosEntries, int **Column,
double *Vector, double *Result, unsigned int Rows, unsigned int *Cols)
{
 unsigned int r1  = 0U, r2 = 0U;
 double Temp      = 0.00;

 for(r1 = 0U; r1 < Rows; r1++)
 {
  for(r2 = 0U; r2 < *Cols; r2++)
  {
   if(Column[r1][r2] == -1)
   {
    break;
   }
   Temp = Temp + (NonZerosEntries[r1][r2] * Vector[Column[r1][r2]]);
  }
  Result[r1] = Temp;
  Temp       = 0.00;
 }
}
```

The Sliced ELLPACK Kernel

```
void SpMV_SlicedELLPACK_R(double **NonZerosEntries, int **Column,
double *Vector, unsigned int *NonZerosCount, double *Result, unsigned int
Rows)
{
 unsigned int r1  = 0U, r2 = 0U;
 unsigned int End = 0U;
 double Temp      = 0.00;

 for(r1 = 0U; r1 < Rows; r1++)
 {
  End = NonZerosCount[r1];
  for(r2 = 0U; r2 < End; r2++)
  {
   Temp = Temp + (NonZerosEntries[r1][r2] * Vector[Column[r1][r2]]);
  }
  Result[r1] = Temp;
  Temp       = 0.00;
 }
}
```

The Sliced ELLPACK-R Kernel

```
void CreateFormat(double **Mat, double **NonZerosEntries, int
**ELLPACK_Column, double *Value, unsigned int *CSR_Column, unsigned
int *AllNonZerosCount, unsigned int *StartArr, unsigned int *EndArr)
{
 unsigned int i = 0U, j = 0U;
 unsigned int r = 0U, c = 0U;
 unsigned int Start     = 0U, End = 0U;
 unsigned int Index     = 0U;
 unsigned char GotIt    = 0U;

 for(i = 0U; i < ROWS; i++)
 {
  c     = 0U;
  GotIt = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(Mat[i][j] != 0.00)
   {
    if(c < (unsigned int) BOUNDARY) /**** ELLPACK Format (0 to
(BOUNDARY - 1)) ****/
    {
     NonZerosEntries[r][c] = Mat[i][j];
     ELLPACK_Column[r][c]  = j;
     c++;
    }
    else            /**** CSR Format ****/
    {
     Value[Index]      = Mat[i][j];
     CSR_Column[Index] = j;
     if(GotIt == 0U) { GotIt = 1U; Start = Index;}
     Index++;
    }
   }
  }
  r++;
  if(AllNonZerosCount[i] > (unsigned int) BOUNDARY) /**** CSR Format
****/
  {
   End        = Index;
   StartArr[i] = Start;
   EndArr[i]   = End;
  }
 }
}
```

The Proposed Format

```
void SpMV_Hybrid_ELLPACKandCSR(double **NonZerosEntries, int
**ELLPACK_Column, double *Value, unsigned int *CSR_Column, unsigned
int *StartArr, unsigned int *EndArr, unsigned int *AllNonZerosCount, double
*Vector, double *Result)
{
 unsigned int r1    = 0U, r2  = 0U, j = 0U;
 unsigned int Start = 0U, End = 0U;
 double Temp        = 0.00;

 /**** ELLPACK Format ****/
 for(r1 = 0U; r1 < ROWS; r1++)
 {
  for(r2 = 0U; r2 < BOUNDARY; r2++)
  {
   if(ELLPACK_Column[r1][r2] == -1)
   {
    break;
   }
   Temp += (NonZerosEntries[r1][r2] * Vector[ELLPACK_Column[r1][r2]]);
  }

  /**** CSR Format ****/
  if(AllNonZerosCount[r1] > (unsigned int) BOUNDARY)
  {
   Start = StartArr[r1];
   End   = EndArr[r1];
   for(j = Start; j < End; j++)
   {
    Temp += (Value[j] * Vector[CSR_Column[j]]);
   }
  }

