2014

# GPU Accelerated Approach to Numerical Linear Algebra and Matrix Analysis with CFD Applications

Adam Phillips
*University of Central Florida*

# GPU ACCELERATED APPROACH TO NUMERICAL LINEAR ALGEBRA AND MATRIX ANALYSIS WITH CFD APPLICATIONS

by

ADAM D. PHILLIPS

A thesis submitted in partial fulfilment of the requirements
for the Honors in the Major Program in Mathematics
in the College of Sciences
and in The Burnett Honors College
at the University of Central Florida
Orlando, Florida

Spring Term 2014

Thesis Chair: Dr. Bhimsen Shivamoggi

# ABSTRACT

A GPU accelerated approach to numerical linear algebra and matrix analysis with CFD applications is presented. The works objectives are to (1) develop stable and efficient algorithms utilizing multiple NVIDIA GPUs with CUDA to accelerate common matrix computations, (2) optimize these algorithms through CPU/GPU memory allocation, GPU kernel development, CPU/GPU communication, data transfer and bandwidth control to (3) develop parallel CFD applications for Navier Stokes and Lattice Boltzmann analysis methods. Special consideration will be given to performing the linear algebra algorithms under certain matrix types (banded, dense, diagonal, sparse, symmetric and triangular). Benchmarks are performed for all analyses with baseline CPU times being determined to find speed-up factors and measure computational capability of the GPU accelerated algorithms. The GPU implemented algorithms used in this work along with the optimization techniques performed are measured against preexisting work and test matrices available in the NIST Matrix Market. CFD analysis looked to strengthen the assessment of this work by providing a direct engineering application to analysis that would benefit from matrix optimization techniques and accelerated algorithms. Overall, this work desired to develop optimization for selected linear algebra and matrix computations performed with modern GPU architectures and CUDA developer which were applied directly to mathematical and engineering applications through CFD analysis.

# DEDICATION

This thesis wouldn't have been possible without several key individuals and softwares. I have been blessed throughout my life with opportunities that have exposed me to new experiences, allowed me to meet and interact with interesting and intelligent people, and directly impact others through my two passions: teaching and research. I look to mention a few now.

Guidance provided by thesis advisor, Dr. Bhimsen Shivamoggi along with committee members Dr. Zhisheng Shuai and Dr. Alain Kassab served as the backbone throughout my academic career at UCF. Their insight, willingness, and trustworthiness helped instill a confidence in my research abilities, a passion for scientific discovery and a desire to teach academically.

A special thanks goes out to fellow researcher and friend, Kevin Gleason. The challenges in the process of writing an undergraduate thesis, presenting research at conferences, and publishing academic work were made easier by having someone to experience it with. My best wishes goes out to him and his masters thesis under Dr. Shawn Putnam. Dr. Shawn Putnam gave my first exposure my undergraduate research at UCF, and I am grateful for his career and research advice. To Alan, Harish, Josh and Mehrdad I wish you the best in your academic careers and future endeavors.

I would like to thank Peter Bradley and Dr. Robert Amaro at the National Institute of Standards and Technology for their assistance and guidance during my research internship with the Department of Homeland Security. Upon completing this internship, I kindled a passion for scientific discovery and saw research as a viable career option, both direct results of their mentorship.

I would like to thank my bosses at Lockheed Martin, Kenneth Flowers and Dr. Paul Zarda along with other Lockheed Martin engineers and interns for the challenging projects while allowing me to keep my academics first. Similar thanks goes to my bosses at Mitsubishi Power Systems of America, Michael Glover and Alex Martinez for taking a chance on me as a freshman to be a capable engineering intern.

I would like to thank Dr. Miao Liu for the opportunity to be an undergraduate teaching assistant. The freedom to create audio solutions for the homework, help grade and administer exams and understand the procedural background of teaching at a university gave me appreciated teaching experience. Similar thanks goes to Mrs. Kim Small for the opportunity to teach freshman engineering students. My exposure gained through experiences and knowledge gained through mentorships, blended perfectly with my passion to help and teach other students. These experiences validated my choice to continue graduate studies to pursue research and teaching.

To all my friends, colleagues and acquaintances I thank you dearly. I will not attempt to list names, as there have been too many individuals who fall under this category who have been helpful at certain times in my life.

Most importantly, this would have possible without my family: Mom, Dad, Breanna, Noah, Grandma and Grandpa. To my brother Noah, you are the inspiration to my life and will be who I will always live through. To my grandpa, I did what has to be done. The lessons I have learned from each of them has lead to the creation of a unique individual who will always do what is right at all cost and lives to help others succeed.

Thanks goes out to NVIDIA for providing its free parallel computing platform in CUDA equipped with the CUDA Toolkit which included programming guides, user manuals, and API reference. Without NVIDIA CUDA, programming GPUs would have been a convoluted and strenuous task. Thanks goes out to LaTeX for its elegant typesetting system, TeX MAKER for its effective editor in writing this thesis in LaTeX and MiKTeX for compiling this thesis. Without LaTeX, I would have been enslaved to write this thesis in Microsoft Word.

Thanks goes out to the Burnett Honors College for the Honors in the Major program that allows undergraduates to work on thesis projects, and the Office of Undergraduate Research for encouraging research. And to the University of Central Florida, I thank you for the best years of my life. The accomplishments, experiences and impacts made during my time as an undergraduate is something that will always make me a Knight.

# ACKNOWLEDGMENTS

This work was partially funded by the UCF Undergraduate Research Grant and the Burnett Honors College HIM Scholarship.

# NOMENCLATURE

## Engineering Symbols

| | |
|---|---|
| $A$ | Area |
| $n_\alpha(x_i, t)$ | Boolean Particle Number |
| $C_\alpha(\{n_\beta\})$ | Collision Operator |
| $\rho$ | Density |
| $\alpha$ | Discrete Velocity |
| $\beta$ | Discrete Velocity |
| $B$ | Extensive Property |
| $f^{eq}$ | Equilibrium Function |
| $f(x, v, t)$ | Fluid Particle Distribution Function |
| $\epsilon$ | Fluid Parameter |
| $\nabla$ | Gradient |
| $\mathbf{g}$ | Gravity |
| $b$ | Intensive Property |
| $l$ | Length |
| $m$ | Mass |
| $\mathbf{n}$ | Normal Vector |
| $t$ | Time |
| $v_\alpha$ | Particle Velocity |
| $p$ | Pressure |
| $\tau$ | Relaxation Time |
| $v$ | Velocity |
| $\mathbf{V}$ | Velocity Tensor |
| $\mu$ | Viscosity |

| | |
|---|---|
| $V$ | Volume |
| $\omega$ | Weight |

## Mathematical Symbols

| | |
|---|---|
| $(x, y)$ | 2D Cartesian Coordinate |
| $(x, y, z)$ | 3D Cartesian Coordinate |
| $+$ | Addition Operator |
| $j$ | Column Index |
| $\delta$ | Change in Quantity (Small) |
| $\Delta$ | Change in Quantity (Large) |
| $\cdots$ | Ellipses |
| $q$ | Exponent |
| $\in$ | Contained In |
| $\forall$ | For All |
| $\implies$ | Implies |
| $i$ , $j$ | Indices |
| $\infty$ | Infinity |
| $\int$ | Integral Notation |
| $\bar{\text{m}}$ | Minus |
| $\cdot$ | Multiplication Operator |
| $O(\ )$ | Order of Magnitude |
| $\frac{\partial}{\partial t}$ | 1st Order Partial Derivative Notation |
| $\frac{\partial^2}{\partial t^2}$ | 2nd Order Partial Derivative Notation |
| $\bar{\text{p}}$ | Plus |
| $\pm$ | Plus-Minus |
| $\prod$ | Product Notation |

| | |
|---|---|
| $\frac{d}{dt}$ | Regular Derivative Notation |
| $a \leq x \leq b$ | Ranging Index $x$ from $a$ to $b$ |
| $i$ | Row Index |
| $A$ | Scalar $A$ |
| $s$ | Sign |
| $c$ | Significand |
| $\sum$ | Summation Notation |
| $\exists$ | There Exists |
| $\vec{A}$ | Vector $A$ |

## Matrix Symbols

| | |
|---|---|
| $\|A\|$ | Cardinality of set $A$ |
| $n$ | Column Dimension of Matrix |
| $\overline{M}$ | Conjugate Matrix |
| $\det(M)$ | Determinant of Matrix |
| $\cap$ | Intersection of Sets |
| $M^{-1}$ | Inverse Matrix |
| $M \mid M$ | Partitioned Matrix |
| $m$ | Row Dimension of Matrix |
| $A$ | Set $A$ |
| $\|M\| = m \times n$ | Size of Matrix |
| $M$ | Standard Matrix |
| $\subset$ | Subset |
| $M^T$ | Transpose of Matrix |
| $\cup$ | Union of Sets |

## Matrices and Vectors

| | |
|---|---|
| $M_\lambda$ | Algebraic Multiplicity of $\lambda$ |
| $E$ | Banded Matrix |
| $A$ | Base Matrix |
| $B$ | Base Matrix |
| $D(\lambda)$ | Characteristic Equation |
| $CP$ | Characteristic Polynomial |
| $C'$ | Cofactor Matrix |
| $\mathbb{C}$ | Complex Numbers Set |
| $y$ | Decomposition Vector |
| $z$ | Decomposition Vector |
| $\Delta\lambda$ | Defect of $\lambda$ |
| $F$ | Dense Matrix |
| $G$ | Diagonal Matrix |
| $\lambda$ | Eigenvalue |
| $\xi$ | Eigenvector |
| $E_k$ | Elementary Matrix |
| $\emptyset$ | Empty Set |
| $m_\lambda$ | Geometric Multiplicity of $\lambda$ |
| $I$ | Identity Matrix |
| $\mathbb{Z}$ | Integer Numbers Set |
| $\mathbb{I}$ | Irrational Numbers Set |
| $k_1, k_2$ | Left-Half/Right-Half Bandwidth |
| $L$ | Lower Matrix |

| | |
|---|---|
| $M^{'}$ | Minor Matrix |
| $\mathbb{N}$ | Natural Numbers Set |
| $K$ | Orthogonal Matrix |
| $\mathbb{Q}$ | Rational Numbers Set |
| $\mathbb{R}$ | Real Numbers Set |
| $C$ | Resultant Addition Matrix |
| $D$ | Resultant Multiplication Matrix |
| $b$ | Right-Hand Side Vector |
| $H$ | Sparse Matrix |
| $M$ | Standard Matrix |
| $J$ | Symmetric Matrix |
| $x$ | Solution Vector |
| $U$ | Universal Set |
| $U$ | Upper Matrix |

## Acronyms

| | |
|---|---|
| BGK | Bhatnagar-Gross-Krook |
| BLAS | Basic Linear Algebra Subprgrams |
| CFD | Computational Fluid Dynamics |
| COO | Coordinate Matrix Format |
| CSR | Compressed Sparse Row Matrix Format |
| CUDA | Compute Unified Device Architecture |
| CV | Control Volume |
| DIA | Diagonal Format |
| ELL | ELLPACK Matrix Format |

| | |
|---|---|
| FLOPS | Floating-point Operations Per Second |
| GPGPU | General Purpose Graphics Processing Unit |
| GPU | Graphics Processing Unit |
| IEEE | Institute of Electrical and Electronics Engineers |
| IC | Integrated Circuit |
| LAPACK | Linear Algebra PACKage |
| LB | lattice Boltzmann |
| LGA | Lattice Gas Automation Model |
| LU | Lower-Upper Decomposition |
| MAGMA | Matrix Algebra on GPU and Multicore Architectures |
| NS | Navier-Stokes |
| PDE | Partial Differential Equation |
| SpMV | Sparse Matrix Vector |
| SYMV | Symmetric Matrix Vector |
| SYS | System |

# TABLE OF CONTENTS

LIST OF FIGURES . . . . . xvi

LIST OF TABLES . . . . . xix

CHAPTER 1. INTRODUCTION . . . . . 1

1. Motivation . . . . . 1

2. Research Objectives . . . . . 3

3. Thesis Outline . . . . . 4

CHAPTER 2. BACKGROUND . . . . . 5

1. Algebra . . . . . 12

1.1 Matrix Operations . . . . . 35

1.1.1 Addition . . . . . 35

1.1.2 Multiplication . . . . . 39

1.2 Matrix Applications . . . . . 43

1.2.1 Determinant . . . . . 43

1.2.2 Inverse . . . . . 47

1.2.3 LU Decomposition . . . . . 50

1.2.4 System of Linear Equations . . . . . 53

1.3 Matrix Types . . . . . 57

1.3.1 Banded . . . . . 57

1.3.2 Dense . . . . . 60

1.3.3 Diagonal . . . . . 61

1.3.4 Sparse . . . . . 62

1.3.5 Symmetric . . . . . 65

1.3.6 Triangular . . . 65
2. Graphic Processing Unit . . . 66
2.1 Historical Overview . . . 66
2.2 Architecture Overview . . . 70
2.3 Memory Overview . . . 73
2.4 Precision & Accuracy Overview . . . 75
2.5 CUDA C Overview . . . 76
3. Computational Fluid Dynamics . . . 77
3.1 Historical Overview . . . 77
3.2 CFD Analysis Methods . . . 78
3.2.1 Navier-Stokes . . . 78
3.2.2 Lattice Boltzmann . . . 84

CHAPTER 3. LITERARY REVIEW . . . 87
1. GPU Computing for Numerical Linear Algebra and Matrices . . . 87
2. GPU Computing for CFD Applications . . . 91

CHAPTER 4. METHODOLOGY . . . 93
1. Computational Linear Algebra . . . 94
2. GPU Integration . . . 96
2.1 Scalable Link Interface (SLI) . . . 100
2.2 Algorithm Verification . . . 102
2.3 Algorithm Performance Metrics . . . 103
2.4 GPU Comparison . . . 106
3. CFD Applications . . . 108
3.1 Steady Flow Past a Cylinder . . . 110
3.2 Flat Plate Boundary Layer . . . 111

CHAPTER 5. LINEAR ALGEBRA RESULTS . . . . . 112
0.3 Addition . . . . . 113
0.4 Multiplication . . . . . 115
0.5 Determinant . . . . . 117
0.6 Inverse . . . . . 118
0.7 LU Decomposition . . . . . 120

CHAPTER 6. CFD RESULTS . . . . . 121
1. Flow Around a Cylinder . . . . . 121
2. Flat Plate Boundary Layer . . . . . 126

CHAPTER 7. CONCLUSIONS . . . . . 129
1. Concluding Statements . . . . . 129
2. Recommendations for Future Work . . . . . 131

APPENDIX A. EXAMPLE MATRICES . . . . . 132

REFERENCES . . . . . 134

# LIST OF FIGURES


**Figure 2.1** Relationships Between Mathematics, Science and Engineering . . . . . 6

**Figure 2.2** Mathematical, Scientific and Engineering Analysis Processes . . . . . 9

**Figure 2.3** Scalar and Vector Quantities . . . . . 9

**Figure 2.4** Geometric Vector Representation . . . . . 10

**Figure 2.5** Stages of Algebra . . . . . 12

**Figure 2.6** Babylonian Mathematics Example . . . . . 16

**Figure 2.7** Greek Mathematics Example . . . . . 17

**Figure 2.8** Chinese Mathematics Example . . . . . 18

**Figure 2.9** Arabian Mathematics Example . . . . . 22

**Figure 2.10** French Mathematics Example . . . . . 25

**Figure 2.11** Geometric Determinant . . . . . 44

**Figure 2.12** Upper and Lower Triangular Matrices . . . . . 51

**Figure 2.13** System of Linear Equations . . . . . 53

**Figure 2.14** Column-major vs. Row-major memory ordering . . . . . 60

**Figure 2.15** Dense and Sparse Matrices . . . . . 62

**Figure 2.16** Diagonal (DIA) Format . . . . . 62

**Figure 2.17** ELLPACK (ELL) Format . . . . . 63

**Figure 2.18** Coordinate (COO) Format . . . . . 64

**Figure 2.19** Compressed Sparse Row (CSR) Format . . . . . 64

**Figure 2.20** Graphic Processors: 1970-1990 . . . . . 66

**Figure 2.21** Graphic Processors: 1990-2014 . . . . . 67

**Figure 2.22** CUDA Programming Model: Kernel, Grid and Block . . . . . 71

**Figure 2.23** CPU Memory Breakdown . . . . . 73

**Figure 2.24** GPU Memory Hierarchy . . . . . 74

**Figure 2.25** Reynolds Transport Theorem . . . . . 78
**Figure 2.26** Conservation of Mass . . . . . 80
**Figure 2.27** Conservation of Momentum . . . . . 82
**Figure 2.28** Lattice Gas Automaton (LGA) Model . . . . . 85
**Figure 2.29** Bhatnagar-Gross-Krook (BGK) Approximation . . . . . 86

**Figure 4.1** Thesis Outline . . . . . 93
**Figure 4.2** CityPlots: Dense (Left), Banded (Middle) and Sparse (Right) . . . . . 95
**Figure 4.3** 3D Interactive Matrix Plot . . . . . 95
**Figure 4.4** GPU Detection K2000M: Device Manager . . . . . 96
**Figure 4.5** CUDA Samples: deviceQuery . . . . . 97
**Figure 4.6** CUDA Samples: bandwidthTest . . . . . 98
**Figure 4.7** CUDA Design Cycle . . . . . 99
**Figure 4.8** CPU and GPU Limited Setups . . . . . 100
**Figure 4.9** PCIe GPU Setups . . . . . 101
**Figure 4.10** Default Log File for Compute Command Line Profiler . . . . . 105
**Figure 4.11** Numerical Solution Method . . . . . 109
**Figure 4.12** CFD Application: Steady Flow Past a Cylinder . . . . . 110
**Figure 4.13** CFD Application: Flat Plate Boundary Layer . . . . . 111

**Figure 5.1** Addition: Acceleration Factor as a Function of Matrix Dimension . . . . 114
**Figure 5.2** Addition: Execution Time as a Function of Matrix Dimension . . . . . 114
**Figure 5.3** Multiplication: Acceleration Factor as a Function of Matrix Dimension . 116
**Figure 5.4** Multiplication: Execution Time as a Function of Matrix Dimension . . . 116
**Figure 5.5** Inverse: Acceleration Factor as a Function of Matrix Dimension . . . . . 119
**Figure 5.6** Inverse: Execution Time as a Function of Matrix Dimension . . . . . 119

**Figure 6.1** Flow Around Cylinder: Re = 1 . . . . . 121
**Figure 6.2** Flow Around Cylinder: Re = 10, 20 . . . . . 122
**Figure 6.3** Flow Around Cylinder: Re = 25 . . . . . 122
**Figure 6.4** Flow Around Cylinder: Re = 40 . . . . . 123
**Figure 6.5** Flow Around Cylinder: Re = 50 . . . . . 123
**Figure 6.6** Flow Profile at t = 10, 20, 30, 40, 50 seconds . . . . . 125
**Figure 6.7** Flat Plate : Re = 1, 20 . . . . . 126
**Figure 6.8** Flat Plate: Re = 50, 100 . . . . . 127
**Figure 6.9** Flat Plate: Re = 250, 500 . . . . . 127
**Figure 6.10** Flat Plate: Re = 1000, 2500 . . . . . 128
**Figure 6.11** Flat Plate: Re = 5000, 10000 . . . . . 128

# LIST OF TABLES


**Table 2.1** Common Sets . . . . . 29

**Table 2.2** Set Properties and Rules . . . . . 30

**Table 2.3** Field Properties . . . . . 31

**Table 2.4** Vector Space Properties . . . . . 32

**Table 2.5** Matrix Addition Properties . . . . . 36

**Table 2.6** Matrix Multiplication Properties . . . . . 40

**Table 2.7** Determinant Properties . . . . . 43

**Table 2.8** Inverse Properties . . . . . 47

**Table 3.1** Dense Linear Algebra Selected Prior Work . . . . . 88

**Table 3.2** Sparse Linear Algebra Selected Prior Work . . . . . 89

**Table 3.3** Regular Linear Algebra Selected Prior Work . . . . . 90

**Table 3.4** Navier-Stokes Selected Prior Work . . . . . 91

**Table 3.5** Lattice Boltzmann Methods Selected Prior Work . . . . . 92

**Table 4.1** Comprehensive Command Line Profiler Commands . . . . . 105

**Table 4.2** CPU/GPU Specifications . . . . . 107

**Table 6.1** Flow Around a Cylinder Variables . . . . . 124

# CHAPTER 1

# INTRODUCTION

## 1. Motivation

Linear algebra is a central discipline in mathematics with applications present in the fields of science and engineering such as chemical kinetics, fluid flow simulations, and economic modeling [91]. In total, there are just two main approaches to linear algebra: abstract and concrete. The abstract approach is an axiomatic analysis of the subject with the goal of developing matrix theory, whereas the concrete approach focuses on direct numerical implementation of the matrix theory [63]. A strong foundation in both approaches to linear algebra is essential to being successful in the fields of mathematics, science and engineering. For this work, an abstract approach is first taken to mathematically develop numerical matrix algorithms, which are secondarily applied through a concrete approach to scientific parallel computing and engineering fluid dynamic applications.