  Result[r1] = Temp;
  Temp       = 0.00;
 }
}
```

The Proposed Kernel

```
void CreateFormat(double **Mat, double **NonZerosEntries, int
**ELLPACK_Column, double *Value, unsigned int *CSR_Column, unsigned
int *AllNonZerosCount, unsigned int *StartArr, unsigned int *EndArr,
unsigned int Rows, unsigned int Boundary)
{
 unsigned int i = 0U, j = 0U;
 unsigned int r = 0U, c = 0U;
 unsigned int Start     = 0U, End = 0U;
 unsigned int Index     = 0U;
 unsigned char GotIt    = 0U;

 for(i = 0U; i < Rows; i++)
 {
  c     = 0U;
  GotIt = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(Mat[i][j] != 0.00)
   {
    if(c < Boundary) /**** ELLPACK Format (0 to (Boundary - 1)) ****/
    {
     NonZerosEntries[r][c] = Mat[i][j];
     ELLPACK_Column[r][c]  = j;
     c++;
    }
    else            /**** CSR Format ****/
    {
     Value[Index]      = Mat[i][j];
     CSR_Column[Index] = j;
     if(GotIt == 0U) { GotIt = 1U; Start = Index;}
     Index++;
    }
   }
  }
  r++;
  if(AllNonZerosCount[i] > Boundary) /**** CSR Format ****/
  {
   End        = Index;
   StartArr[i] = Start;
   EndArr[i]   = End;
  }
 }
}
```

A Variation of the Proposed Model (Format)

```
void SpMV_Hybrid_SlicedELLPACKandCSR(double **NonZerosEntries, int
**ELLPACK_Column, double *Value, unsigned int *CSR_Column, unsigned
int *StartArr, unsigned int *EndArr, unsigned int *AllNonZerosCount, double
*Vector, double *Result, unsigned int Rows, unsigned int Boundary)
{
 unsigned int r1    = 0U, r2  = 0U, j = 0U;
 unsigned int Start = 0U, End = 0U;
 double Temp        = 0.00;

 /**** ELLPACK Format ****/
 for(r1 = 0U; r1 < Rows; r1++)
 {
  for(r2 = 0U; r2 < Boundary; r2++)
  {
   if(ELLPACK_Column[r1][r2] == -1)
   {
    break;
   }
   Temp += (NonZerosEntries[r1][r2] * Vector[ELLPACK_Column[r1][r2]]);
  }

  /**** CSR Format ****/
  if(AllNonZerosCount[r1] > Boundary)
  {
   Start = StartArr[r1];
   End   = EndArr[r1];
   for(j = Start; j < End; j++)
   {
    Temp += (Value[j] * Vector[CSR_Column[j]]);
   }
  }
```

A Variation of the Proposed Model (Kernel)

APPENDIX 4 – SpMV FORMATS AND KERNELS FOR THE GPU

```
__host__ void CreateFormat(double *h_Mat, double *h_Value, unsigned int
*h_Column, unsigned int *h_RowPtr, unsigned char *h_Flag, unsigned int
*AllNonZeros)
{
 unsigned int i      = 0U, j = 0U;
 unsigned int Index  = 0U;
 unsigned char GotIt = 0U;

 for(i = 0U; i < ROWS; i++)
 {
  GotIt = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(h_Mat[(i * COLS) + j] != 0.00)
   {
    h_Value[Index]  = h_Mat[(i * COLS) + j];
    h_Column[Index] = j;

    if(GotIt == 0U)
    {
     GotIt = 1U;
     h_RowPtr[i]   = Index;
     h_Flag[Index] = 1U;
    }
    Index++;
   }
  }
 }
 h_RowPtr[i] = *AllNonZeros;
}
```

The CSR Format

```
__global__ void SpMV_CSR(double *d_Value, unsigned int *d_Column,
double *d_Vector, unsigned int *d_RowPtr, double *d_Result)
{
 unsigned int Index = (blockDim.x * blockIdx.x) + threadIdx.x;
 unsigned int j     = 0U;
 unsigned int Start = 0U, End = 0U;
 double Temp        = 0.00;

 if(Index < (unsigned int) ROWS)
 {
  Start               = d_RowPtr[Index];
  End                 = d_RowPtr[Index + 1];
  for(j = Start; j < End; j++)
  {
   Temp += (d_Value[j] * d_Vector[d_Column[j]]);
  }
  d_Result[Index] = Temp;
 }
}
```

The CSR Kernel

```
__host__ void CreateFormat(double *h_Mat, double *h_NonZerosEntries, int
*h_Column, unsigned int *MaxNonZeros)
{
 unsigned int i = 0U, j = 0U;
 unsigned int r = 0U, c = 0U;

 for(i = 0U; i < ROWS; i++)
 {
  c = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(h_Mat[(i * COLS) + j] != 0.00)
   {
    h_NonZerosEntries[(r * *MaxNonZeros) + c] = h_Mat[(i * COLS) + j];
    h_Column[(r * *MaxNonZeros) + c]          = j;
    c++;
   }
  }
  r++;
 }
}
```

The ELLPACK Format