Numerical algorithms are a central focus for applications in science and engineering, becoming instrumental in advanced analyses and simulations with progressive technological advancements in computer science. CAD design, business computations, and structural analysis are examples applications which have benefited from the likes of software running numerical algorithms such as Autodesk AutoCAD, Microsoft Excel, and Siemens PLM NX. The computer, a machine capable of executing mathematical computations at a speed and efficiency outperforming any human provides computational resources to seamlessly carry out these numerical algorithms [25]. Historically as computational power increased numerical algorithms became more complex, models became more intricate, and simulations became more detailed. With the technology of integrated circuits rapidly increasing, computers became more powerful (more memory, higher bandwidth, multiple cores) and much faster (higher clock and memory access speeds), and were capable of meeting consumer demands for scientific computing [74]. Unfortunately this ap-

parent trend of unbounded increasing computational performance appeared to find its limit in the early 2000's, as heat loads experienced by integrated circuits were incapable of being removed by current cooling technologies [34]. Researchers are currently developing materials capable of dissipating higher heat loads to construct future integrated circuits and cooling technologies are being investigated for faster methods to cool computer components, however no feasible solution is currently available for this heat dissipation problem and numerical algorithms are bottlenecked as a result [41].

Graphic processing units (GPUs), computer hardware responsible for rendering and generating imagery on a computer screen were studied in onset of the computational power crisis as a possible tool for added computing resources. High resolution computer screens and application with fast frame rates force GPUs to be efficient at rendering million of pixels quickly. As a result GPUs operate on a model that allow for many small tasks (individual pixels) to be carried out simultaneously, and are inherently parallel computational devices. General purpose computing on graphic processing units (GPGPU) was born by replacing individual pixels with scientific computations [23]. NVIDIA, a primary company developing graphic cards released CUDA in 2007, a toolkit and parallel computing platform for scientific computing [100]. CUDA provides a similar computing environment to standard serial programming performed on the central processing unit (CPU). Serial code run on the CPU bottlenecked by heat restrictions could now effectively be run in a parallel nature by GPUs to remove time constraints in execution [22]. Overall GPUs allow numerical algorithms to continue to increase in complexity and accuracy, with results obtained in a fraction of the time they would on the CPU.

Through an interdisciplinary approach between the fields of mathematics, science and engineering, this work desires to utilize programmable NVIDIA graphic processing units to accelerate computationally intensive and common mathematical algorithms in the form of linear algebra and matrix operations, as well as two main applications in computational fluid dynamics (CFD) in the Navier Stokes and Lattice Boltzmann methods.

## 2. Research Objectives

**• Develop stable and efficient algorithms using CUDA for NVIDIA GPUs**

NVIDIA GPUs are the standard for parallel computing, and current CUDA toolkit presents promising amount of resources to optimize computational resources and accelerate parallel GPU code. Integration with Visual Studio provides development and compilation of C parallel code easy to write and accelerate. Algorithms are aimed to be stable (yield correct, convergent, and predictable results) and efficient (be robust, fast and complete).

**• Optimize linear algebra algorithms for performance**

Development of the parallel GPU algorithms is vital to ensure accuracy and precision. Optimization of the parallel GPU algorithms is the vital as it ensures optimal performance. Proper GPU memory allocation provides maximum acceleration factors. Test matrices from the NIST Matrix Market possessing different matrix types will be studied for performance.

**• Develop parallel CFD algorithms**

Computational Fluid Dynamics (CFD) presents a field that relies heavily on numerical algorithms that possess CPU bottlenecks that can be parallelized and accelerated with GPU computing. Two primary examples covered in this work are steady flow past a cylinder and flat plate flow. Taking parallelized numerical algebraic code and techniques, these CFD applications will be parallelized with MATLAB's Parallel Computing Toolbox and compared to ANSYS FLUENT models.

**• Implement single vs. multiple GPU approach**

NVIDIA possesses three type of graphic cards: GeForce, Quadro, and Tesla. This work looks to compare the performance results between a Quadro mobile workstation GPU and a GeForce desktop workstation GPU for a single GPU approach. Then two identical GeForce GPUs through an SLI connection will be introduced to understand GPU scaling laws and implementation of a multi-GPU approach.

• **Introduce GPU computing to the general audience**

GPU computing is the future of code acceleration. A cheaper alternative to supercomputers or computer clusters, GPU computing only requires a programmable graphics card. NVIDIA provides a multitude of resources and guides for parallel computing, a free toolkit and developer with CUDA, and integration with primary software in Visual Studio. The author's current university does not consistently offer a GPU computing course (sparingly a parallel computational course is offered: Fall 2005, Spring 2010). Much research is done with computational intense applications that could greatly benefit from GPU acceleration, and the author desires to present seminars that hopefully will lead to its further integration in research.

## 3. Thesis Outline

The remainder of this thesis will be outlined as follows. Chapter 2 initially provides the fundamental background information on the three subject manners discussed in this work. It establishes the connection between mathematics, science and engineering as well as bestows the essential information required to understand this work through historical, mathematical and numerical overviews. Chapter 3 then outlines the prior research and results for each topic covered in this work. Chapter 4 likewise explains the methodology implemented for this work. Chapter 5 and 6 subsequently discuss the numerical linear algebra and CFD results of this work through the analysis and verification. Chapter 7 lastly arranges a summary of the results through a discussion of this work by revisiting the research objectives as well as list areas of interest for future studies. Appendix A stores the example matrices referenced in Chapter 2 to help explain the topics of this work. The entirety of the codes used in this work can be found on the authors personal website at https://sites.google.com/site/adamphillipshomepage/codes.

# CHAPTER 2

# BACKGROUND

In this chapter the background foundation is developed for the three main topics covered in this work: linear algebra, the graphic processing unit and computational fluid dynamics. Each main topic is organized into overview sections consisting of historical and mathematical examinations. These examinations look to add value and appreciation, as well as aid in comprehension of the mathematical theory and numerical analysis covered for each topic. A philosophical overview is initially presented to explain the subject manners this work encompasses: mathematics, science and engineering. The philosophical overview not only ties the three subject manners directly to their respective topic counterparts, but it also presents several advantages of the multidisciplinary research approach this work implements. To commence, the topic of linear algebra, composed of historical, mathematical and numerical overviews on matrix operations and their applications is discussed. In addition, the topic of the graphic processing unit, consisting of historical, architectural and memory overviews is reviewed. To conclude, the topic of computational fluid dynamics, composed of a historical overview coupled with mathematical and numerical overviews on CFD analysis methods is explained.

While the background chapter is not an exhaustive teaching of the three topics covered, completion of this chapter should give the reader all necessary knowledge in numerical linear algebra and its applications to graphic processing units and computational fluid dynamics needed to fully comprehend this work. Readers who desire more in-depth knowledge into the topics covered, are encouraged to check out the references section of this work. Lastly, to aid in the comprehension of the topics covered in this work, standard example matrices are introduced to explain matrix theory, then are extended to GPU computing to explain programming applications and lastly used to relate to the CFD analysis.

*Philosophical Overview.* Mathematics, science and engineering are three subject manners which encompass the three main topics discussed in this work: linear algebra, GPU computing (GPGPU) and CFD. The pursuit of knowledge is the one aspect these subject manner share, but they serve different fundamental purposes as shown in Figure 2.1. Mathematics desires knowledge for insight to build relationships, and through an axiomatic approach uses proper logic and reasoning based on principal abstractions to develop mathematical theory. Abstractions are statements accepted as truth in mathematics that become the foundation of mathematical proofs. Mathematics benefits from science and engineering as areas of interest in these two subject manners can be emphasized for future theory development. Science on the other hand desires knowledge to answer questions, and through an empirical approach performs experiments and collects measurements for these inquiries. Science benefits from mathematics as it provides the theory for the models, and benefits from engineering as it creates the instrumentation needed for precise experimentation. Engineering likewise desires knowledge to solve problems, and through a practical approach applies scientific knowledge to develop leading-edge technology and inventions. Engineering benefits from mathematics as it provides the tool for analysis, and benefits from science as it provides models backed by experimental evidence for designs [13, 119]. The following pages further detail the three topics.

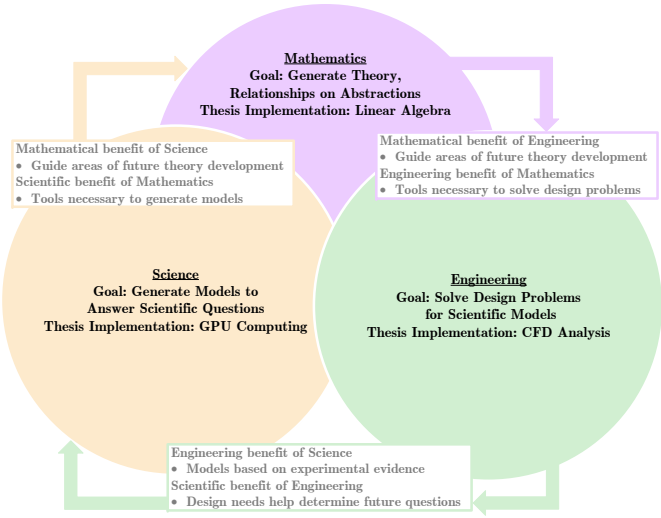



**Figure 2.1:** Relationships Between Mathematics, Science and Engineering

Mathematics is any collection of accepted abstractions through which observable patterns and relationships can be built using rational thought into absolute mathematical theory. Being built upon accepted abstractions, mathematics is necessary truth from which all sound mathematical questions can be conclusively resolved. These fundamental abstractions lead to theorems; proven truths beyond the possibility of falsification that become the groundwork for mathematical theory [61]. Mathematical theory has an abstract importance by relating previously unrelated topics in unexpected fashions, as well as a concrete importance through applications into nature. Surprisingly, history has shown that mathematics presently considered abstract usually develops later into explanations of behaviors or patterns for the physical world. Fractals (snowflake), geometric sequence (bacteria growth), and symmetry (butterfly) are all examples of these applications [44, 86, 135].

Science is any system or process of obtaining knowledge through careful unbiased observations or systematic experimentation that result in a greater understanding of the physical world. As abstractions in mathematics form theorems and principles, hypotheses in science form theories and laws. Science looks to produce knowledge gained from studying patterns in nature, and uses mathematics as a tool to symbolically represent observations as well as explain and validate proposed conclusions [84, 119]. Unlike mathematics, science is not based on absolute truth and is subject to change based on later realized false presumptions or broader well-defined theories. The existence of caloric, a fluid that flows from warmer to colder bodies and the human bodies four humors (fire, earth, water and air) is an example of a superseded scientific theory [29]. The extension of Newtonian mechanics to Einsteins theory of relativity and classical to quantum physics are examples of theories that increasingly approximate the physical world [36].

Engineering is the application of knowledge (mathematical or scientific) to creativity solve prevalent societal problems with given constraints, resulting in advanced technologies. The abstractions in mathematics as well as the hypotheses in science relate directly to the design procedures and standards produced in engineering [119]. Whereas mathematics and science are primarily focused on knowing that certain theorems and theories hold true, engineering is primarily

focused on knowing how to apply these theorems and theories to unique challenges affecting the physical world. While mathematical theorems are proven truths and scientific theories are descriptions of reality able to be falsified, engineering practices constitute reality and improve steadily.

Given that the three subject manners structurally differ in their goals, it can be expected that each subject manner exhibits a different approach to analysis. Figure 2.2 outlines these analysis processes for mathematics, science and engineering. The mathematical method is founded on the desire for insight, in which abstractions are created to present arguments with sound and valid logic to generate theory. The two common areas of refinement in the mathematical process occurs if invalid logic is used to present an argument, or if the results obtained from the application of the argument are unsound. In both cases, the logic used to expand the argument must be reviewed. The scientific method is founded on the ability to answer a question, in which a hypothesis is first constructed and then an experiment is formed to test the hypothesis and collect data. The two common areas of refinement in the scientific process occurs if the procedure used to perform the experiment is not working properly, or if the results obtained from the experiment do not match with the original hypothesis. In the first case, the procedure used must be refined and in the second case the original hypothesis must be reevaluated. The engineering method is founded on the ability to define and solve a problem, in which requirements are first specified and a solution in the form of a prototype is constructed. The two common areas of refinement in the engineering process occurs if the prototype is not practical or does not meet the specified requirements for design. In both cases, the prototype must be reevaluated [45, 119].

Overall, mathematics seeks patterns in abstractions to make logical connections, science seeks patterns in phenomena to make the world understandable, and engineering seeks patterns in designs to make the world manipulable. Mathematics seeks to show logical proofs of abstract connections, science seeks to show developed theories fit data, and engineering seeks to demonstrate that designs work. Most importantly, mathematics can't provide all possible connections, science can't provide answers to all questions, engineering can't design solutions for all problems [119].

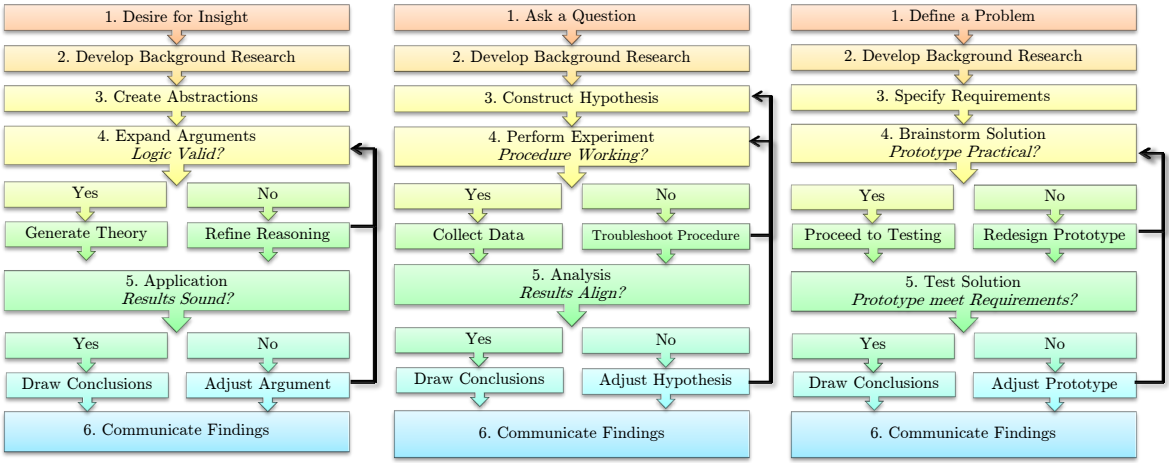



**Figure 2.2:** Mathematical (left), Scientific (middle) and Engineering (right) Analysis Processes

Mathematics, science and engineering are quantitative subject manners that rely on base quantities to generate abstractions, take experimental measurements, and analyze designs. For these quantitative subjects there exists two base quantities, the scalar and the vector that accomplish such tasks [79]. A scalar quantity represents a single numerical value or magnitude, whereas a vector quantity represents numerical value(s) with associated direction(s). Thus, a scalar quantity is a single numerical entity while a vector can be a single scalar quantity with direction or a collection of scalar quantities with directions. Examples of common scalar quantities are temperature, mass and distance. A temperature gradient, force and displacement are all examples of their vector counterparts. These quantities are shown in Figure 2.3.

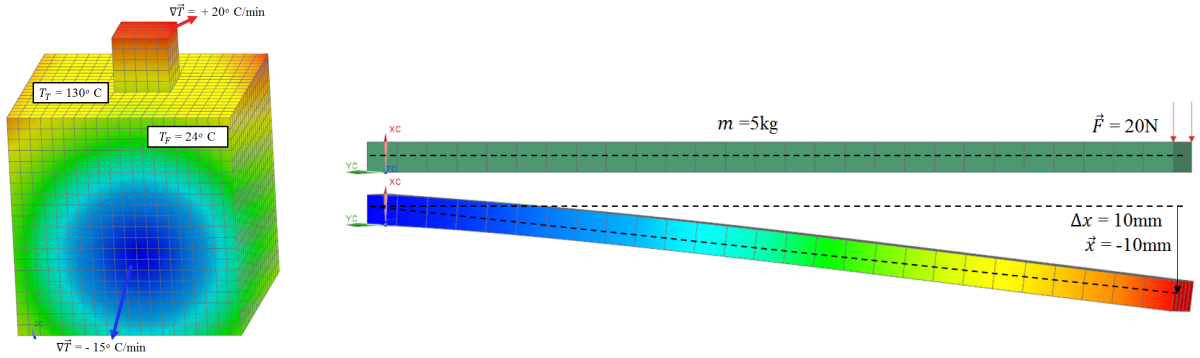



**Figure 2.3:** Scalar and Vector Quantities

Scalar and vector quantities are ubiquitous in mathematics, science and engineering analyses. Early application of these quantities is seen through geometry [127, 144]. Figure 2.4 geometrically displays scalar quantities represented by points on the graph and vector quantities represented by directed line segment. A vector is often represented by an arrow from an initial point to a terminal point, in which the length of the arrow denotes the magnitude of the vector and the direction of the arrow denotes the direction of vector [52]. Figure 2.4 displays two scalar quantities $A, B$ (as explicit points) and five vectors $\vec{A}, \vec{B}, \vec{C}, \vec{D}, \vec{E}$. Note, in total there are twelve points on this graph (two explicit scalar points and ten points from the vectors, five initial and five terminal).

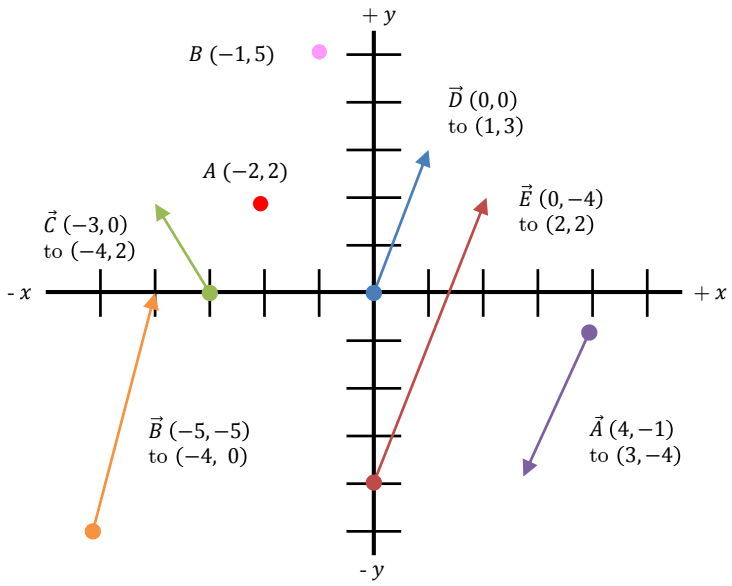



**Figure 2.4:** Geometric Vector Representation

Figure 2.4 is presented to show basic geometric laws regarding vectors. The standard procedure of how to construct a vector is now presented. To construct a vector, one must first select a initial point and a terminal point. Taking points $A$ and $B$ as initial and terminal points respectively, vector $\vec{F}$ can now be constructed. Next, the coordinates of the initial point are subtracted from the coordinates of the terminal point. Subtracting the coordinates of $B$ from the coordinate of $A$ results in $(5 - 2, -1 - (-2)) = (3, 1)$. Finally, to denote the direction of the vector the arrow base is placed at the initial point and taken to the terminal point where the arrow tip is placed.

Every vector possesses what is termed the negative vector. The negative of a vector is a vector with the same magnitude but opposite direction. Vector $\vec{A}$ is the negative of vector $\vec{D}$, since $\vec{A} = (-4-(-1), 3-4) = (-3,-1) = -(3,1) = -\vec{D}$. A vector can also be written in what is termed a linear combination. A linear combination is the sum of a collection of vectors multiplied by nonzero scalars. Vector $\vec{D}$ for example can be written as the linear combination of vector $\vec{B}$ and $\vec{C}$. That is $-2\vec{C}+\vec{B} = -2(-4-(-3),-2-0)+(-3-(-4),-2-(-1)) = -2(-2,-2)+(-1,-3) = \vec{D}$. Lastly, a scalar multiple of a vector is a vector with similar direction but different magnitude. Vector $\vec{E}$ is a scalar multiple of vector $\vec{D}$ since $\vec{E} = (2-(-4), 2-0) = (6,2) = 2(3,1) = 2\vec{D}$. Universally there are multiple approaches and applications taken towards scalars and vectors. The origin of these basic geometric principles, often called the parallelogram vector laws dates back to around 350BC from Aristotle. The geometric approach to vectors was presented initially because historically Aristotle realized these laws in similar fashion [32, 127].

In summary, while the three subject manners presented have separate objectives and analysis processes, the union of mathematics, science and engineering can certianly strengthen any research topic. In fact a multidisciplinary research approach between these subject manners has produced the fields of bioinformatics, nanotechnology and quantum computing [43, 90, 113]. With this in mind, a multidisciplinary research approach was incorporated between these three subjects in this work. Through the combination of matrix theory and linear algebra principles as a development tool, numerical models were generated to answer questions regarding GPGPU uses for algorithm acceleration, which then were applied to perform CFD analysis on selected design problems. The two base quantities types introduced, the scalar and the vector, provide a physical linkage between these three subjects, and will next be extended to the elementary ideas of algebra that allow for the exploration into matrix mathematics. As a reference for the reader, definitions words are bolded in the following sections for the elementary ideas of algebra. The reader is encouraged to refer to these initial definitions as recurrence of these elementary words is common.

## 1. Algebra

Algebra is the initial central theme of mathematics, under which the field of linear algebra exists. The discussion into linear algebra thus starts with an introduction of algebra, and develops into the mathematical connection between the two topics. Broad in scope, algebra is challenged with finding a proper definition. Concisely, **algebra** can be described as the science which teaches how to determine unknown quantities by means of those that are known [46]. Historically, algebra is considered to be constructed into three primary stages based on mathematical representation: the rhetorical stage, syncopated stage and symbolic stage [116]. The early rhetorical stage represents an era of mathematics in which abstractions were solely presented in words and sentences. The intermediate syncopated stage covers a later time in which abstractions became shortened through abbreviations, but primarily were still presented through words and sentences. The symbolic stage describes the current period in which abstractions are only represented in total symbolization. Mathematical historians have classified four conceptual stages within the three primary stages of algebra. These conceptual stages are referred to respectfully as the geometric stage, static equation-solving stage, dynamic function stage and abstract stage [67]. Figure 2.5 displays the primary stages of algebra with historical time frames and corresponding conceptual stages.

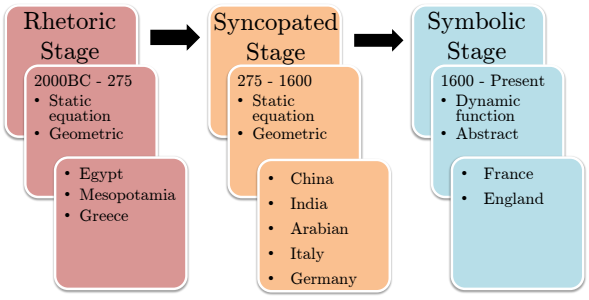



**Figure 2.5:** Three Primary Stages of Algebra

**Linear algebra** as a field of algebra is the mathematical topic comprised of elementary concepts. These concepts are the linear equation, matrix, determinant, linear transformation, linear independence, dimension, and vector space which all fall within the three primary stages of algebra [73]. The following thereby begins the discussion of these concepts as they relate historically to algebra.

*Historical Overview.* The early rhetorical stage of algebra commenced around 2000 B.C. with the Egyptians developing procedures to solve linear equations [20]. The method of false position and method of factorization were the two main techniques implemented to find solutions to these equations [67]. The **method of false position** was utilized if the problem had the form

$$x + ax = b$$

where $a$ and $b$ were known scalar values, and $x$ was the unknown value(or heap). The following problem will be used to illustrate the method of false position. Problem $24$ in Ahmes Papyrus ($\sim$1650 B.C.) asks for the value of a heap, if heap and a seventh of heap is $19$. Symbolically,

$$x + \left(\frac{1}{7}\right)x = 19.$$

First a guessed value would be assigned to the heap, usually to eliminate the fraction. In the case of this problem the heap is assigned a value of $7$. Next the guessed value would be plugged into the left hand side of the equality sign and evaluated

$$7 + \left(\frac{1}{7}\right)7 = 8.$$

If this computed value was equal to the initial right hand side, the calculation would be complete and the guessed value would be the actual value of the heap. However, this was not the case in most instances (the reasoning behind method of *false position*). Thus, the value obtained from the left hand side would next be divided by the initial right hand side. This value became the proportion that needed to be multiplied by the original value to achieve the correct value for the heap.

$$\left(\frac{19}{8}\right)7 = \left(\frac{133}{8}\right) = x$$

Verification would be carried out by plugging the achieved value into the original equation for $x$.

$$\left(\frac{133}{8}\right) + \left(\frac{1}{7}\right)\left(\frac{133}{8}\right) = 19$$

This simple verification shows the limited extent of justification introduced in ancient Egyptian mathematical proofs.

**The method of factorization**, the second technique implemented by the Egyptians to solve linear equation was utilized if the problem had the form

$$x + ax + bx = c$$

where $a$ , $b$ and $c$ are known scalar values and $x$ was the unknown value [24]. The following problem will be used to illustrate the method of factorization.

$$x + \left(\frac{2}{3}\right)x + \left(\frac{1}{2}\right)x + \left(\frac{1}{7}\right)x = 37$$

To determine the value of the heap, the left hand side of the equality was first factored to collect the similar $x$ value present in all terms.

$$x\Big(1 + \left(\frac{2}{3}\right) + \left(\frac{1}{2}\right) + \left(\frac{1}{7}\right)\Big) = 37$$

Next, the right hand side was divided by the sum within parenthesis to find the value of the heap.

$$x = 37/\Big(1 + \left(\frac{2}{3}\right) + \left(\frac{1}{2}\right) + \left(\frac{1}{7}\right)\Big) = \frac{1554}{97}$$

Verification again was carried out validate the value for the heap by substituting this value back into the original linear equation.

$$\left(\frac{1554}{97}\right) + \left(\frac{2}{3}\right)\left(\frac{1554}{97}\right) + \left(\frac{1}{2}\right)\left(\frac{1554}{97}\right) + \left(\frac{1}{7}\right)\left(\frac{1554}{97}\right) = 37$$

Most of the linear equation problems were solely practice for young Egyptian students, however some had real practical applications [20]. While the examples shown were presented in symbolic notation, the Egyptians being in the early rhetorical stage only represented equations in terms of words and sentences. As a note, the author has decided to present all examples relating to historical algebra in modern symbolization with the goal of increasing reader comprehension and achieving unity of presentation. Overall the ancient Egyptians were only capable of solving linear equations. As will be shown next, around the same time North of Egypt, the Babylonians were working not just on solutions methods for linear equations, but also those of higher order (i.e. quadratic).

Babylonians, inhabitants of ancient Mesopotamia were also focused on finding solutions to linear equations about the same time (2000 –1700 B.C.) as the early Egyptians. The problems solved by Babylonians were of greater complexity and value to the early rhetorical stage of algebra [20]. As an example, given the linear equation with a single unknown value $x$

$$\left(x + \frac{x}{7}\right) + \left(\frac{1}{11}\right)\left(x + \frac{x}{7}\right) = 60.$$

The value $x = 48\frac{7}{30}$ is then simply provided without any methodology behind its solution. The Babylonians were also capable of solving two simultaneous linear equations in two unknowns [24]. Consider the following $2 \times 2$ system as an example of this

$$\frac{x}{7} + \frac{y}{11} = 1 \ , \ \frac{6x}{7} = \frac{10y}{11} \ .$$

Where the solutions

$$\frac{x}{7} = \frac{11}{7+11} + \frac{1}{72} \ , \ \frac{y}{11} = \frac{7}{7+11} - \frac{1}{72}$$

again are provided without explanation. This implies these problems were considered trivial, requiring no solution procedure. Altogether Babylonian mathematics had two main roots, those from accountancy problems and those from cut-and-paste geometry [67]. Accountancy problems were primarily for bureaucratic purposes, whereas cut-and-paste geometry was implemented by land surveyors to understand land divisions. The surveyors cut-and-paste geometry became the greatest algebraic achievement of the Babylonians, producing a methodology to solve quadratic equations [68]. Consider the following problem from table YBC4663 as an example of this cut-and-paste geometry. Given the sum of the length and width of a rectangle is $6\frac{1}{2}$ and its area is $7\frac{1}{2}$, determine the rectangles length and width. Figure 2.6 displays this setup with a length of $x$ and a width of $y$. Symbolic translation of the two problem statements become $x + y = b$ and $xy = c$. To verify the problem solved is indeed quadratic, $xy = c$ can be expressed as $y = c/x$ and then substituted into $x + y = b$. After reorganization the quadratic equation $x^2 + c = bx$ in fact appears.

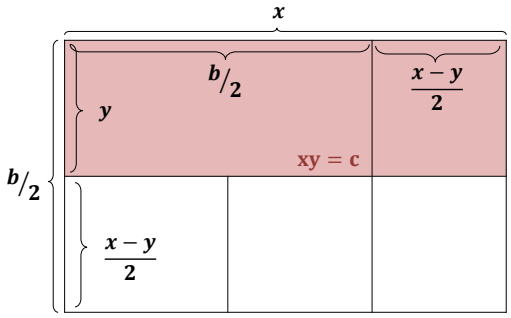



**Figure 2.6:** Babylonian Mathematics Example

The Babylonian arithmetic procedure to solve for $x$ and $y$ is as follows

1. Half the sum: $\dfrac{x+y}{2} = \dfrac{b}{2}$
2. Square the value in Step 1: $\left(\dfrac{b}{2}\right)^2 = \dfrac{b^2}{4}$
3. Subtract total area from value in Step 2: $\dfrac{b^2}{4} - c = \dfrac{b^2}{4} - xy = \dfrac{(x+y)^2 - 4xy}{4} = \dfrac{(x-y)^2}{4}$
4. Square root the value in Step 3: $\sqrt{\dfrac{(x-y)^2}{4}} = \pm\dfrac{x-y}{2}$

5a. Length: value in Step 1 + value in Step 4: $x = \dfrac{b}{2} + \dfrac{x-y}{2}$

5b. Width: value in Step 1 - value in Step 4: $y = \dfrac{b}{2} - \dfrac{x-y}{2}$

Quadratic equations in ancient times were grouped into three main types, the problem just solved was type three [67]. These three types appear during the discussion on Arabic mathematics. Most importantly, while the Babylonians were capable of solving all three types, they never attributed any distinction when solving. Broadly, Egyptians and Babylonians mathematics while helpful for an initial arithmetic understanding of algebra, experienced deficiencies that prevented its further advancement [20]. Deficiencies such as lack of general formulation of problems and rather focus on specific cases, and presentation in large quantities suggested their use being for mere exercises rather than mathematical study. The rather absence of mathematical proofs for justification of the applied arithmetic is frequent in pre-Hellenic mathematics. As will be shown next, the Greeks approached the same quadratic problems with a geometric interpretation, introducing abbreviations.

The Ancient Greeks of 600 B.C. developed geometric solutions to quadratic problems the Egyptians and Babylonians focused solely arithmetically on, searched for generality in procedure and placed importance in mathematical proofs [20]. The Greeks were the first to develop distinguished algebraic mathematicians known by name, in Euclid and Diophantus. With this the Greeks ushered in the geometric conceptual stage and transitioned into the syncopated stage of algebra [68].

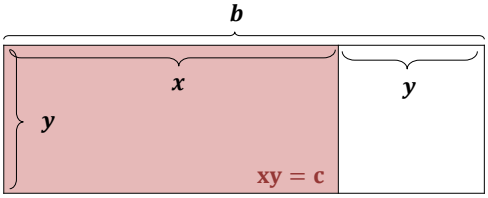



**Figure 2.7:** Greek Mathematics Example

Figure 2.7 displays Euclid's method of application of areas found in Books II–III of *Elements* to solve quadratic equations [24]. Rewriting the equation for the length $b = x+y$ as $y = b-x$ to plug into the equation the area of the rectangle $xy = c$, results in $x^2+c = bx$. This is the same quadratic equation found by the ancient Babylonians. Euclid's *Elements* consisted of 13 books (9 explicitly geometric) and is responsible for bringing the geometric conceptual stage into Greece. Greek algebraist Diophantus often called the father of algebra, authored *Artithmetica* which followed the Babylonian arithmetical approach to algebra. In it Diophantus explains methodology for solutions to determinate and indeterminate systems of equations order three. Diophantus also implemented abbreviations in *Artithmetica*, transitioning Greece into the syncopated stage of algebra [67]. As an example of the abbreviations of Diophantus, $5x^4 + 7x^3 - 6x^2 + 3x + 1$ might be abbreviated as 4SS 7C 3x M 6S u1 (SS: squared, C: cubed, x: unknown, M: minus, and u: unit) [20]. Another example of Diophantus' abbreviation is given when asked for two numbers whose sum was 20, and sum of their squares was 208. The usual representation of variables $x$ and $y$ for unknowns were not chosen. Instead Diophantus chose to represent the two unknown numbers as $10 - x$ and $10 + x$, satisfying condition one. To satisfy condition two, $(x - 10)^2 + (x + 10)^2 = 208$. Solving this quadratic problem resulted in $x = 2$, and thus the two numbers become 8 and 12. As will be shown next, the Chinese present the second elementary topic in linear algebra: the matrix.

The Chinese civilization around 200 B.C. produced the *Nine Chapters on the Mathematical Art*, an extensive work with algebraic importance. In similar format to the Egyptians and Babylonians, this text compiled 246 problems placed into specific problem sets [68]. Chapter 8 possesses the most algebraic significance as is presents solutions of simultaneous linear equations, resulting in both positive and negative numbers for both determinate and indeterminate systems of equations [20]. The first resemblance of equations as a matrix (termed the 'magic square') is presented in the *Nine Chapters on the Mathematical Art*. Consider the following system of simultaneous linear equations

$$2x + 4y + 6z = 4$$

$$x + 5y + 9z = 2$$

$$2x + y + 3z = 7$$

Figure 2.8 displays the magic square (or matrix). Solutions are obtained easily by substitution.