```
__global__ void SpMV_ELLPACK_ThreadPerRow(double
*d_NonZerosEntries, int *d_Column, double *d_Vector, double *d_Result,
unsigned int MaxNonZeros)
{
 unsigned int Index = (blockDim.x * blockIdx.x) + threadIdx.x;
 unsigned int r2    = 0U;
 double Temp        = 0.00;

 if(Index < (unsigned int) ROWS)
 {
  for(r2 = 0U; r2 < MaxNonZeros; r2++)
  {
   if(d_Column[(Index * MaxNonZeros) + r2] == -1)
   {
    break;
   }
   Temp += (d_NonZerosEntries[(Index * MaxNonZeros) + r2] *
d_Vector[d_Column[(Index * MaxNonZeros) + r2]]);
  }
  d_Result[Index] = Temp;
 }
}
```

The ELLPACK Kernel (Thread per Row)

```
__global__ void SpMV_ELLPACK_R_ThreadPerRow(double
*d_NonZerosEntries, int *d_Column, double *d_Vector, unsigned int
*d_NonZerosCount, double *d_Result, unsigned int MaxNonZeros)
{
 unsigned int Index = (blockDim.x * blockIdx.x) + threadIdx.x;
 unsigned int r2    = 0U;
 unsigned int End   = 0U;
 double Temp        = 0.00;

 if(Index < (unsigned int) ROWS)
 {
  End = d_NonZerosCount[Index];
  for(r2 = 0U; r2 < End; r2++)
  {
   Temp += (d_NonZerosEntries[(Index * MaxNonZeros) + r2] *
d_Vector[d_Column[(Index * MaxNonZeros) + r2]]);
  }
  d_Result[Index] = Temp;
 }
}
```

The ELLPACK-R Kernel (Thread per Row)

```
__global__ void SpMV_ELLPACK_ThreadPerColumn(double
*d_NonZerosEntries, int *d_Column, double *d_Vector, double *d_Result,
unsigned int MaxNonZeros)
{
 unsigned int Row = (blockDim.y * blockIdx.y) + threadIdx.y;
 unsigned int Col = (blockDim.x * blockIdx.x) + threadIdx.x;

 if((Row < (unsigned int) ROWS) && (Col < MaxNonZeros))
 {
  if(d_Column[(Row * MaxNonZeros) + Col] != -1)
  {
   atomicAdd(&d_Result[Row], (d_NonZerosEntries[(Row * MaxNonZeros) +
Col] * d_Vector[d_Column[(Row * MaxNonZeros) + Col]]));
   // atomicAdd(&d_Result2[Row], (d_NonZerosEntries[(Row *
MaxNonZeros) + Col] * d_Vector[d_Column[(Row * MaxNonZeros) +
Col]]));
  }
 }
}
```

The ELLPACK Kernel (Thread per Column)

```
__host__ void CreateFormat(double *h_Mat, double *h_NonZerosEntries, int
*h_Column, unsigned int Rows, unsigned int *Cols)
{
 unsigned int i = 0U, j = 0U;
 unsigned int r = 0U, c = 0U;

 for(i = 0U; i < Rows; i++)
 {
  c = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(h_Mat[(i * COLS) + j] != 0.00)
   {
    h_NonZerosEntries[(r * *Cols) + c] = h_Mat[(i * COLS) + j];
    h_Column[(r * *Cols) + c]          = j;
    c++;
   }
  }
  r++;
 }
}
```

The Sliced ELLPACK Format

```
__global__ void SpMV_SlicedELLPACK_ThreadPerRow(double
*d_NonZerosEntries, int *d_Column, double *d_Vector, double *d_Result,
unsigned int Rows, unsigned int Cols)
{
 unsigned int Index = (blockDim.x * blockIdx.x) + threadIdx.x;
 unsigned int r2    = 0U;
 double Temp        = 0.00;

 if(Index < Rows)
 {
  for(r2 = 0U; r2 < Cols; r2++)
  {
   if(d_Column[(Index * Cols) + r2] == -1)
   {
    break;
   }
   Temp = Temp + (d_NonZerosEntries[(Index * Cols) + r2] *
d_Vector[d_Column[(Index * Cols) + r2]]);
  }
  d_Result[Index] = Temp;
 }
}
```

The Sliced ELLPACK Kernel (Thread per Row)

```
__global__ void SpMV_SlicedELLPACK_R_ThreadPerRow(double
*d_NonZerosEntries, int *d_Column, double *d_Vector, unsigned int
*d_NonZerosCount, double *d_Result, unsigned int Rows, unsigned int Cols)
{
 unsigned int Index = (blockDim.x * blockIdx.x) + threadIdx.x;
 unsigned int r2    = 0U;
 unsigned int End   = 0U;
 double Temp        = 0.00;

 if(Index < Rows)
 {
  End = d_NonZerosCount[Index];
  for(r2 = 0U; r2 < End; r2++)
  {
   Temp = Temp + (d_NonZerosEntries[(Index * Cols) + r2] *
d_Vector[d_Column[(Index * Cols) + r2]]);
  }
  d_Result[Index] = Temp;
 }
}
```

The Sliced ELLPACK-R Kernel (Thread per Row)

```
__global__ void SpMV_SlicedELLPACK_ThreadPerColumn(double
*d_NonZerosEntries, int *d_Column, double *d_Vector, double *d_Result,
unsigned int Rows, unsigned int Cols)
{
 unsigned int Row = (blockDim.y * blockIdx.y) + threadIdx.y;
 unsigned int Col = (blockDim.x * blockIdx.x) + threadIdx.x;

 if((Row < Rows) && (Col < Cols))
 {
  if(d_Column[(Row * Cols) + Col] != -1)
  {
   // atomicAdd(&d_Result[Row], (d_NonZerosEntries[(Row * Cols) + Col] *
d_Vector[d_Column[(Row * Cols) + Col]]));
  }
 }
}
```

The Sliced ELLPACK-R Kernel (Thread per Column)

```
__host__ void CreateFormat(double *h_Mat, double *h_NonZerosEntries, int
*h_ELLPACK_Column, double *h_Value, unsigned int *h_CSR_Column, unsigned int
*h_AllNonZerosCount, unsigned int *h_StartArr, unsigned int *h_EndArr)
{
 unsigned int i = 0U, j = 0U;
 unsigned int r = 0U, c = 0U;
 unsigned int Start     = 0U, End = 0U;
 unsigned int Index     = 0U;
 unsigned char GotIt    = 0U;

 for(i = 0U; i < ROWS; i++)
 {
  c     = 0U;
  GotIt = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(h_Mat[(i * COLS) + j] != 0.00)
   {
    if(c < BOUNDARY) /**** ELLPACK Format (0 to (BOUNDARY - 1)) ****/
    {
     h_NonZerosEntries[(r * BOUNDARY) + c] = h_Mat[(i * COLS) + j];
     h_ELLPACK_Column[(r * BOUNDARY) + c]  = j;
     c++;
    }
    else            /**** CSR Format ****/
    {
     h_Value[Index]      = h_Mat[(i * COLS) + j];
     h_CSR_Column[Index] = j;
     if(GotIt == 0U) { GotIt = 1U; Start = Index;}
     Index++;
    }
   }
  }
  r++;

  if(h_AllNonZerosCount[i] > (unsigned int) BOUNDARY) /**** CSR Format ****/
  {
   End          = Index;
   h_StartArr[i] = Start;
   h_EndArr[i]   = End;
  }
 }
}
```

The Proposed Format

```
unsigned int Thread_id = (blockDim.x * blockIdx.x) + threadIdx.x; // Global
thread index.
 unsigned int Warp_id   = Thread_id / 32;                    // Global warp index.
 unsigned int Lane      = Thread_id & (32 - 1);              // Thread index
within the warp.
 unsigned int Row       = Warp_id;                           // One warp per row.
 unsigned int r2        = 0U, j   = 0U;
 unsigned int Start     = 0U, End = 0U;
 unsigned int x         = 0U;
 int          y         = 0;

 __device__ __shared__ double SharedMem1[32];
 __device__ __shared__ double SharedMem2[32];

 memset(SharedMem1, 0.00, (32 * sizeof(double)));
 memset(SharedMem2, 0.00, (32 * sizeof(double)));

 if(Row < (unsigned int) ROWS) // one warp per row
 {
  /**** ELLPACK Format ****/

  SharedMem1[threadIdx.x] = 0.00; // Compute running sum per thread.

  for(r2 = 0U + Lane; r2 < (unsigned int) BOUNDARY; r2 += 32)
  {
   x = (Row * (unsigned int) BOUNDARY) + r2;
   y = d_ELLPACK_Column[x];
   // if(y == -1)
   // {
   //  break;
   // }
   // Temp += (d_NonZerosEntries[x] * d_Vector[y]);
   SharedMem1[threadIdx.x] += (d_NonZerosEntries[x] * d_Vector[y]);
  }