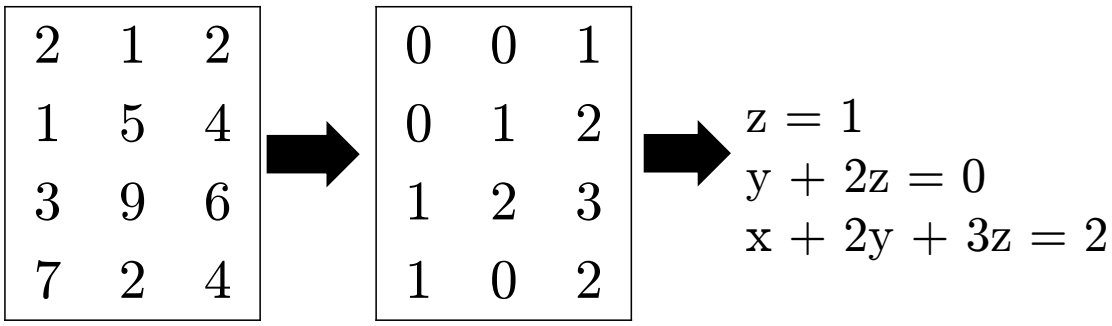



**Figure 2.8:** Chinese Mathematics Example

This method of solution is almost identical to Gaussian Elimination presented some 2000 years later, showing the advanced algebraic intellect of the early Chinese. Unfortunately Chinese mathematics was presented with many cultural challenges during its development, such as the order to burn all books by the emperor in 213 BC, or else it may have created more significant mathematical discoveries [20]. Algebra was still in the static-equation solving stage with the early Chinese, and little use of abbreviations to represent a matrix placed them in the syncopated stage of algebra [24].

The Chinese were also responsible for pursuing interest in another elementary topic of linear algebra: the linear transformation [69]. Medieval Chinese text *Ssu-yuan yu-chien* (1303) written by Chu Shih-chieh describes the transformation method (termed Horners Method some 500 years later) implemented to find roots of equations (up to order 14) [24]. The following example shows the **transformation** (called **fan-fa**) method

$$x^2 + 252x - 5292 = 0.$$

First Chu Shih-chieh looked for two integer values, such that when substituted into the given equation gave opposite polarities. These two values happen to be $x = 19$ (evaluated is $-143$) and $x = 20$ (evaluated is $148$). Next, the lower value, $x = 19$ would be chosen as the initial approximation for the root. Fan-fa would next give the transformation $y = x - 19$. Then the transformation would be written explicitly for $y$ in terms of $x$ ($x = y + 19$). Substitution into the original equation resulted in

$$y^2 + 290y - 143 = 0.$$

Again Chu Shih-chieh would look for two integer values. These two values happen to be $y = 0$ (evaluated is $-143$) and $y = 1$ (evaluated is $148$). The higher value, $y = 1$ would be chosen as the final approximation for the root

$$y = 143/(1 + 290).$$

Fan-fa was not an exclusive mathematical tool for transforming order of equations implemented by just Chu Shih-chieh, as atleast three other mathematicians in the later Sung period are credited with its later use [20]. Up to this point historically, algebra was still presented with multiple unsolved issues. These issues related to roots(negative and irrational roots were never considered and indecision existed if zero was a root), specific presentation of problems sets (general cases were not yet developed) and representation (words and sentences were still primarily used) [68]. As will be shown next, the Indians were successful in dealing with these problems.

Indian mathematics in around the 6th century saw great mathematical advancement in the area of algebra [70]. This advancement is mainly due to Brahmagupta and his text *Extensive Treatise of Brahma* published in 1628. This work covered general solutions to quadratic equations, in which for the first time historically two roots were found and negative roots were considered actual solutions. Irrational roots of numbers were also treated as numbers, unlike previously with prior civilizations. This was a great advancement for mathematics, ironically the result of logical innocence rather than genuine mathematical insight [20]. Brahmagupta is also credited with presenting all integral solutions to indeterminate equations, rather than choose particular solutions as has done previously by Diophantus. Brahmaguptas representation resembled that of the syncopated algebra stage, with addition (juxtaposition), subtraction (dot over subtrahend), division (division below dividend) and multiplication/roots all receiving some form of abbreviation. Bhaskara later filled in some of Brahmaguptas gaps in his mathematical work. In his own treastie *Livavati*, Bhaskara compiled problems of Brahmagupta, which consisted of linear and quadratic equations of determinate and indeterminate types [24]. An example of the linear Diophantine equation solved by Brahmagupta and Bhaskara is presented

$$ax + by = c$$

where $a, b$ and $c$ are three scalars and $x, y$ are two unknowns. Diophantus when presenting a solution to this equation gave only a particular solution. Brahmagupta and Bhaskara are credited with discovering the conditions necessary to determine all integral solutions. To possess an integral solution, the greatest common divisor of $a$ and $b$ would have to divide $c$. All integral solutions are given in the form of

$$x = p + mb\,,\; y = q - ma$$

where $m$ is an integer, and $p, q$ are scalar values to be determined. As will be shown next, the Arabs worked on the issue of specific presentation of problems sets and developed general cases.

Early Arabian mathematics of the 8th century like its Indian counterpart had two famed algebraists. Al-Khowarizmi was the first great Arabian algebraists, his book *Al-jabr wal muabalah* written in 820 is where the origin of the word 'algebra' comes from [24]. Al jabr is translated into Arabic as restoration or completion (subtracting a term on one side of an equation is equal to adding the term to the other side of an equation). Likewise, muabalah translates from Arabic as reduction or balancing (cancelling like terms on opposite sides of an equation) [20]. The six chapters of *Al-jabr wal muabalah* are broken into discussions on squares equals to roots (Ch. 1), squares equal to numbers (Ch. 2), roots equal to numbers (Ch. 3), squares and roots equal to numbers (Ch. 4), squares and numbers equal to roots (Ch. 5) and roots and numbers equal to squares (Ch. 6). Each chapter comprises of three examples illustrating when the coefficient of the variable term was less than, equal to, and greater than one. Consider the following problems given in Chapter 1.

$$\frac{x^2}{3} = 4x \implies x = 12$$

$$x^2 = 5x \implies x = 5$$

$$5x^2 = 10x \implies x = 2$$

The root $x = 0$ was not recognized in any of the problems given. In fact, Al-Khowarizmi rejected negative roots and absolute negative magnitudes, restricting his examples to those containing all positive roots. Al-Khowarizmi has been suggested to be considered the true 'father of algebra', because *Al-jabr wal muqabalah* comes closer to elementary algebra than the work produced by Diophantus [20]. Al-Khowarizmi's work was still mainly rhetorical rather than syncopated. The second known early Arabaian algebraist Omar Khayyam, authored *Algebra* a text that went beyond the works of al-Khowarizmi. *Algebra* included equations of the third degree, and presented both arithmetic and geometric solutions to these equations in a more syncopated manner [24]. Fundamentally, Arabic mathematics stressed the importance of clearly presented arguments in systematic order, from premise statement to concluding remarks.

As was mentioned in the discussion on Babylonian mathematics, medieval mathematics classified quadratic equations into three primary types [24]. Arabian mathematics recognized these three types and presented methods for their solutions. The three types of **quadratic equations** are

$$\text{Type 1: } x^2 + ax = b,$$

$$\text{Type 2: } x^2 + b = ax,$$

$$\text{Type 3: } x^2 = ax + b,$$

where $a, b$ are known scalar values, and $x$ the unknown value. Figure 2.9 displays the Arabian process of geometrically solving Type 1 quadratic equations. Given the equation $x^2 + 10x = 39$, a square of area $x^2$ units is initially drawn. Next, the middle term $10x$ would be split up into four rectangles on each side of the square with an area $2\frac{1}{2}x$ units. Then, to compute the area of the large square, the area of the four corner squares would need to be found. Given each square has dimensions of $2\frac{1}{2} \times 2\frac{1}{2}$ units and their are four squares, the total area of the squares is $25$ units. This value is added to both sides of the original quadratic equation as $x^2 + 10x + 25 = 39 + 25 = 64$. Since the area of the square is now $64$ units, this implies each side has a length of $8$. Thus to find the unknown, $8$ is subtracted from $2 \times 2\frac{1}{2} = 5$ to get $x = 3$. This method known as completing the square was carried out for all three types, with increasing geometric complexity in each case [20]. As will be shown next, the Renaissance produced solutions to equations of higher order (cubic and quartic) and with increased use of abbreviations helped begin the modern age of algebra.

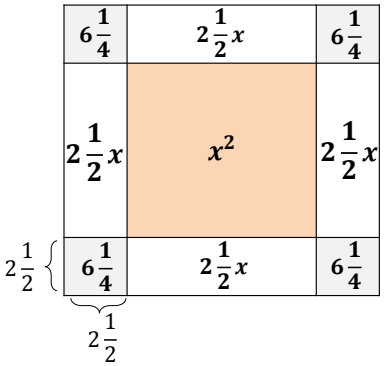



**Figure 2.9:** Arabian Mathematics Example

Renaissance Europe marks the introduction of algebra flourishing regionally rather than culturally. With focus shifting into solving higher order equations using modern mathematical symbology, Renaissance Europe started the symbolic stage in algebra [67]. Germany and Italy were the two primary avenues from which Arabic algebra entered early Renaissance Europe [20, 24]. German mathematician Regiomontanus, is credited with continuing the rhetorical algebra stage, citing influences from early Greek and Arabian mathematics. Johann Widman authored *Rechenung auff allen Kauffmanschafft* (1489) in which the first recognition of the modern plus and minus (+ and –) occurs [24]. Christoph Rudolff in his work *Coss* (1525) is the earliest use of decimal fractions and modern symbols for roots [24]. Michael Stifel in *Arithmetica integra* (1544) used negative coefficients in equations, thus reducing the cases of quadratic equations. Stifel explained the relationships between +/– and roots of quadratic equations. Geronimo Cardano with his work *Ars magna* (1545) marked the beginning of the modern period in mathematics with solution methods for cubic and quartic equations [24]. Up until this point in history, only approximate solutions were capable of being found. In his work, irrational numbers were accepted but not soundly based. As an example, the equation $x^2 = 2$ would be approximated by rational numbers and not solved exactly. Negative numbers were more difficult to express, as they were not readily approximated by positive numbers. As an example, the equation $x + 2 = 0$ was plausible only if considering direction on a number line. Imaginary numbers were avoided in all instances. As an example, the equation $x^2 + 1 = 0$ was deemed unsolvable. Italian mathematics provided symbolic representation for plus and minus ($\bar{\mathrm{p}}$ and $\bar{\mathrm{m}}$) that was utilized by French mathematician Nicolas Chuqet who continued the rhetoric stage with his work *Triparty en la science des nombres* (1484). This work did not resemble any prior work done in arithmetic or algebra, and was the first to present zero and negative exponents alongside positive integral exponents. The equation $5x^4 + 7x^3 - 6x^2 + 3x + 1 + 9x^{-1} - 3x^{-2}$ would be presented as $.5.^4\ \bar{\mathrm{p}}\ .7.^3\ \bar{\mathrm{m}}\ .6.^2\ \bar{\mathrm{p}}\ .3.^1\ \bar{\mathrm{p}}\ .1.^0\ \bar{\mathrm{p}}\ .9.^{1.m.}\ \bar{\mathrm{m}}\ .3.^{2.m.}$.

Like Germany and Italy in early Renaissance Europe, England and France became the two main avenues along which algebra flourished in late Renaissance Europe. English mathematician Robert Recorde with his work *Whetstone of Witte* (1557) first introduced the modern equality sign (=) [24]. Francois Viete author of *De numerosa potestatum resolutione* (1600), provided a novel approach of solving cubic equations and a unique application to Horners method. Transformation of a cubic $x^3 + 3ax = b$ equation occurred by simply introducing a new unknown quantity $y$ related to $x$ such that $y^2 + xy = a$. The application of Horner's method is given a quadratic equation $x^2 + 7x = 60750$

1. Find a lower approximation for $x$: $x = 200 + x_1$,
2. Plug into original equation, reducing roots by 200: $x_1^2 + 407x_1 = 19350$,
3. Find a second approximation for $x$: $x_1 = 40 + x_2$,
4. Plug into modified equation, reducing roots by 40: $x_2^2 + 487x_2 = 1470$,
5. It can be seen that: $x_2 = 3$,
6. Thus: $x_1 = 40 + 3 = 43$ and the root $x = 200 + (40 + 3) = 243$.

Also upset geometry could easily represent all triangles symbolically by ABC, whereas algebra had no physical representation for equation classes, Viete sought universal algebraic symbology. Euclid had once represented magnitudes with letters, however no distinguishable way existed to separate magnitudes from unknown quantities. Viete then decided to represent unknown quantities as vowels and magnitude quantities as consonants. As an example, $BA^2 + CA + D = 0$ could equate to $4x^2 + 3x + 5 = 0$, where $A$ represents the unknown quantity and $B, C$ and $D$ its magnitudes. Girard in *Invention nouvelle en lalgebre* (1629) stated the relationship between roots and coefficients of equations. In it negative roots, roots directed in a opposite sense to positive roots, and imaginary roots were recognized. Girard also realized that an equation can have as many roots as is indicated by the degree of the equation. Girard is also credited with conjuring the fundmental theorem of algebra, which states that every polynomial equation, $f(x) = 0$ having complex coefficients and degree greater than zero has at least one complex root.

Thomas Harriot in *Artis analyticae praxis* (1631) introduced the modern equality signs ($<$ and $>$) and symbolically represented the work of Veite (whose work was solely syncopated). For example $A^3$ would be represented by Harriot as $AAA$ and Veite as $A$ cubus. French mathematician Rene Descartes followed geometrically in the path of Euclid, interested in the application of algebra to geometry in his work *La geometrie* (1637) [24]. This work represents the earliest mathematical text that a person presently could study without having difficulties in notation [20]. In *La geometrie*, parameters are represented by letters at the beginning of the alphabet ($a, b, c, d$, etc.) and unknown quantities from the end of the alphabet($w, x, y, z$, etc.). Unknown quantities are thought purely in the geometrical sense. The quantity $x$ is thought of as a line segment. The quantities $x^2$ and $x^3$ likewise are not thought of as areas and volumes respectively, but rather similarly as line segments. Figure 2.10 geometrically represents the procedure to solve the quadratic equation $x^2 = ax + b^2$.

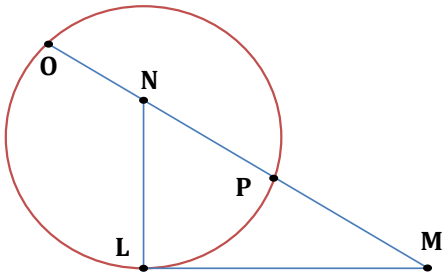



**Figure 2.10:** French Mathematics Example

First, line segment LM of length $b$ is drawn. Next, line segment NL of length $a/2$ is erected perpendicular to LM. With a center at point N, a circle of radius $a/2$ is then constructed. A line starting at point M is then drawn through point N, continuing until it intersects the other side of the circle at point O. The line segments OM and PM become the positive and negative roots of the given quadratic equation. Desecartes never considered PM to be a root because it was negative. Similar geometric results can be derived for the other two primary cases of quadratic equations. As will be shown next, the 19th century brought about the dynamic function and abstract conceptual stages of algebra as well as introduced most of the elementary concepts in linear algebra: linear independence, dimension and vector space.

The golden age of mathematics refers to the 19th century, during which additions to the subject of algebra outweigh both in quantity and quality the aggregate productivity of all preceding ages. George Peacock author of *The Treatise on Algebra* (1830) sought to provide algebra with a logical structure similar in nature to Euclids *Elements*. Peacocks desire to formulate fundamental laws of arithmetic (i.e commutative and associative laws for addition/multiplication and a distributive law for multiplication over addition) marks the beginning of postulational thinking in algebra. Fellow British mathematician Augustus De Morgan worked alongside Peacock to start the stage of abstract algebra through the development of two algebras: single and double algebra [20]. The single algebra represented the real number system, and the double algebra the complex number system. De Morgan though that the two systems exhausted all types of algebra and that fundamental laws developed help true between the two systems. William Hamilton defined another formal algebra of real number couples for the rules of complex numbers, disproving De Morgans thought of just two fundamental algebras. This algebra defined multiplication, which is though as an operation involving rotation, for example as $(a, b)(x, y) = (ax - by, ay + bx)$. Realizing that ordered pairs could be thought of as directed entities (i.e. vectors) in the plane, Hamilton tried to then extend his idea to three dimensions. To accomplish this task, Hamilton extended the two dimensional $a + bi$ to the three dimensional $a + bi + cj$. The definition of addition created no difficulties; however multiplication of n-tuples greater than order two created issues. Ultimately, Hamilton discovered these issues disappeared if quadruples were used instead and the commutative law of multiplication was abandoned. Consider Hamilton's quadruple, $a + bi + cj + dk$ with the following rules

$$i^2 = j^2 = k^2 = -1 = ijk,$$
$$ij = k \text{ and } ji = -k,$$
$$jk = i \text{ and } kj = -i,$$
$$ki = j \text{ and } ik = -j.$$

Hamilton's quadruple began the expansion of mathematics into discussions of algebras that didn't obey traditional laws, and helped paved the introduction of the matrix.

Hermann Grassmann in *Die linale Ausdehnungslehre, ein neuer Zweig der Mathematik* (1844) developed the idea of noncommutaive multiplication. Author Cayley is credited as the first to work with matrices [24]. As an example of these matrices, given the two transformations

$$T_1 : x' = ax + by\ ,\ y' = cx + dy,$$
$$T_2 : x'' = Ax' + By'\ ,\ y'' = Cx' + Dy'.$$

Grassman showed that applying transformation $T_1$ and then transformation $T_2$ results in

$$T_2T_1 : x'' = (Aa + Bc)x + (Ab + Bd)\ ,\ y'' = (Ca + Dc)x + (Cb + Dd)y.$$

Which in matrix representation is

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} Aa + Bc & Ab + Bd \\ Ca + Dc & Cb + Dd \end{pmatrix}$$

Likewise, considering the two transformations switched

$$T_2 : x' = Ax + By\ ,\ y' = Cx + Dy,$$
$$T_1 : x'' = ax' + by\ ,\ y'' = cx' + dy.$$

Now applying transformation $T_1$ and then transformation $T_2$ results in

$$T_1T_2 : x'' = (aA + bC)x + (aB + bD)\ ,\ y'' = (cA + dC)x + (cB + dD)y.$$

Which in matrix representation is

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} aA + bC & aB + bD \\ cA + dC & cB + dD \end{pmatrix}$$

The study of noncommutative algebras (matrix algebra being one of them) has been one of the chief factors in the development of an increasingly abstract view of algebra. With this the historical background into algebra is concluded, and begins discussion of the vector space in linear algebra.