  // Parallel reduction in shared memory (warp-level reductions).
  if(Lane < 16) { atomicAdd(&SharedMem1[threadIdx.x],
SharedMem1[threadIdx.x + 16]); }
  if(Lane < 8)  { atomicAdd(&SharedMem1[threadIdx.x],
SharedMem1[threadIdx.x + 8]);  }
  if(Lane < 4)  { atomicAdd(&SharedMem1[threadIdx.x],
SharedMem1[threadIdx.x + 4]);  }
  if(Lane < 2)  { atomicAdd(&SharedMem1[threadIdx.x],
SharedMem1[threadIdx.x + 2]);  }
  if(Lane < 1)  { atomicAdd(&SharedMem1[threadIdx.x],
SharedMem1[threadIdx.x + 1]);  }
```

```
// if(Lane == 0)
  // {
  //  d_Result[Row] += SharedMem1[threadIdx.x];
  // }

  /**** CSR Format ****/
  // if(d_AllNonZerosCount[Row] > (unsigned int) BOUNDARY)
  {
   Start = d_StartArr[Row];
   End   = d_EndArr[Row];

   SharedMem2[threadIdx.x] = 0.00; // Compute running sum per thread.

   for(j = Start + Lane; j < End; j += 32)
   {
    SharedMem2[threadIdx.x] += d_Value[j] * d_Vector[d_CSR_Column[j]];
   }

   // Parallel reduction in shared memory (warp-level reductions).
   if(Lane < 16) { atomicAdd(&SharedMem2[threadIdx.x],
SharedMem2[threadIdx.x + 16]); }
   if(Lane < 8)  { atomicAdd(&SharedMem2[threadIdx.x],
SharedMem2[threadIdx.x + 8]);  }
   if(Lane < 4)  { atomicAdd(&SharedMem2[threadIdx.x],
SharedMem2[threadIdx.x + 4]);  }
   if(Lane < 2)  { atomicAdd(&SharedMem2[threadIdx.x],
SharedMem2[threadIdx.x + 2]);  }
   if(Lane < 1)  { atomicAdd(&SharedMem2[threadIdx.x],
SharedMem2[threadIdx.x + 1]);  }

   if(Lane == 0) // The first thread writes the result
   {
    d_Result[Row] += (SharedMem1[threadIdx.x] +
SharedMem2[threadIdx.x]);
   }
  }
 }
}
```

The Proposed Kernel

```
__host__ void CreateFormat(double *h_Mat, double *h_NonZerosEntries, int
*h_ELLPACK_Column, double *h_Value, unsigned int *h_CSR_Column, unsigned
int *h_AllNonZerosCount, unsigned int *h_StartArr, unsigned int *h_EndArr,
unsigned int Rows, unsigned int Boundary)
{
 unsigned int i = 0U, j = 0U;
 unsigned int r = 0U, c = 0U;
 unsigned int Start     = 0U, End = 0U;
 unsigned int Index     = 0U;
 unsigned char GotIt    = 0U;

 for(i = 0U; i < Rows; i++)
 {
  c     = 0U;
  GotIt = 0U;
  for(j = 0U; j < COLS; j++)
  {
   if(h_Mat[(i * COLS) + j] != 0.00)
   {
    if(c < Boundary) /**** ELLPACK Format (0 to (Boundary - 1)) ****/
    {
     h_NonZerosEntries[(r * Boundary) + c] = h_Mat[(i * COLS) + j];
     h_ELLPACK_Column[(r * Boundary) + c]  = j;
     c++;
    }
    else            /**** CSR Format ****/
    {
     h_Value[Index]      = h_Mat[(i * COLS) + j];
     h_CSR_Column[Index] = j;
     if(GotIt == 0U) { GotIt = 1U; Start = Index;}
     Index++;
    }
   }
  }
  r++;
  if(h_AllNonZerosCount[i] > Boundary) /**** CSR Format ****/
  {
   End          = Index;
   h_StartArr[i] = Start;
   h_EndArr[i]   = End;
  }
 }
}
```

A Variation of the Proposed Model (Format)

```
__global__ void SpMV_Hybrid_SlicedELLPACKandCSR(double
*d_NonZerosEntries, int *d_ELLPACK_Column, double *d_Value, unsigned
int *d_CSR_Column, unsigned int *d_StartArr, unsigned int *d_EndArr,
unsigned int *d_AllNonZerosCount, double *d_Vector, double *d_Result,
unsigned int Rows, unsigned int Boundary)
{
 unsigned int Index = (blockDim.x * blockIdx.x) + threadIdx.x;
 unsigned int r2    = 0U, j   = 0U;
 unsigned int Start = 0U, End = 0U;
 double Temp        = 0.00;

 if(Index < Rows)
 {
  /**** ELLPACK Format ****/
  for(r2 = 0U; r2 < Boundary; r2++)
  {
   if(d_ELLPACK_Column[(Index * Boundary) + r2] == -1)
   {
    break;
   }
   Temp += (d_NonZerosEntries[(Index * Boundary) + r2] *
d_Vector[d_ELLPACK_Column[(Index * Boundary) + r2]]);
  }