The discussion of a vector space in linear algebra begins with a fundamental building block in mathematics: the mathematical set. A **set** in mathematics is defined as an unordered collection of distinct objects called **elements**. Elements with common characteristics are grouped into sets to establish relationships and aid in mathematical analysis. Overall there exists three types of sets with respect to the number of elements they possess. These sets are referred to as the empty set, the finite set, and the infinite set. **Cardinality** of a set is defined as a measure of the number of elements in a set. To denote cardinality the set is placed between vertical bars, and the **cardinal number** represents the number of elements the set possesses. The **empty set** also referred to as the void or null set, is the set with no elements and is denoted by $\emptyset$. The empty set has cardinality 0, denoted $|\emptyset| = 0$. A **finite set** is a set which possesses a countable number of elements. A finite set $S$ of $n$ elements $a_1, a_2, \ldots, a_n$ is denoted

$$S = \{a_1, a_2, \ldots, a_n\}\,,\ |S| = n\,. \tag{2.1}$$

A **infinite set** is set which possesses an uncountable number of elements. The infinite set $Q$ of $\infty$-many elements $a_1, a_2, \ldots$ is denoted

$$Q = \{a_1, a_2, \ldots\}\ |Q| = \infty\,. \tag{2.2}$$

Overall, there exists only one empty set. In which case, we say the empty set is a **unique** set. Likewise, there exists many such finite and infinite sets, thus they are not generally unique sets. The **ellipsis** $(\ldots)$ is used for shorthand notation, and represent the elements in a set which exist but are not formally written out. To express an **element** $a$ **as a member of a set** $S$, the following notation is introduced $a \in S$ and read $a$ in $S$. Similarly, to express that an **element** $a$ **does not belong to a set** $S$, the notation $a \notin S$ is introduced and read $a$ not in $S$. The construction of a set is **immaterial**, in that element order or repetition numbers do not distinguish two sets as unique. As an example, the following representations of the finite set $S$

$$S = \{1, 2, 3\} = \{2, 3, 1\} = \{1, 1, 2, 3\}$$

are all equivalent. In all cases, the elements $1, 2$ and $3$ are contained in the finite set $S$, that is $1, 2, 3 \in S$. Also, the size of set $S$ is equal to the number of unique elements in $S$, that is $|S| = 3$.

Certain sets appear frequently in mathematics, and hence have distinct names and representation. These special sets shown in Table 2.1, consist of the universal set $U$ and the sets of number systems $(\mathbb{C}, \mathbb{N}, \mathbb{I}, \mathbb{R}, \mathbb{Z})$ [65]. The universal set is a common example of a finite set, whereas the sets of number systems are common examples of infinite sets.

**Table 2.1:** Common Sets

| Symbol | Set Notation | Set Name |
|---|---|---|
| $U$ | $\{a \mid a \in U\}$ | Universal Set |
| $\mathbb{C}$ | $\{a + bi \mid a, b \in \mathbb{R}, i^2 = -1\}$ | Complex Numbers |
| $\mathbb{N}$ | $\{1, 2, 3, \ldots\} = \mathbb{Z}^+$ | Natural Numbers |
| $\mathbb{I}$ | $\{a \mid a \notin \mathbb{Q}\}$ | Irrational Numbers |
| $\mathbb{R}$ | $\{a + bi \mid a \in \mathbb{R}, b = 0\}$ | Real Numbers |
| $\mathbb{Q}$ | $\{a/b \mid a, b \in \mathbb{Z}, b \neq 0\}$ | Rational Numbers |
| $\mathbb{Z}$ | $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ | Integers |

The universal set $U$ is very useful when given a system of multiple sets and tasked with performing set operations on them. While single sets have been introduced for definitional purposes thus far, it is quite rare to consider only a single set when performing any given analysis. To show the application of the universal set and set operations (union and intersection) consider the four sets

$$U = \{-5, -1, 0, 1, 2, 3, 8, 11, 14\}\,, |U| = 9\,,$$

$$T = \{-5, 1, 2, 3, 11\}\,, |T| = 5\,,$$

$$V = \{-1, 0, 2, 3, 8, 11, 14\}\,, |V| = 7\,,$$

$$W = \{2, 3, 11\}\,, |W| = 3\,.$$

Set $U$ by being the universal set of the system of sets $T$ and $V$, possesses the elements present in either set $T$ or set $V$. Set $U$ is termed the **union** of sets $T$ and $V$, denoted $T \cup V$. Likewise, set $W$ possesses only the elements present in both sets $T$ and $V$, is defined as the **intersection** of sets $T$ and $V$, denoted $T \cap V$. Also, set $S = \{1, 2, 3\}$ defined previously possesses all the elements of set $T$. In such an occurrence it is said set $S$ is a **subset** of set $T$, denoted $S \subset T$. In all cases, the intersection of a system of sets will be a subset of the union of that same system. These three base set operations just presented are all that is required to comprehend this work in regards to set operations. Table 2.2 outlines these three properties for general sets $S$ and $T$, all of which are commutative, associative and distributive [122, 131].

**Table 2.2:** Set Properties and Rules

| Set Notation | Set Property |
|---|---|
| $S \cup T = \{a \mid a \in S \text{ or } a \in T\}$ | Union of sets $S$ and $T$ |
| $S \cap T = \{a \mid a \in S \text{ and } a \in T\}$ | Intersection of sets $S$ and $T$ |
| $S \subset T = \{a \mid a \in S \implies a \in T\}$ | Set $S$ is a subset of set $T$ |

Mathematical sets provide a method of grouping elements together based on shared characteristics. The mathematical analysis into sets goes much deeper than what was just presented. Set theory is a realm of mathematics focused on this [129]. Set theory is by construction an abstract analysis, whereas the information introduced above on sets is for the applied analysis this work covers. From the mathematical set, its properties and operations another fundamental building block in mathematics is now formally introduced. Given a set, a corresponding algebraic structure exists. An **algebraic structure** is a unique set that obeys certain operations, for which all elements of that set possess. In linear algebra the two primary algebraic structures are the field and vector spaces. The definition of the matrix follows immediately from these two algebraic structures.

**Field** $F$ is an algebraic structure that possesses the ten properties listed in Table 2.3 [52].

**Table 2.3:** Field Properties

| | | |
|---|---|---|
| F. 1 | $\forall\, a, b \in F,\ a + b = b + a$ | Commutativity of Addition |
| F. 2 | $\forall\, a, b \in F,\ a \cdot b = b \cdot a$ | Commutativity of Multiplication |
| F. 3 | $\forall\, a, b, c \in F,\ a + (b + c) = (a + b) + c$ | Associativity of Addition |
| F. 4 | $\forall\, a, b, c \in F,\ a \cdot (b \cdot c) = (a \cdot b) \cdot c$ | Associativity of Multiplication |
| F. 5 | $\exists!\ 0 \in F$ s.t. $\forall\, a \in F,\ a + 0 = a = 0 + a$ | Identity Element of Addition |
| F. 6 | $\exists!\ 1 \in F$ s.t. $\forall\, a \in F,\ a \cdot 1 = a = 1 \cdot a$ | Identity Element of Multiplication |
| F. 7 | $\forall\, a \in F,\ \exists\, c \in F$ s.t. $a + c = 0$ | Inverse Element of Addition |
| F. 8 | $\forall\, b \in F\ (b \neq 0),\ \exists\, d \in F$ s.t. $b \cdot d = 1 = d \cdot b$ | Inverse Element of Multiplication |
| F. 9 | $\forall\, a, b, c \in F,\ a \cdot (b + c) = a \cdot b + a \cdot c$ | Left Multiplication over Addition |
| F. 10 | $\forall\, a, b, c \in F,\ (b + c) \cdot a = b \cdot a + c \cdot a$ | Right Multiplication over Addition |

Elements of the field $F$, such as $a, b$ and $c$ are denoted as **scalars**. These scalar elements come from either the set of real numbers, $\mathbb{R}$ or set of complex numbers, $\mathbb{C}$ presented in Table 2.1 [121]. The set of scalars is a subset of the set of vectors since all vectors are constructed of scalar values with associated directions. The identity elements in $F$ for addition, $0$ and multiplication, $1$ are unique. They are the only elements which satisfy properties F. 5 and F. 6 in Table 2.3. A field guarantees commutativity of multiplication and the existence of an inverse element under multiplication. However, a field does not list distribution of addition over multiplication as one of its defining properties. To present why distribution over multiplication is not guaranteed to hold, consider the case when $a = 1$, $b = 3$ and $c = 4$

$$(a \cdot b) + c = (1 \cdot 3) + 4 = 7 \neq 16 = (1 + 3) \cdot 4 = (a + b) \cdot c$$

Field elements are numbers people associate with normally when performing everyday arithmetic. These elements come from $\mathbb{R}$ and $\mathbb{C}$ because the other sets of number systems either violate field properties (i.e. $\mathbb{N}$ with F. 7, $\mathbb{Z}$ with F. 8) or are subsets (i.e. $\mathbb{I}$ , $\mathbb{Q} \subset \mathbb{R}$).

**Vector space** $V$ is an algebraic structure that possesses the eight properties listed in Table 2.4.

**Table 2.4:** Vector Space Properties

| | | |
|---|---|---|
| VS. 1 | $\forall \boldsymbol{x}, \boldsymbol{y} \in V,\ \boldsymbol{x} + \boldsymbol{y} = \boldsymbol{y} + \boldsymbol{x}$ | Commutativity of Addition |
| VS. 2 | $\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in V,\ \boldsymbol{x} + (\boldsymbol{y} + \boldsymbol{z}) = (\boldsymbol{x} + \boldsymbol{y}) + \boldsymbol{y}$ | Associativity of Addition |
| VS. 3 | $\exists!\ \boldsymbol{0} \in V$ s.t. $\forall \boldsymbol{x} \in V,\ \boldsymbol{x} + \boldsymbol{0} = \boldsymbol{x} = \boldsymbol{0} + \boldsymbol{x}$ | Identity Element of Addition |
| VS. 4 | $\forall \boldsymbol{x} \in V,\ \exists \boldsymbol{y} \in V$ s.t. $\boldsymbol{x} + \boldsymbol{y} = \boldsymbol{0}$ | Inverse Element of Addition |
| VS. 5 | $\exists!\ \boldsymbol{1} \in V$ s.t. $\forall \boldsymbol{x} \in V,\ \boldsymbol{1x} = \boldsymbol{x}$ | Identity Element of Multiplication |
| VS. 6 | $\forall a, b \in F$ and $\forall \boldsymbol{x} \in V,\ (ab)\boldsymbol{x} = a(b\boldsymbol{x})$ | Associativity of Multiplication |
| VS. 7 | $\forall a \in F$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in V,\ a(\boldsymbol{x} + \boldsymbol{y}) = a\boldsymbol{x} + a\boldsymbol{y}$ | Distribution over Vector Addition |
| VS. 8 | $\forall a, b \in F$ and $\forall \boldsymbol{x} \in V,\ (a + b)\boldsymbol{x} = a\boldsymbol{x} + b\boldsymbol{x}$ | Distribution over Scalar Addition |

Elements of vector space $V$, such as $\boldsymbol{x}, \boldsymbol{y}$ and $\boldsymbol{z}$ are **vectors**. Vectors are given bold representation to distinguish them from unbolded scalar elements. Association, commutation, the existence of identity and inverse elements, and distribution are standard algebraic structure properties vector spaces possess. The identity elements in $V$ for addition, $0$ and multiplication $1$ are unique. They are the only elements that can satisfy properties VS. 3 and VS. 5 in Table 2.4. Certain sets of properties in vector spaces are mirrored for addition and multiplication (associative, identity element, distribution over addition) but not for others (commutative, inverse element, distribution over multiplication). Section 1.1.2 will explain why the commutative property of multiplication in general does not hold in a vector space $V$. Section 1.2.2 will present why an identity element under multiplication may not exist in a vector space $V$. To present why distribution over multiplication never holds, consider the two vectors $\boldsymbol{x} = (1, 2), \boldsymbol{y} = (0, 1)$ in $V$ and the scalar $a = 3$ in $F$

$$(\boldsymbol{x} \cdot \boldsymbol{y}) + a = [(1)(0) + (2)(1)] + 3 = 5 \neq (3, 9) = (1, 3) \cdot 3 = (1 + 0, 2 + 1) = (\boldsymbol{x} + \boldsymbol{y}) \cdot a$$

The two quantities $(\boldsymbol{x} \cdot \boldsymbol{y}) + a$ and $(\boldsymbol{x} + \boldsymbol{y}) \cdot a$ are never comparable and never equate as they produce different quantity types (scalar and vector, respectively). We are now ready to introduce the matrix.

A **matrix** $M$ is defined as a rectangular array of numbers (or functions) enclosed within brackets. These numbers (or functions) are called the **elements** (or entries) of the matrix [121]. Elements of the matrix $M$ are scalars from the field $F$. Elements can be arranged in horizontal arrays called **rows**, and vertical arrays called **columns**. The horizontal and vertical arrays of a matrix are both vectors in the vector space $V$. Thus, the fundamental relationship between the algebraic structures (field and vector spaces) and a matrix is established. A matrix will now be presented in both standard format (emphasis placed on field scalars) and array format (emphasis placed on vector space vectors). Matrix $M$ with $m$ rows and $n$ columns is first presented in **standard format** as

$$M = \begin{array}{c} \\ i=1 \\ i=2 \\ i=3 \\ \vdots \\ i=m \end{array} \begin{array}{c} \begin{array}{ccccc} j=1 & j=2 & j=3 & \dots & j=n \end{array} \\ \begin{bmatrix} m_{11} & m_{12} & m_{13} & \cdots & m_{1n} \\ m_{21} & m_{22} & m_{23} & \cdots & m_{2n} \\ m_{31} & m_{32} & m_{33} & \cdots & m_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{m1} & m_{m2} & m_{m3} & \cdots & m_{mn} \end{bmatrix} \end{array}, |M| = m \times n \,. \tag{2.3}$$

**Indicies** $i$ and $j$ are used for shorthand notation of rows and columns of matrix $M$, respectively. The elements of a matrix are typically presented in double-subscript notation. **Double-subscript notation** presents row numbers as the first subscript and column numbers as the second subscript. As an example, $m_{12}$ represents the element in the first row and second column of matrix $M$. The size of a matrix is determined by its dimensions. The **dimensions** represent the number of rows and number of columns in a matrix. Matrix $M$ with $m$ rows and $n$ columns has size $|M| = m \times n$. The diagonal of the matrix $M$ consisting of elements with the first and second subscript equal, is called the **main (or principle) diagonal** with elements $m_{11}, m_{22}, m_{33}, \ldots, m_{mn}$. Special matrices such as banded, identity, and tridiagonal matrices have special properties related to the main diagonal.

Matrix $M$ represented in **array format** as the set of all $n$-tuple column vectors $M_j$ , $1 \leq j \leq n$

$$M = \left[ \; M_1 \, , \, M_2 \, , \, M_3 \, , \, \ldots \, , \, M_n \; \right] , |M| = m \times n. \tag{2.4}$$

Similarly, changing the indicie to $1 \leq i \leq m$ matrix $M$ can be represented as the set of all $m$-tuple row vectors $M_i$. Standard representation of matrices in array format are given in column vectors. A single row and column of matrix $m$ are denoted as a row $M_i$ and column vector $M_j$, respectively [52, 121]. The $i$-th row of $M$ is given as the **horizontal array (row vector)**

$$M_i = \left[ \; m_{i1} \, , \, m_{i2} \, , \, m_{i3} \, , \, \ldots \, , \, m_{in} \; \right] , |M_i| = 1 \times n. \tag{2.5}$$

Likewise, the $j$-th column of $M$ is given as the **vertical array (column vector)**

$$M_j = \begin{bmatrix} m_{1j} \\ m_{2j} \\ m_{3j} \\ \vdots \\ m_{mj} \end{bmatrix} , |M_j| = m \times 1. \tag{2.6}$$

A **square matrix** is a special rectangular array in which the number of rows and number of columns is the same. Thus, if $M$ is a square matrix then $m = n$, $|M| = n \times n$ is and said to have order $n$. Certain matrix operations such as matrix inversion, eigenvalue problems, and determinant computations only hold if the matrix is square. For this background chapter, if not stated the matrices used will have the standard $m$ rows and $n$ columns. For the selected cases that require square matrices, matrices will then have $n$ rows and $n$ columns. Also, additional properties regarding matrices will be introduced as they relate to the matrix operations and applications. Next the two main matrix operations are introduced. Matrices in these sections are referred to as base and resultant matrices, respectively. A **base matrix** is a standard matrix, that is used to perform the matrix operations, and a **resultant matrix** is a matrix resulting from the matrix operations performed.

## 1.1 Matrix Operations

### 1.1.1 Addition

*Mathematical Overview.* Given the $m \times n$ base matrix $A$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, |A| = m \times n\,, \tag{2.7}$$

and $m \times n$ base matrix $B$

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ a_{21} & a_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}, |B| = m \times n\,. \tag{2.8}$$

The **addition** of base matrices $A$ and $B$ is defined as the resultant matrix $C$

$$C = A + B = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{bmatrix}, |C| = m \times n\,. \tag{2.9}$$

Matrix addition has only one requirement for resultant matrix $C$ to exist. This sole requirement is that the base matrices $A$ and $B$ must have equivalent dimensions. Since the number of rows, $m$, and number of columns, $n$, in base matrices $A$ and $B$ are the same, the resultant matrix $C$ is defined. Under these conditions, base matrices $A$ and $B$ are said to be **additively comfortable** [121]. Once defined, resultant matrix $C$ by definition possesses the dimensions of its defining base matrices. This demonstrates the reason behind resultant matrix $C$ having size $m \times n$. While the resultant

matrix $C$ is currently only defined as the addition of two matrices, the principle of matrix addition can be extended to any finite number of base matrices. Taking a finite number $k \in \mathbb{N}$, the resultant matrix $C$ can be expressed as

$$C = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} + \cdots + \begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ k_{m1} & k_{m2} & \cdots & k_{mn} \end{bmatrix}. \quad (2.10)$$

Example base matrices $A^*$ in reference equation (A.1) and $B^*$ in reference equation (A.2) from Appendix A are now introduced to display the computation of matrix addition. To define resultant matrix $C^*$, the dimensions of the base matrices $A^*$ and $B^*$ must first be checked for equivalence. From the definition of matrix addition, since $|A^*| = |B^*| = 3 \times 3$ the resultant matrix $C^*$ exists. Being that $A^*$ and $B^*$ are additively comfortable, it then follows that $|C^*| = 3 \times 3$. The elements of the resultant matrix $C^*$ are defined as the addition of corresponding elements in base matrices $A^*$ and $B^*$

$$C^* = A^* + B^* = \begin{bmatrix} 1+3 & -2+0 & 0+2 \\ 2+1 & -6+(-4) & 3+3 \\ 0+2 & 1+(-8) & 1+6 \end{bmatrix} = \begin{bmatrix} 4 & -2 & 2 \\ 3 & -10 & 6 \\ 2 & -7 & 7 \end{bmatrix}, |C^*| = 3 \times 3 .$$

Table 2.5 outlines common properties associated with matrix addition [65].

**Table 2.5:** Matrix Addition Properties

| | | |
|---|---|---|
| M.A. 1 | $A + B = B + A$ | Commutative Property |
| M.A. 2 | $(A + B) + C = A + (B + C)$ | Associative Property |
| M.A. 3 | $A + 0 = A = 0 + A$ | Additive Identity |
| M.A. 4 | $A + B = 0 \iff B = -A$ | Additive Inverse |

For additively comfortable base matrices, matrix addition obeys the commutative and associative laws in properties M.A. 1 and M.A. 2, respectively. M.A. 1 states swapping the **operands** (base matrices $A$ and $B$) with respect to the **operator** (+) does not change the resulting value. Property M.A. 2 states grouping the operands with respect to the operator does not change the resulting value. Note, base matrix $C$ in property M.A. 2 is any matrix with the same size of base matrices $A$ and $B$, and not necessarily the resultant matrix found in Eq. (2.14). The additive identity matrix $0$ and additive inverse matrix $B$ introduced in properties M.A. 3 and M.A. 4, respectively, are unique matrices. Note that the additive inverse matrix $B$ introduced in property M.A. 4 is any matrix with the same size of base matrix $A$, and not necessarily the base matrix found in Eq. (2.8). To compute $B = -A$ in property M.A. 4, the relationship between matrix addition and subtraction must first be established as follows. Given the base matrices $A$ in (2.7) and $B$ in (2.8), the **subtraction** of base matrix $B$ from base matrix $A$ is defined as the resultant matrix $D$ as [127]

$$D = A - B = \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1n} \\ d_{21} & d_{22} & \cdots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{m1} & d_{m2} & \cdots & d_{mn} \end{bmatrix}, |D| = m \times n\,, \tag{2.11}$$

where $-B$ is the multiplication of a scalar value $-1$ with the base matrix $B$. This multiplication simply changes the **polarity (or sign)** of every element in base matrix $B$. If an element in $B$ had a value of $3$, introducing a change in polarity would result the same element in $-B$ becoming $-3$. Or if the element in $B$ had a value of $-3$, introducing a change in polarity would result the same element in $-B$ becoming $3$. In other words, matrix subtraction can be thought of the addition of a base matrix $A$ with base matrix $B$ having opposite signs, expressed $A - B = A + (-1)B$.

Example base matrices $A^*$ in reference equation (A.1) and $B^*$ in reference equation (A.2) are now reintroduced to display the computation of matrix subtraction. To define the resultant matrix $D^*$, the dimensions of the base matrices $A^*$ and $B^*$ must first be checked for equivalence.

Similar to matrix addition, since $|A^*| = |B^*| = 3 \times 3$ the resultant matrix $D^*$ exists. Being that $A^*$ and $B^*$ are additively comfortable, it then follows that $|D^*| = 3 \times 3$. The elements of the resultant matrix $D^*$ are defined as the addition of elements in base matrix $A^*$ and base matrix $B^*$ with its sign of polarity changed. The elements of resultant matrix $D$ are then

$$D^* = A^* - B^* = \begin{bmatrix} 1-3 & -2-0 & 0-2 \\ 2-1 & -6-(-4) & 3-3 \\ 0-2 & 1-(-8) & 1-6 \end{bmatrix} = \begin{bmatrix} -2 & -2 & -2 \\ 1 & -2 & 0 \\ -2 & 9 & -5 \end{bmatrix}, |D^*| = 3 \times 3\,.$$

*Numerical Overview.* The algorithm to perform matrix addition and compute the elements of $C$ is presented in Sigma notation as well as expansion notation in Equation (2.12). **Sigma notation** represents the summation of the indices ($i$ and $j$) starting from their initial values $i = 1$ and $j = 1$, and increasing by one until reaching their terminal values $i = m$ and $j = n$. Common shorthand notation to represent ranging indexes is $1 \leq i \leq m$ and $1 \leq j \leq n$. **Expansion notation** is presented in equation (2.12) to the right of Sigma notation. The ellipses in the expansion notation represents $i = 3, 4, 5 \ldots m-1$ and $j = 3, 4, 5 \ldots n-1$. Plugging $i = 1, 2$ and $m$ and $j = 1, 2$ and $m$ generates the elements shown in equation (2.12). Sigma notation is a more compact representation of the expansion notation with ellipses and easier to manipulate mathematically. Expansion notation is presented solely to give the reader a visual understanding of the algorithm.

$$c_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij} + b_{ij}) = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \ldots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \ldots & a_{2n} + b_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \ldots & a_{mn} + b_{mn} \end{bmatrix} \tag{2.12}$$

A **FLOP** is defined as a single addition, subtraction, multiplication or division applied to two matrix elements. FLOPS represent the minimum number of operations required to evaluate an algorithm. The FLOPS (floating point operations) required to compute $C$ is given as $mn$ [121].

### 1.1.2 Multiplication

*Mathematical Overview.* Given base matrices $A$ and $B$ in Equations (2.7) and (2.8) respectively, the **multiplication** of $A$ and $B$ is defined as the resultant matrix $F$

$$F = AB = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f_{m1} & f_{m2} & \cdots & f_{mn} \end{bmatrix}, |F| = m \times n. \tag{2.13}$$

The only requirement to perform matrix multiplication is inner dimensions of the multiplied base matrices must be equal [52]. From Equations (2.7) - (2.8) the size of $A$ and $B$ are both $m \times n$. For $F$ to be well-defined, the dimensions of base matrices must be modified so that the inner dimension of $A$ equals the inner dimension of $B$. To accomplish this, dimensions are redefined on matrix $A$ as $m \times p$ and $B$ as $p \times n$, so that resultant $m \times n$ matrix $F$ becomes **multiplicatively comfortable** [121].

$$\text{Before Modification: } AB = (m \times \mathbf{n}) \cdot (\mathbf{m} \times n)\,,\ n \neq m\,,\ F \text{ undefined}$$

$$\text{After Modification: } AB = (m \times \mathbf{p}) \cdot (\mathbf{p} \times n)\,,\ p = p\,,\ |F| = m \times n$$

The definition of matrix **inversion** (division) is derived from matrix multiplication, $A/B = AB^{-1}$. While the definition of $F$ is currently only the multiplication of two base matrices, the principle of matrix multiplication can be extended to any finite number of matrices. Taking a finite number $k \in \mathbb{N}$, the resultant matrix $F$ can be expressed as

$$F = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} \cdot \cdots \cdot \begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ k_{m1} & k_{m2} & \cdots & k_{mn} \end{bmatrix}. \tag{2.14}$$

Table 2.6 outlines common properties associated with matrix multiplication [121]. Properties MM.

**Table 2.6:** Matrix Multiplication Properties

| | | |
|---|---|---|
| MM. 1 | $a(A+B) = aA + aB$ | Distribution of Scalar Multiplication |
| MM. 2 | $(a+b)A = aA + bA$ | Distribution of Matrix Multiplication |
| MM. 3 | $(ab)A = a(bA)$ | Associativity of Scalar/Matrix Multiplication |
| MM. 4 | $IA = A = AI$ | Identity Element of Multiplication |
| MM. 5 | $A_iB = F_i$ | Row Vector $\times$ Matrix = Row Vector |
| MM. 6 | $AB_j = F_j$ | Matrix $\times$ Column Vector = Column Vector |
| MM. 7 | $A_iB_j = f_{ij}$ | Row Vector $\times$ Column Vector = Scalar |
| MM. 8 | $A_jB_i = F$ | Column Vector $\times$ Row Vector = Matrix |
| MM. 9 | $AB \neq BA$ | Matrix Multiplication is not Commutative |
| MM. 10 | $AB = 0$ doesn't imply $A = 0$ or $B = 0$ | Zero product doesn't guarantee zero matrices |
| MM. 11 | $AB = AC$ doesn't imply $B = C$ | Cancellation Law doesn't always hold |

1 and MM. 2 are the distribution of scalar multiplication over matrix addition and distribution of matrix multiplication over scalar addition, respectively. Property MM. 3 states grouping scalars with respect to a matrix doesn't change its value. Property MM. 4 states the identity matrix is the identity element of multiplication. The **identity matrix** is defined by the Kronecker delta [52, 65]

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}, \quad |I| = m \times n \tag{2.15}$$

Properties MM. 5-8 represent multiplication combinations which are useful in determining outputs. For example, if a dataset is stored in row vector and the desired output is a scalar value, using MM. 7 one must store an additional dataset in a column vector and postmultiply it by the first dataset. In general given two base matrices multiplied together as $AB$, we say $A$ is **premultiplied (or multipled by the left)** to $B$ or that $B$ is **postmultiplied (or multiplied by the right)** to $A$.

Properties MM. 9-11 represent three properties that hold for elements in a field, but not in general for matrix multiplication. Counterexamples for these properties are shown accordingly below.

$$\text{MM. 9: } \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 5 \\ 3 & 8 \end{bmatrix} \neq \begin{bmatrix} 2 & 3 \\ 5 & 8 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$$

$$\text{MM. 10: } \begin{bmatrix} 0 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \text{ but } \begin{bmatrix} 0 & 1 \\ 0 & 2 \end{bmatrix} \neq \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \text{ or } \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \neq \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\text{MM. 11: } \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 3 & 1 \end{bmatrix} \text{ but } \begin{bmatrix} 0 & 1 \\ 0 & 2 \end{bmatrix} \neq \begin{bmatrix} 0 & 1 \\ 3 & 1 \end{bmatrix}$$

The same base matrices $A^*$ and $B^*$ from Appendix A are reintroduced to display the computation of matrix multiplication. To define resultant matrix $F^*$, the inner dimension of $A^*$ must equal the inner dimension of $B^*$. Since $A^*$ has 3 columns and $B^*$ has 3 rows, the resultant matrix $F^*$ exists. Next, to compute the first column of $F^*$ horizontally traverse the first row of $A^*$ and vertically traverse the first column of $B^*$, multiplying the respective elements and adding their sum. This procedure is shown below. Repeat this procedure to generate the second and third columns of $F^*$.

$$F^* = A^*B^* = \begin{bmatrix} 1(3) - 2(1) + 0(2) & 8 & -4 \\ 2(3) - 6(1) + 3(2) & 0 & 4 \\ 0(3) + 1(1) + 1(2) & -12 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 8 & -4 \\ 6 & 0 & 4 \\ 3 & -12 & 9 \end{bmatrix}, |F^*| = 3 \times 3\,.$$

This example presents the case of multiplication of two base square matrices, which always produce a resultant square matrix with the same dimensions. Note, it is not a requirement in matrix multiplication for the outer dimensions of the base matrices being multiplied to be equal in value. In general they are not, this just means they are not square matrices.

*Numerical Overview*: Equation 2.16 is the algorithm for matrix multiplication and computing $F$. Resultant matrix $F$ is the multiplication of components of base matrices $A$ and $B$, as $A$ moves from left to right and $B$ moves from top to bottom.

$$f_{ij} = \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{p} a_{ik}b_{kj} = \begin{bmatrix} a_{11}b_{11} + \ldots + a_{1p}b_{p1} & \ldots & a_{11}b_{1n} + \ldots + a_{1p}b_{pn} \\ a_{21}b_{11} + \ldots + a_{2p}b_{p1} & \ldots & a_{21}b_{1n} + \ldots + a_{2p}b_{pn} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \ldots + a_{mp}b_{p1} & \ldots & b_{m1}b_{1n} + \ldots + a_{mp}b_{pn} \end{bmatrix} \tag{2.16}$$

**Order of magnitude** is another representation used to denote the number of operations required to evaluate an algorithm. The order of magnitude to compute $F$ is given as $O(n^3)$. To see how this value is computed, consider the following. To compute element $f_{11}$, it will require $p$ multiplications and $p-1$ additions, or $2p-1$ total FLOPS. To compute the first row vector, $F_1$ the number of FLOPS to compute that element will be have to be multiplied by the number of columns, $n$. This brings the total FLOP count up to $n(2p-1)$. To compute all the rows, the number of FLOPS to compute a single row will have to be multiplied by the number of rows, $m$. This brings the total FLOP count to $mn(2p-1)$. Expanding the total FLOP count out, $2mnp - mn$. As the size of the matrices becomes large, the second term $-mn$ becomes computationally irrelevant relative to the larger term $2mnp$. Considering the case when $m = p = n$, the order of magnitude becomes $O(n^3)$. The constant $2$ gets dropped because as $n$ gets very large, the constant term becomes computationally irrelevant. To see this, consider the case when $m = 1,000$, $n = 1,000$, and $p = 1,000$. The term $2mnp = 2$ x$10^9$, whereas the term $-mn = -1$ x$10^6$. The second term is already an order of $1000$ less than the first term, and this will only magnify as the matrices being multiplied increase in size. Computationally, matrix multiplication is more complicated compared to matrix addition. Special algorithms have been developed to efficiently perform matrix multiplication when the matrices being multiplied are dense and sparse [9, 60]. These algorithms for matrix multiplication with dense and sparse matrices will be discussed in the following sections on matrix types.

## 1.2 Matrix Applications

### 1.2.1 Determinant

*Mathematical Overview* : Given base matrix $A$, the **determinant** of $A$ is defined implicitly as

$$\det(A) = \frac{\text{adj}(A)}{A^{-1}}, \tag{2.17}$$

where $\text{adj}(A)$ is termed the **classical adjoint** (or **adjugate**) **matrix** of $A$, and $A^{-1}$ is termed the **inverse matrix** of $A$. The classical adjoint matrix develops by removing the $i^{th}$ row and $j^{th}$ column from matrix $A$. Section 1.2.2 begins the discussion into matrix inversion and how to compute $A^{-1}$. The only requirement necessary to compute a matrix determinant is the matrix must be square. Table 2.7 outlines common properties associated with the determinant of a matrix [52, 65].

**Table 2.7:** Determinant Properties

| | | |
|---|---|---|
| D. 1 | $\det(A) = \det(A^T)$ , $\det(A^{-1}) = [\det(A)]^{-1}$ | Transpose/Inverse Properties |
| D. 2 | Two identical rows/columns $\implies$ $\det(A) = 0$ | Zero Determinant Property |
| D. 3 | Row/column of all zero entries $\implies$ $\det(A) = 0$ | Zero Determinant Property |
| D. 4 | $\det(AB) = \det(A)\det(B)$ | Multiplication Property |
| D. 5 | Interchange rows/columns: $\det(B) = -\det(A)$ | Type 1 Operation Property |
| D. 6 | Multiply a row/column by scalar: $\det(B) = k\det(A)$ | Type 2 Operation Property |
| D. 7 | Add multiple of a row/column: $\det(B) = \det(A)$ | Type 3 Operation Property |
| D. 8 | $\det(kA) = k^n\det(A)$ | Scalar Factor Property |
| D. 9 | $\det(A) = 0 \implies A^{-1}$ doesn't exist | Inverse Property |

Property D. 1 states the property of the determinant is independent of matrix transposition, and the determinant of a inverse matrix is equal to the inverse of the determinant matrix. Properties D. 2 and D. 3 state two conditions that guarantee a numerical zero determinant. Property D. 4 states the determinant of a matrix product is equal to the product of the determinants. Properties D. 5 through D. 7 correspond to the three types of Elementary Matrix Operations. Section 1.2.2 discusses these properties. Property D. 8 states if a matrix is multiplied by a scalar $k$, its resulting determinant will

equate to the product of that scalar raised to the matrix size, $n$ and the matrix determinant $\det(A)$. Property D. 9 states the relationship between the zero determinant and inverse matrix.

*Numerical Overview* : The explicit definition and algorithm for $\det(A)$ [52]

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det(\widetilde{A}_{ij}) = \sum_{i=1}^{n} (-1)^{i+j} a_{ij} \det(\widetilde{A}_{ij}), \tag{2.18}$$

where $\det(\widetilde{A}_{ij})$ is termed the **minor** of $a_i j$, and $(-1)^{i+j} \det(\widetilde{A}_{ij})$ is termed the **cofactor** $c_{ij}$ of $a_{ij}$ [121]. The minor is found by removing the $i^{th}$ row and $j^{th}$ column from matrix $A$, and proceeding to take the determinant of the modified matrix $\widetilde{A}_{ij}$. The order of magnitude of this algorithm is represented recursively as $O(n(O_{n-1} + 2) - 1))$ [59]. The explicit definition of $\det(A)$ can easily be expanded. For example, expansion along the first row of matrix $A$ results in

$$\det(A) = a_{11} \begin{pmatrix} a_{22} & a_{23} & \cdots & a_{2n} \\ a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} + \cdots \pm (-1)^{1+n} a_{1n} \begin{pmatrix} a_{21} & a_{22} & \cdots & a_{2(n-1)} \\ a_{31} & a_{32} & \cdots & a_{3(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n(n-1)} \end{pmatrix}. \tag{2.19}$$

A geometric interpretation of the determinant is now presented. Figure 2.11 displays two vectors $\boldsymbol{a} = (a_1, a_2)$ and $\boldsymbol{b} = (b_1, b_2)$ in the 2D coordinate plane. The determinant is the area of the parallelogram formed between the two vectors. In three dimensions, vectors $\boldsymbol{a} = (a_1, a_2, a_3)$ and $\boldsymbol{b} = (b_1, b_2, b_3)$ would compose a volume [121].