  /**** CSR Format ****/
  if(d_AllNonZerosCount[Index] > Boundary)
  {
   Start = d_StartArr[Index];
   End   = d_EndArr[Index];
   for(j = Start; j < End; j++)
   {
    Temp += (d_Value[j] * d_Vector[d_CSR_Column[j]]);
   }
  }

  d_Result[Index] = Temp;
 }
}
```

A Variation of the Proposed Model (Kernel)

REFERENCES

[1] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," *Nvidia Tech. Rep.*, pp. 1–32, 2008.

[2] D. A, L. A, and M. M, "A memory efficient and fast sparse matrix vector product on a GPU," *Electromagnetics*, vol. 116, no. March, pp. 49–63, 2011.

[3] D. Guo, W. Gropp, and L. N. Olson, "A hybrid format for better performance of sparse matrix-vector multiplication on a GPU," *Artic. Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 103–120, 2016.

[4] R. B. Muthu Manikandan Baskaran and K. Yelick, "Optimizing Sparse Matrix-Vector Multiplication on GPUs," *Ninth SIAM Conf. Parallel Process. Sci. Comput.*, no. RC24704, 2008.

[5] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," *Proc. Conf. High Perform. Comput. Networking, Storage Anal. - SC '09*, no. 1, p. 1, 2009.

[6] I. Reguly and M. Giles, "Efficient sparse matrix-vector multiplication on cache-based GPUs," *2012 Innov. Parallel Comput. InPar 2012*, 2012.

[7] M. Maggioni and T. Berger-Wolf, "AdELL: An adaptive warp-Balancing ELL format for efficient sparse matrix-Vector multiplication on GPUs," *Proc. Int. Conf. Parallel Process.*, pp. 11–20, 2013.

[8] E. Anderson *et al.*, "LAPACK Users' Guide: Third Edition," p. 434, 1999.

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *AMC Queue*, vol. 6, no. April, pp. 40–53, 2008.

[10] NVIDIA Corporation, NVIDIA CUDA Programming Guide, June 2008, Version 2.0, Section 1.2.: CUDA®: "A General-Purpose Parallel Computing Platform and Programming Model".

[11] F. Ye, C. Calvin, and S. G. Petiton, "A study of SpMV implementation using MPI and OpenMP on intel many-core architecture," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8969, pp. 43–56, 2015.

[12] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," *Proc. twenty-first Annu. Symp. Parallelism algorithms Archit. - SPAA '09*, vol. i, p. 233, 2009.

[13] H. M. Aktulga, A. Buluc, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," *Proc. Int. Parallel Distrib. Process. Symp. IPDPS*, pp. 1213–1222, 2014.

[14] H. M. Aktulga *et al.*, “A High Performance Block Eigensolver for Nuclear Configuration Interaction Calculations,” *IEEE Trans. Parallel Distrib. Syst.*, pp. 1–1, 2016.

[15] P. Zardoshti, F. Khunjush, and H. Sarbazi-Azad, “Adaptive sparse matrix representation for efficient matrix-vector multiplication,” *J. Supercomput.*, vol. 72, no. 9, pp. 3366–3386, 2016.

[16] D. M. RITCHIE, “The Development of the C Language,” *Proc. 2nd ACM SIGPLAN Conf. Hist. Program. Lang. - HoPL*, vol. 28, no. 3, pp. 201–208, 1993.

[17] J. C. Slater, “A simplification of the Hartree-Fock method,” *Phys. Rev.*, vol. 81, no. 3, pp. 385–390, 1951.

[18] R. F. Bishop, “The Coupled Cluster Method,” *Lect. Notes Phys.*, vol. 510, p. 1, 1998.

[19] P. E. M. Siegbahn, “The Configuration Interaction Method,” *Eur. Summer Sch. Quantum Chem.*, vol. 1, pp. 241–284, 2000.

[20] NVidia developer’s guide, “about CUDA, *https://developer.nvidia.com/about-cuda*,” access date: 05/10/2017.

[21] G. Poli and J. Saito, “Parallel Face Recognition Processing using Neocognitron Neural Network and GPU with CUDA High Performance Architecture,” Milos Oravec, ISBN, 2010.

[22] F. Vázquez, G. Ortega, J. J. Fernández, and E. M. Garzón, “Improving the performance of the sparse matrix vector product with GPUs,” *Proc. - 10th IEEE Int. Conf. Comput. Inf. Technol. CIT-2010, 7th IEEE Int. Conf. Embed. Softw. Syst. ICESS-2010, ScalCom-2010*, pp. 1146–1151, 2010.