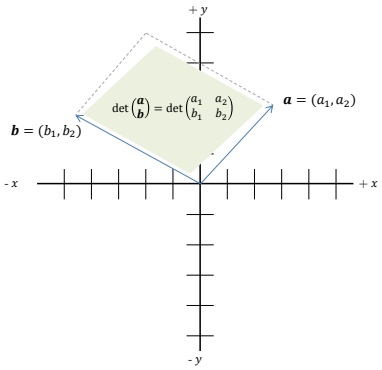



**Figure 2.11:** Geometric Determinant

Consider example base matrix $C^*$ in Reference Equation A.3

$$C^* = \begin{bmatrix} 0 & 2 & 1 \\ 3 & -1 & 2 \\ 4 & 0 & 1 \end{bmatrix}$$

Starting with element $c^*_{11}$, delete the first row and first column of $C^*$ to find minor $M_{11}$

$$\begin{bmatrix} \mathbf{0} & 2 & 1 \\ 3 & -1 & 2 \\ 4 & 0 & 1 \end{bmatrix} \implies M_{11} = \det \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix} = (-1)(1) - (0)(2) = -1$$

Again with element $c^*_{12}$, delete the first row and second column of $C^*$ to find minor $M_{12}$

$$\begin{bmatrix} 0 & \mathbf{2} & 1 \\ 3 & -1 & 2 \\ 4 & 0 & 1 \end{bmatrix} \implies M_{12} = \det \begin{bmatrix} 3 & 2 \\ 4 & 1 \end{bmatrix} = (3)(1) - (2)(4) = -5$$

Continue this procedure to compute all minors, shown below.

$$M_{11} = -1\,,\ M_{12} = -5\,,\ M_{13} = 4\,,$$

$$M_{21} = 2\,,\ M_{22} = -4\,,\ M_{23} = -8\,,$$

$$M_{31} = 5\,,\ M_{32} = -3\,,\ M_{33} = -6.$$

Next to compute the cofactors, multiply the minors by $-1$ if the sum of the subscripts is odd, and $1$ if the sum of the subscripts is even. For example, consider $C_{11}$ and $C_{12}$ computed below.

$$C_{11} = (-1)^{1+1} M_{11} = (-1)^{1+1}(-1) = (1)(-1) = -1\,,$$

$$C_{12} = (-1)^{1+2} M_{12} = (-1)^{1+2}(-5) = (-1)(-5) = 5.$$

Continue this procedure to compute all cofactors, shown below.

$$C_{11} = -1\,,\; C_{12} = 5\,,\; C_{13} = 4\,,$$

$$C_{21} = -2\,,\; C_{22} = -4\,,\; C_{23} = 8\,,$$

$$C_{31} = 5\,,\; C_{32} = 3\,,\; C_{33} = -6.$$

To compute the determinant, expand along any of the three rows or three columns.

First row expansion

$$\begin{aligned}\det(C^*) &= a_{11}C_{11} + a_{12}C_{12} + a_{13}C_{13} \\ &= (0)(-1) + (2)(5) + (1)(4) = 14\end{aligned}$$

Second row expansion

$$\det(C^*) = a_{21}C_{21} + a_{22}C_{22} + a_{23}C_{23} = 14$$

Third row expansion

$$\det(C^*) = a_{31}C_{31} + a_{32}C_{32} + a_{33}C_{33} = 14$$

First column expansion

$$\det(C^*) = a_{11}C_{11} + a_{21}C_{21} + a_{31}C_{31} = 14$$

Second column expansion

$$\det(C^*) = a_{12}C_{12} + a_{22}C_{22} + a_{32}C_{32} = 14$$

Third column expansion

$$\det(C^*) = a_{13}C_{123} + a_{23}C_{23} + a_{33}C_{33} = 14$$

As expected from Equation 2.18, the determinant is the same regardless of row/column expansion.

### 1.2.2 Inverse

*Mathematical Overview*: Given square base matrix $A$, the **inverse** of $A$ denoted $A^{-1}$ is defined explicitly and implicitly as

$$A^{-1} = \frac{\text{adj}(A)}{\det(A)} \ , \ AA^{-1} = I = A^{-1}A. \tag{2.20}$$

For $A^{-1}$ to exist, $A$ must be square and $\det(A) \neq 0$. If the second condition is met, $A$ is termed a **non-singular** matrix. If the second condition fail to be satisfied, $A$ is termed a **singular** matrix. If $A^{-1}$ exists, it is unique and $A$ is termed an **invertible matrix** [52, 144]. A non-singular matrix can be transformed into $I$ by a sequence of elementary operations. The same sequence of elementary operations can then transform $I$ into $A^{-1}$. Table 2.8 outlines common properties associated with the inverse of a matrix. Property i. 1 and i. 2 similar to matrix transposition state, the inverse of a product is equal to the product of the inverses in reverse order, and the inverse of a inverse matrix is the original matrix. Property i. 3 shows the interchangeability of matrix inversion and transposition. Property i. 4 displays the simplified computation associated with diagonal matrices.

**Table 2.8:** Inverse Properties

| | | |
|---|---|---|
| i. 1 | $(AB)^{-1} = B^{-1}A^{-1}$ | Inverse Multiplication |
| i. 2 | $(A^{-1})^{-1} = A$ | Inverse Property |
| i. 3 | $(A^T)^{-1} = (A^{-1})^T$ | Transpose Property |
| i. 4 | $A$ diagonal $\implies$ $A^{-1}$ is inverse of diagonal entries | Diagonal Property |

*Numerical Overview*: To compute $A^{-1}$, a partitioned matrix $A|I$ is developed.

$$\left[\begin{array}{cccc|cccc} a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & 0 & 0 & \cdots & 1 \end{array}\right] , |A|I| = m \times 2n \tag{2.21}$$

Elementary matrix operations are then performed to transform $A$ into $I$. Elementary operations (1) interchange two rows/columns, (2) multiply a row/column by a nonzero constant, and (3) add a multiple of a row/column to another row/column apply to both matrices [121]. After performing these operations, the resulting right-hand side of the partitioned matrix becomes $A^{-1}$. This method of analysis shown below is commonly referred to as **Gauss-Jordan elimination** [79].

$$\left[\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & 1 & \cdots & 0 & a'_{21} & a'_{22} & \cdots & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & a'_{m1} & a'_{m2} & \cdots & a'_{mn} \end{array}\right], |I|A^{-1}| = m \times 2n \tag{2.22}$$

Gauss-Jordan elimination is a very efficient numerical algorithm to invert a matrix. Gauss-Jordan elimination is also capable of solving the system $A\boldsymbol{x} = \boldsymbol{b}$, however Gauss Elimination is preferred. There exist certain conditions under which $A^{-1}$ doesn't exist, which relate to the value of $\det(A)$. That is $A^{-1}$ does not exist if (1) $A$ possesses two identical rows/columns, or (2) $A$ possesses rows/columns that are multiples of eachother or (3) $A$ possesses a row/column of zero entries [76].

$$\text{Identical Columns} \begin{bmatrix} 1 & \mathbf{2} & \mathbf{2} \\ 2 & \mathbf{3} & \mathbf{3} \\ 7 & \mathbf{8} & \mathbf{8} \end{bmatrix}, \text{Identical Rows} \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{2} \\ \mathbf{1} & \mathbf{2} & \mathbf{2} \\ 3 & 2 & 1 \end{bmatrix}$$

$$\text{Multiples Columns} \begin{bmatrix} \mathbf{4} & \mathbf{2} & 0 \\ \mathbf{8} & \mathbf{4} & 3 \\ \mathbf{16} & \mathbf{8} & 8 \end{bmatrix}, \text{Multiples Rows} \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{2} \\ 1 & 3 & 6 \\ \mathbf{2} & \mathbf{4} & \mathbf{4} \end{bmatrix}$$

$$\text{Zero Columns} \begin{bmatrix} 1 & \mathbf{0} & 0 \\ 2 & \mathbf{0} & 3 \\ 1 & \mathbf{0} & 8 \end{bmatrix}, \text{Zero Rows} \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 1 & 5 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

Consider example base matrix $E^*$ in Reference Equation A.4.

$$E^* = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 0 & -1 \\ -6 & 2 & 3 \end{bmatrix}$$

To compute $E^{*-1}$, first setup a partitioned matrix as shown in Equation 2.21.

$$\left[\begin{array}{ccc|ccc} 1 & -1 & 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 \\ -6 & 2 & 3 & 0 & 0 & 1 \end{array}\right]$$

Next, perform elementary matrix operations to transform $E^*$ into $I$.

EMO1: $R_2 + (-1)R_1 \rightarrow R_2$, EMO2: $R_3 + (6)R_1 \rightarrow R_3$

$$\left[\begin{array}{ccc|ccc} 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 & 0 \\ -6 & 2 & 3 & 0 & 0 & 1 \end{array}\right], \left[\begin{array}{ccc|ccc} 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 & 0 \\ 0 & -4 & 3 & 6 & 0 & 1 \end{array}\right]$$

EMO3: $R_3 + (4)R_2 \rightarrow R_3$, EMO4: $(-1)R_3 \rightarrow R_3$

$$\left[\begin{array}{ccc|ccc} 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 & 0 \\ 0 & 0 & -1 & 2 & 4 & 1 \end{array}\right], \left[\begin{array}{ccc|ccc} 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 & 0 \\ 0 & 0 & 1 & -2 & -4 & -1 \end{array}\right]$$

EMO5: $R_2 + R_3 \rightarrow R_2$, EMO6: $R_1 + R_2 \rightarrow R_1$

$$\left[\begin{array}{ccc|ccc} 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -3 & -3 & -1 \\ 0 & 0 & 1 & -2 & -4 & -1 \end{array}\right], \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & -2 & -3 & -1 \\ 0 & 1 & 0 & -3 & -3 & -1 \\ 0 & 0 & 1 & -2 & -4 & -1 \end{array}\right]$$

Thus, $E^{*-1}$ is given as the right partitioned matrix as shown in Equation 2.22.

### 1.2.3 LU Decomposition

*Mathematical Overview*: **LU decomposition** is a matrix factorization technique, in which base matrix $A$ is expressed as the product of a lower triangular matrix $L$ and upper triangular matrix $U$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{m1} & l_{m2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} 1 & u_{12} & \cdots & u_{1n} \\ 0 & 1 & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{2.23}$$

$U$ is found first by applying $k$ elementary matrices until $A$ becomes an upper-triangular matrix

$$U = E_k E_{k-1} E_{k-2} \cdots E_3 E_2 E_1 A. \tag{2.24}$$

Then $L$ becomes the inverse of the multiplication of the elementary matrices applied to $A$ for $U$

$$L = \Big( E_k E_{k-1} E_{k-2} \cdots E_3 E_2 E_1 \Big)^{-1} = \; E_1 E_2 E_3 \cdots E_{k-2} E_{k-1} E_k \tag{2.25}$$

The pair of equations are then constructed, solving first for $\boldsymbol{y}$ using back-substitution in Equation 2.26 and then second for $\boldsymbol{x}$ using forward-substitution in Equation 2.27.

$$L\boldsymbol{y} = \boldsymbol{b} \tag{2.26}$$

$$U\boldsymbol{x} = \boldsymbol{y} \tag{2.27}$$

*Numerical Overview*: The algorithm to solve Equation 2.26 for $\boldsymbol{x}$ via forward substitution is

$$y_1 = \frac{b_1}{l_{11}}$$

$$y_i = \frac{1}{l_{ii}} \Big( b_i - \sum_{j=1}^{i-1} l_{ij} y_j \Big) \, , \; 2 \leq i \leq n.$$

The algorithm to solve Equation 2.27 then for $\boldsymbol{x}$ via backward substitution is

$$x_n = \frac{y_n}{u_{nn}}$$

$$x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j\right),\ 1 \leq i \leq n-1.$$

The FLOP count to solve for the solution vector $\boldsymbol{x}$ is given as $\frac{2}{3}n^3 + n^2$, where $\frac{2}{3}n^3$ comes from the factorization of $A$ and $n^2$ the forward and backward substitutions. This method becomes very useful if new right-hand side vectors $\boldsymbol{b}$ are needed because the order of magnitude becomes only $O(n^2)$ as the factorization of doesn't have to be recomputed [51]. While LU decomposition is the cheapest algorithm computationally, instability can appear if zero elements appear in unfavorable matrix positions (the algorithm will attempt to divide by zero). This can be easily fixed by implementing pivoting. Matrix pivoting occurs either in partial (either row or column interchange) or full (both row and column interchange) pivoting with order of magnitude $O(m^3)$ [51].

Upper and lower triangular matrices have special characteristics that make them favorable in numerical analysis. A **upper triangular matrix** by definition is a square matrix that can have nonzero entries only on or above the main diagonal. A **lower triangular** matrix by definition is a square matrix that can have nonzero entries only on or below the main diagonal. Figure 2.12 displays the simplified solutions to the matrix presented below.

$$\begin{bmatrix} 1 & -2 & -2 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix}$$

| Inverse | Determinant | Eigenvalue/ Eigenvector |
|---|---|---|
| Inverse of diagonal entries | Product of diagonal entries | Eigenvalues are diagonal entries |
| $\begin{bmatrix} 1 & 2/3 & 2 \\ 0 & 1/3 & 1/2 \\ 0 & 0 & 1/2 \end{bmatrix}$ | (1)(3)(2) = 6 | $\lambda_1 = 1$<br>$\lambda_2 = 3$<br>$\lambda_3 = 2$ |

**Figure 2.12:** Upper and Lower Triangular Matrices

Consider example base matrix $G^*$ in Reference Equation A.6

$$G^* = \begin{bmatrix} 1 & -2 & -1 \\ 2 & 8 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Apply elementary matrix operations through Gaussian Elimination to transform $G^*$ into $U$.

EMO1: $-2R_1 + R_2 \rightarrow R_2$ , $R_1 + R_3 \rightarrow R_3$

$$E_1 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \implies E_1 G^* = \begin{bmatrix} 1 & -2 & -1 \\ 0 & 12 & 3 \\ 0 & -2 & 0 \end{bmatrix}$$

EMO2: $\frac{1}{6}R_2 + R_3 \rightarrow R_3$

$$E_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/6 & 1 \end{bmatrix} \implies E_1(E_1 G^*) = \begin{bmatrix} 1 & -2 & -1 \\ 0 & 12 & 3 \\ 0 & 0 & 1/2 \end{bmatrix} = U$$

To compute $L$, apply the EMOs in reverse order on $A$ as shown in Equation 2.25.

$$L = (E_2 E_1)^{-1} = E_1^{-1} E_2^{-1}$$

These inverse matrices are easy to compute. Simply switch the signs of all elements not on the main diagonal. The multiplication of the inverse matrices yields $L$.

$$E_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \; E_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/6 & 1 \end{bmatrix} \implies L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1/6 & 1 \end{bmatrix}$$

### 1.2.4 System of Linear Equations

*Mathematical Overview* : An equation is termed **linear** it it's in the form [121]

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b. \tag{2.28}$$

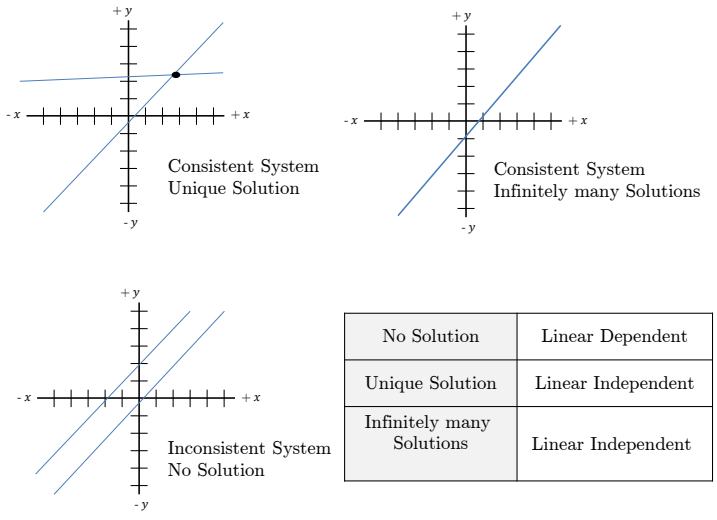
A **system of** $m$ **linear equations in** $n$ **unknowns** is then

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \quad + \quad \vdots \quad + \quad \vdots \quad + \quad \vdots \quad &= \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned} \tag{2.29}$$

A linear equations is said to be **linear independent** if

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = \mathbf{0} \tag{2.30}$$

where $a_1 = a_2 = \cdots = a_n = 0$. If there exists at least one $a_i \neq 0$, the linear equation is said to be then **linear dependent**. If a system of equations is linearly dependent, it is termed **singular**. Linear dependence results in degenerate matrices. A **row degeneracy** can occur if one or more of the $m$ linear equations is a linear combination of the others, it which case a unique solution will not exist. Likewise, a **column degeneracy** can occur if all equations contain certain variables only in exactly the same linear combination. Figure 2.13 gives a geometric approach to the system types.




| | |
|---|---|
| No Solution | Linear Dependent |
| Unique Solution | Linear Independent |
| Infinitely many Solutions | Linear Independent |

**Figure 2.13:** System of Linear Equations

A system of linear equations is written compactly in vector notation as $A\boldsymbol{x} = \boldsymbol{b}$. Matrix $A$ is termed the **coefficient matrix**, vector $\boldsymbol{x}$ the **solution vector**, and vector $\boldsymbol{b}$ the **right-hand side vector**, respectively. The goal in solving a system of linear equations is finding solution vector, $\boldsymbol{x}$. To find $\boldsymbol{x}$, elementary matrix operations are applied on the system of equations until matrix $A$ becomes upper triangular. These elementary matrix operations exist it three categories. First, **interchange** is where two rows/columns switch their positions. Second, **scaling** is where a row/column is multiplied by a nonzero constant. Lastly, **replacement** is where a multiple of a row/column is added to another row/column in the system. These operations allow for manipulation of $A$, such that back substitution becomes quickly efficient to compute $\boldsymbol{x}$. Most importantly, the solution vector $\boldsymbol{x}$ is not affected by applying these operations to the system. This method of solving a system of linear equations is termed **Gaussian Elimination**, and is shown below with the original system $A\boldsymbol{x} = \boldsymbol{b}$ in Equation 2.31 and the reduced system in Equation 2.32.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}. \tag{2.31}$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_m \end{bmatrix}. \tag{2.32}$$

*Numerical Overview* : The algorithm to perform Gaussian elimination and compute $\boldsymbol{x}$ is given

$$x_n = \frac{b'_m}{a'_{mn}} \tag{2.33}$$

$$x_i = \frac{1}{a_{ij}^{'}}\left(b_j^{'} - \sum_{j=i+1}^{n} a_{ij}^{'} x_j\right),\ i = n-1, n-2, \cdots 2, 1. \tag{2.34}$$

This algorithm has order of magnitude of $O(\frac{2}{3}n^3)$, which is exactly the same as LU decomposition. This order of magnitude comes from the $n^3/3 + n^2 - n/3$ multiplication and division FLOPS, and $n^3/3 + n^2/2 - 5n/6$ addition and subtraction FLOPS. LU decomposition is a type of Gaussian elimination, and $L$ and $U$ can be found while performing Gaussian elimination. However unlike LU Decomposition, Gaussian elimination is not useful if new right-hand side vectors $\boldsymbol{b}$ are produced. This is because unlike LU Decomposition which allows a new $\boldsymbol{b}$ vector to be substituted directly to compute a new $\boldsymbol{x}$ vector, a new $A^{'}$ matrix must be computed for every new $\boldsymbol{b}$ vector. In engineering and mathematical analysis which request a single output ($\boldsymbol{b}$) for variable input conditions ($A^{'}$) to find a stabilizer ($\boldsymbol{x}$) for the system, it is favorable to perform LU Decomposition over Gaussian Elimination to avoid the repetitive diagonalization of $A^{'}$ [1]. It should be noted that there exists one condition under which no solution exists for a given system of linear equations. This occurs if while applying elementary matrix operations to the system, a given linear equation appear in the form $0 = b_k$, where $b_k$ is a nonzero value. Equation 2.35 displays the system with conditions.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}^{'}x_2 + \cdots + a_{2n}^{'}x_n &= b_2^{'} \\ &\vdots \\ a_{kk}^{'}x_k + \cdots + a_{kn}^{'}x_n &= b_k^{'} \\ 0 &= b_{k+1}^{'} \\ &\vdots \\ 0 &= b_m^{'} \end{aligned} \tag{2.35}$$

No solution will exist if $k < m$, and at least one $b_{k+1}^{'}, \cdots, b_m^{'} \neq 0$. Precisely one unique solution will exist if $k = n$ and $b_{k+1}^{'}, \cdots, b_m^{'} = 0$. In which case, solve first the $n^th$ equation for $x_n$, and back substitute to solve for $x_{n-1}$, then repeat to solve for $x_{n-2}, \cdots, x_2, x_1$. Infinitely many solutions exist if $k < n$ and $b_{k+1}^{'}, \cdots, b_m^{'} = 0$. In which case, solve the $k^th$ equation for $x_k$ and back substitute to solve for $x_{k-1}$, then repeat to solve for $x_{k-2}, \cdots, x_2, x_1$.