[23] M. M. Baskaran, “Optimizing Sparse Matrix-Vector Multiplication on GPUs using Compile-time and Run-time Strategies,” *IBM Res. Rep.*, vol. 24704, 2008.

[24] M. R. Hugues and S. G. Petiton, “Sparse matrix formats evaluation and optimization on a GPU,” *Proc. - 2010 12th IEEE Int. Conf. High Perform. Comput. Commun. HPCC 2010*, pp. 122–129, 2010.

[25] B. Neelima and P. S. Raghavendra, “Effective Sparse Matrix Representation for the GPU Architectures,” *Int. J. Comput. Sci. Eng. Appl.*, vol. 2, no. 2, pp. 151–165, 2012.

[26] A. Monakov, A. Lokhmotov, and A. Avetisyan, “Automatically tuning sparse matrix-vector multiplication for GPU architectures,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5952 LNCS, pp. 111–125, 2010.

[27] J. L. Greathouse and M. Daga, “Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format,” *Int. Conf. High Perform. Comput. Networking, Storage Anal. SC*, vol. 2015–January, no. January, pp. 769–780, 2014.

[28] F. Vázquez, J. J. Fernández, and E. M. Garzõn, "A new approach for sparse matrix vector product on NVIDIA GPUs," *Concurr. Comput. Pract. Exp.*, vol. 23, no. 8, pp. 815–826, 2011.

[29] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput. - ICS '13*, p. 273, 2013.

[30] J. a Mart and J. J. Fern, "The sparse matrix vector product on GPUs," *Aceuales*, pp. 1–13, 2009.

[31] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," *ACM SIGPLAN Not.*, vol. 45, no. 5, p. 115, 2010.

[32] P. Guo and L. Wang, "Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs," *Proc. - 2010 Int. Conf. Comput. Inf. Sci. ICCIS 2010*, pp. 1154–1157, 2010.

[33] D. Grewe and A. Lokhmotov, "Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation," *Proc. Fourth Work. Gen. Purp. Process. Graph. Process. Units - GPGPU-4*, p. 1, 2011.

[34] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications," *Int. Conf. High Perform. Comput. Networking, Storage Anal. SC*, vol. 2015–January, no. January, pp. 781–792, 2014.

[35] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining," *Proc. VLDB Endow.*, vol. 4, no. 4, pp. 231–242, 2011.

[36] K. K. Matam and K. Kothapalli, "Accelerating sparse matrix vector multiplication in iterative methods using GPU," *Proc. Int. Conf. Parallel Process.*, pp. 612–621, 2011.

[37] W. Cao, Y. Lu, Z. Li, Y. Wang, and Z. Wang, "Implementing Sparse Matrix-Vector multiplication using CUDA based on a hybrid sparse matrix format," *ICCASM 2010 - 2010 Int. Conf. Comput. Appl. Syst. Model. Proc.*, vol. 11, 2010.

[38] N. Ahmed, N. Mateev, and K. Pingali, "A framework for sparse matrix code synthesis from high-level specifications," *Supercomput. '00 Proc. 2000 ACM/IEEE Conf. Supercomput.*, p. 58, 2000.

[39] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao, "Optimization of sparse matrix-vector multiplication with variant CSR on GPUs," *Proc. Int. Conf. Parallel Distrib. Syst. - ICPADS*, pp. 165–172, 2011.

[40] P. Guo, L. Wang, and P. Chen, “A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1112–1123, 2014.

[41] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, “An Efficient Two-Dimensional Blocking Strategy for Sparse Matrix-Vector Multiplication on GPUs Categories and Subject Descriptors,” *Ics’14*, pp. 273–282, 2014.

[42] F. Vázquez, J. J. Fernández, and E. M. Garzón, “Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach,” *Parallel Comput.*, vol. 38, no. 8, pp. 408–420, 2012.

[43] W. Liu and B. Vinter, “CSR5 : An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication Categories and Subject Descriptors,” *Ics’15*, pp. 339–350, 2015.

[44] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop, “Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation,” *Proc. 2012 IEEE 26th Int. Parallel Distrib. Process. Symp. Work. IPDPSW 2012*, pp. 1696–1702, 2012.

[45] J. Godwin, J. Holewinski, and P. Sadayappan, “High-performance sparse matrix-vector multiplication on GPUs for structured grid computations,” *Proc. 5th Annu. Work. Gen. Purp. Process. with Graph. Process. Units - GPGPU-5*, pp. 47–56, 2012.

[46] W. Xu, H. Zhang, S. Jiao, D. Wang, F. Song, and Z. Liu, “Optimizing sparse matrix vector multiplication using cache blocking method on Fermi GPU,” *Proc. - 13th ACIS Int. Conf. Softw. Eng. Artif. Intell. Networking, Parallel/Distributed Comput. SNPD 2012*, pp. 231–235, 2012.

[47] Z. Wang, X. Xu, W. Zhao, Y. Zhang, and S. He, “Optimizing sparse matrix-vector multiplication on CUDA,” *ICETC 2010 - 2010 2nd Int. Conf. Educ. Technol. Comput.*, vol. 4, 2010.

[48] B. Neelima, G. R. M. Reddy, and P. S. Raghavendra, “Predicting an optimal sparse matrix format for SpMV computation on GPU,” *Proc. Int. Parallel Distrib. Process. Symp. IPDPS*, pp. 1427–1436, 2014.

[49] P. Guo, H. Huang, Q. Chen, L. Wang, E. Lee, and P. Chen, “A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on GPUs,” *Proc. 2011 TeraGrid Conf. Extrem. Digit. Discov. - TG ’11*, p. 1, 2011.

[50] S. Yan, C. Li, Y. Zhang, and H. Zhou, “yaSpMV: yet another SpMV framework on GPUs,” *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. parallel Program. - PPoPP ’14*, pp. 107–118, 2014.

[51] B. Mark Anthony and J. Stevens, “Virtually going green: The role of quantum computational chemistry in reducing pollution and toxicity in chemistry,” *Phys. Sci. Rev.*, vol. 2, no. 7, 2017.

[52] CUDA toolkit documentation, “cuSPARSE, https://docs.nvidia.com/cuda/cusparse/index.html,” access date: 07/16/2018.

[53] W. Liu and B. Vinter, “CSR5 : An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication Categories and Subject Descriptors,” *Ics’15*, pp. 339–350, 2015.

[54] CUDA toolkit documentation, “nvprof, https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview,” access date: 07/19/2018.

[55] A. Alavi, “Introduction to Full Configuration Interaction Quantum Monte Carlo with Applications to the Hubbard model Stuttgart,” *Autumn Sch. Correl. Electrons*, vol. 6, 2016.

[56] N. S. Blunt, S. D. Smart, J. a F. Kersten, J. S. Spencer, G. H. Booth, and A. Alavi, “Semi-stochastic full configuration interaction quantum Monte Carlo: developments and application,” pp. 1–10.

[57] F. Jensen, Introduction to Computational Chemistry Computational Chemistry, second edition. 2007.

[58] R. D. W., “Computational Chemistry Using the PC,” 2003.

[59] T. Helgaker, P. Jørgensen, and J. Olsen, “Molecular Electronic-Structure Theory,” 2000.

[60] I. . Ramachandran, G. Deepa, and K. Namboori, “Computational Chemistry and Molecular Modeling,” p. 405, 2008.

[61] E. G. Lewars, “Computational Chemistry, Introduction to the Theory and Applications of Molecular and Quantum Mechanics.”

[62] C. J. Cramer, “Essentials of Computational Chemistry, Theories and Models, Second Edition,” p. 596, 2004.

[63] V. P. Vysotskiy and L. S. Cederbaum, “Accurate quantum chemistry in single precision arithmetic: Correlation energy,” *J. Chem. Theory Comput.*, vol. 7, no. 2, pp. 320–326, 2011.

[64] E. Pervin and J. Webb, “Quarternions in Computer vision and Robotics,” 1982.

[65] F. A. Cotton, “Chemical Applications of Group Theory,” p. 480, 1990.

[66] J. M. Grau, C. Miguel, and A. M. Oller-Marcén, “On the Structure of Quaternion Rings Over Z/n,” *Adv. Appl. Clifford Algebr.*, vol. 25, no. 4, pp. 875–887, 2015.

[67] Soo-Chang Pei, Ching-Min Cheng, S.-C. Pei, and C.-M. Cheng, “Color Image Processing by Using Binary Quaternion- Moment-Preserving Thresholding Technique,” *Ieee Trans. Image Process.*, vol. 8, no. 5, pp. 614–628, 1999.

[68] H. S. Camarda, “Determining the eigenvalues of a quaternion matrix with a band structure,” *Comput. Phys.*, vol. 10, no. 2, pp. 180–185, 1996.