Example system of three linear equations in three unknowns from Reference Equation A.5.

$$\begin{aligned} x_1 - 2x_2 + 3x_3 &= 9 \\ -x_1 + 3x_2 &= -4 \\ 2x_1 - 5x_2 + 5x_3 &= 17 \end{aligned}$$

Writing this system compactly in vector notation as shown in Equation 2.31

$$\begin{bmatrix} 1 & -2 & 3 \\ -1 & 3 & 0 \\ 2 & -5 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ -4 \\ 17 \end{bmatrix}.$$

Apply elementary matrix operations to the augmented matrix to solve using Gaussian Elimination

EMO1: $R_2 + R_1 \to R_2$, EMO2: $R_3 + (-2)R_1 \to R_3$

$$\left[\begin{array}{ccc|c} 1 & -2 & 3 & 9 \\ -1 & 3 & 0 & -4 \\ 2 & -5 & 5 & 17 \end{array}\right], \left[\begin{array}{ccc|c} 1 & -2 & 3 & 9 \\ 0 & 1 & 3 & 5 \\ 2 & -5 & 5 & 17 \end{array}\right]$$

EMO3: $R_3 + R_2 \to R_3$, EMO4: $\frac{1}{2}R_3 \to R_3$

$$\left[\begin{array}{ccc|c} 1 & -2 & 3 & 9 \\ 0 & 1 & 3 & 5 \\ 0 & 0 & 2 & 4 \end{array}\right], \left[\begin{array}{ccc|c} 1 & -2 & 3 & 9 \\ 0 & 1 & 3 & 5 \\ 0 & 0 & 1 & 2 \end{array}\right]$$

The reduced matrix after EMO4 presents itself in a stair-step pattern known as **row echelon form**. This is the form presented in Equation 2.32. Using Equations 2.33 and 2.34 back substitution can now be performed to solve for the solution vector.

$$x_3 = 2$$

$$x_2 + 3x_3 = 5 \implies x_2 = -1$$

$$x_1 - 2x_2 + 3x_3 = 9 \implies x_1 = 1$$

## 1.3 Matrix Types

Matrix addition and multiplication are two operations which have many computations performed in engineering and mathematical analyses. General algorithms to perform these computations were given in the previous sections. While these general algorithms work for all matrices, they are not optimized for certain matrix types. Special matrix types appear during analysis, and hybrid algorithms have been developed to optimally perform these computations. These special matrices fall under three categories, banded, dense and structured [112]. Dense matrices relate to the number of nonzero elements in a matrix. Banded and structured matrices have a noticeable behavior or pattern with the elements in a matrix. The following sections explains more about the common matrix types and their properties.

### 1.3.1 Banded

*Mathematical Overview* : Given matrix $E$, it is said $E$ is a **banded matrix** if

$$e_{ij} = \begin{cases} 0 & \text{if } j < i - k_1 \text{ or } j > i + k_2 \\ e_{ij} & \text{elsewhere} \end{cases} \tag{2.36}$$

where $k_1$ is the **left half-bandwidth** and $k_2$ the **right half-bandwidth**. The **bandwidth** of a matrix is the number of diagonals with nonzero entries. Special cases for $k_1$ and $k_2$ and the resulting $E$ matrix are

$$k_1 = k_2 = 0 \implies E \text{ is a diagonal matrix} \tag{2.37}$$

$$k_1 = k_2 = 1 \implies E \text{ is a tridiagonal matrix}$$

$$k_1 = 1\,,\ k_2 = n - 1 \implies E \text{ is a upper triangular matrix}$$

$$k_1 = n - 1\,,\ k_2 = 1 \implies E \text{ is a lower triangular matrix}$$

Examples of the four matrix types are presented below.

$$\text{Diagonal: } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & -9 \end{bmatrix}, \text{ Lower Triangular: } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 4 & 5 & 0 & 0 \\ 3 & 2 & 4 & 0 \\ 1 & 2 & 7 & 9 \end{bmatrix}$$

$$\text{Tridiagonal: } \begin{bmatrix} 1 & 8 & 0 & 0 \\ 4 & 5 & 2 & 0 \\ 0 & 1 & -1 & 6 \\ 0 & 0 & 3 & 2 \end{bmatrix}, \text{ Upper Triangular: } \begin{bmatrix} 1 & 3 & 2 & 8 \\ 0 & 5 & 5 & 6 \\ 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

*Numerical Overview*: Banded matrices have advantages in algorithm simplification, computational storage and output speed. An example of algorithm simplification occurs with the determinant computation. Instead of programming minors and cofactors, to compute the determinant of the matrix one just multiplies the diagonal elements. An example of computational storage occurs with the addition computation. Instead of storing all entries of a matrix, to compute matrix addition one just stores the diagonal elements of the matrices being added, and their resulting values. An example of output speed occurs with matrix inversion. Instead of computing the adjoint matrices and the determinant to compute the inverse matrix, one just applies the reciprocal to diagonal elements.

Banded matrices have applications in many fields of mathematics and engineering. Elliptic Partial Differential Equations (PDEs) that solve Poisson, heat and wave equations utilize Liebmann's Method and ADI Method. Liebmann's Method and ADI method introduce mesh points that populate banded and tridiagonal matrices, respectively [76]. Boundary-Value Problems (BVPs) for Ordinary Differential Equations (ODEs) utilize the Shooting Method and Linear Finite Difference (LFD) methods that solve BVPs with first and second order ODEs. These methods introduce central-difference formulas that populate tridiagonal matrices [49].

The following is a example of the common equation $A\boldsymbol{x} = \boldsymbol{b}$ with a tridiagonal $A$ matrix. The main diagonal is represented by elements, $d_i, 1 \leq i \leq n$. The diagonal above the main is termed the **superdiagonal** and is represented by elements $c_i, 1 \leq i \leq n-1$ [30]. The diagonal below the main is termed the **subdiagonal** and is represented by elements $a_i, 1 \leq i \leq n-1$ [30]. The regular double-subscript notation is not used in tridiagonal systems, rather a single subscript is used to denote diagonal elements. For example $d_{11}$ is simply $d_1$.

$$\begin{bmatrix} d_1 & c_1 & & & & & & \\ a_1 & d_2 & c_2 & & & & & \\ & a_2 & d_3 & c_3 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & a_{i-1} & d_i & c_i & & \\ & & & & \ddots & \ddots & \ddots & \\ & & & & & a_{n-2} & d_{n-1} & c_{n-1} \\ & & & & & & a_{n-1} & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Application of Gaussian Elimination produces the following algorithms for the two stages: forward elimination and backward substitution. For the first stage, forward elimination

$$d_i \leftarrow d_i - \left(\frac{a_{i-1}}{d_{i-1}}\right) c_{i-1}\,,\ 2 \leq i \leq n, \tag{2.38}$$

$$b_i \leftarrow b_i - \left(\frac{a_{i-1}}{b_{i-1}}\right) b_{i-1}\,,\ 2 \leq i \leq n. \tag{2.39}$$

For the second stage, backward elimination

$$x_n \leftarrow \frac{b_n}{d_n} \tag{2.40}$$

$$x_i \leftarrow b_i - \frac{1}{d_i}(b_i - c_i x_{i+1})\,,\ i = n-1, n-2. \cdots, 2, 1. \tag{2.41}$$

### 1.3.2 Dense

*Mathematical Overview* : Matrix $F$ is a **dense matrix** if the majority of elements in $F$ are nonzero. Diagonal and tridiagonal matrices are not considered dense, as the majority of the elements in those matrices are nonzero. Symmetric and triangular matrices on the other hand are considered dense matrices.

*Numerical Overview* : Algorithms exists that partition $F$ into smaller matrices to optimally fill GPU kernels and threads for a dense matrix [6]. These algorithms are used in matrix decomposition and eigenvalue/eigenvector computations [54]. Matrix Algebra on GPU and Multicore Architectures (MAGMA) is a project for designing optimal algorithms to perform linear algebra computations on NVIDIA GPUs. MAGMA has capabilities in performing matrix-matrix multiplication and matrix-vector multiplication as well as symmetric and triangular matrix system solvers [3]. CULA Dense is a NVIDIA library that is designed to optimally perform matrix computations with dense matrices. CULA Dense has capabilities in LU and Cholesky Decomposition, matrix inversion, multiplication, transposition and eigenvalue routines [37]. Figure 2.14 displays the two types of storage techniques for matrices. CULA Dense uses column-major ordering where elements of a column are stored contiguous in memory. C, on the other hand is row-major storage by default [37].

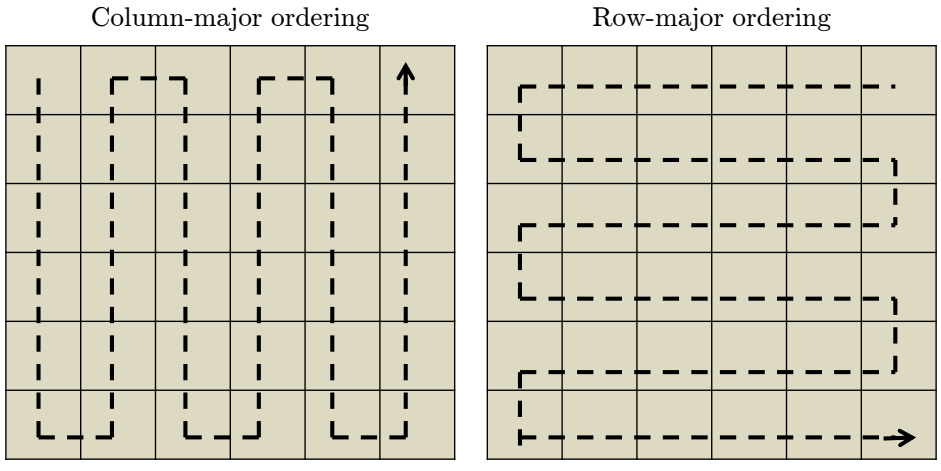



**Figure 2.14:** Column-major vs. Row-major memory ordering

### 1.3.3 Diagonal

*Mathematical Overview* : Matrix $G$ is a **diagonal matrix** if

$$G = \begin{bmatrix} g_{11} & 0 & \cdots & 0 \\ 0 & g_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{mn} \end{bmatrix}, \ |G| = m \times n \tag{2.42}$$

Diagonal matrices have unique multiplication properties [4]. Equation 2.43 displays that two diagonal matrices simplifies matrix multiplication to only the multiplication of each matrices respective diagonal elements. Equation 2.44 displays that repeated multiplication of $k$ diagonal matrices is the same as having each element on the diagonal raised to $k$.

$$\begin{bmatrix} g_{11}h_{11} & 0 & \cdots & 0 \\ 0 & g_{22}h_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{mn}h_{mn} \end{bmatrix} = \begin{bmatrix} g_{11} & 0 & \cdots & 0 \\ 0 & g_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{mn} \end{bmatrix} \begin{bmatrix} h_{11} & 0 & \cdots & 0 \\ 0 & h_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & h_{mn} \end{bmatrix} \tag{2.43}$$

$$G^k = \begin{bmatrix} g_{11} & 0 & \cdots & 0 \\ 0 & g_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{mn} \end{bmatrix}^k = \begin{bmatrix} g_{11}^k & 0 & \cdots & 0 \\ 0 & g_{22}^k & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{mn}^k \end{bmatrix} \tag{2.44}$$

*Numerical Overview* : Below are examples of the traditional diagonal, identity and scalar matrices.

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{bmatrix} \tag{2.45}$$

### 1.3.4 Sparse

*Mathematical Overview*: Matrix $H$ is a **sparse matrix** if the majority of elements in $H = 0$. Diagonal and tridiagonal are considered sparse matrices. Symmetric and triangular matrices on the other hand are considered dense. Figure 2.15 displays a comparison between the two types.

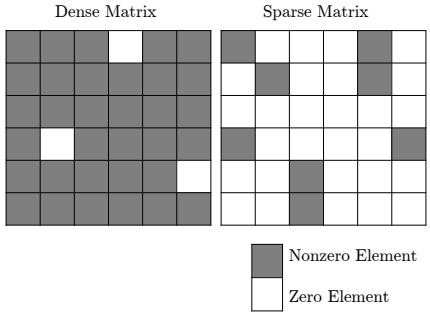
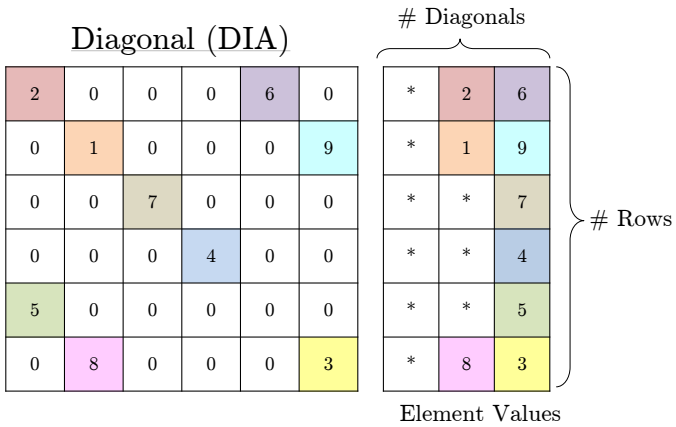



**Figure 2.15:** Dense and Sparse Matrices

There exists special types of storage formats for sparse matrices. Diagonal (DIA) Format is optimal when nonzero elements are restricted to a small number of matrix diagonals. The benefits of this method is memory requirement reductions and reduced data transfer. The downsides are when the sparse matrix doesn't have represent a diagonal pattern, and as a result many zero elements are stored in memory. Figure 2.16 displays the original matrix to the left, with nonzero elements color-coded for visualization, and the storage vector. The storage vector places element values in a matrix of size of $6 \times 3$ since there are 6 rows and 3 diagonals. The *'s symbolize matrix padding.




| | | | | | |
|---|---|---|---|---|---|
| 2 | 0 | 0 | 0 | 6 | 0 |
| 0 | 1 | 0 | 0 | 0 | 9 |
| 0 | 0 | 7 | 0 | 0 | 0 |
| 0 | 0 | 0 | 4 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 |
| 0 | 8 | 0 | 0 | 0 | 3 |

| | | |
|---|---|---|
| * | 2 | 6 |
| * | 1 | 9 |
| * | * | 7 |
| * | * | 4 |
| * | * | 5 |
| * | 8 | 3 |

**Figure 2.16:** Diagonal (DIA) Format

ELLPACK (ELL) Format stores column indicies in a matrix of size $6 \times 2$ since there are 6 rows and a maximum of 2 entries in any row. In total there are 6 columns, with column indicies counted (1,2,...,6). Element 1 for example is in the second column, resulting in a value of 2 in the column indicies matrix. Likewise, element 3 is in the sixth column, resulting in a value of 6 in the column indicies matrix. The second matrix similar to DIA format stores element values. Unlike DIA format where the padding is placed to the left, in ELL Format the padding is placed to the right. ELL is implemented more often that DIA, since nonzero column elements do not have to follow any pattern. However, large unstructured meshes are not optimally supported using ELL (ELL is suited for regular grid patterns or semi-structured meshes). Figure 2.17 displays ELL Format.

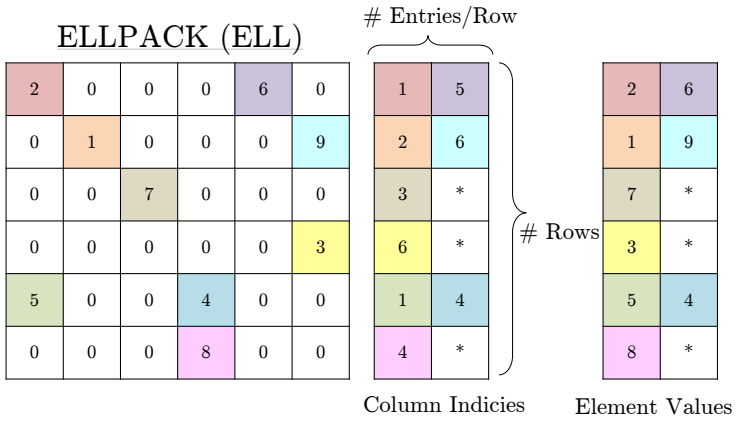



**Figure 2.17:** ELLPACK (ELL) Format

Coordinate (COO) Format is another method to store sparse matrices. COO Format uses three arrays for row indices, column indicies and element values. The required storage is proportional to the number of nonzero elements, and unlike DIA and ELL makes storage of the row and column information. Row-major ordering is used to determine the values for the row indicies matrix. Column-major ordering is used to determine the values for the column indicies matrix. Row-major ordering is used to determine the element values matrix. For example, to generate the row indicies matrix start in the first row sixth column. Moving right to left in the first row, count the times a number is counted (2 times; 6 and 2). Go to row two, move now from left to right and count again. Continue alternating directions until completing sixth row. Figure 2.18 displays COO Format.

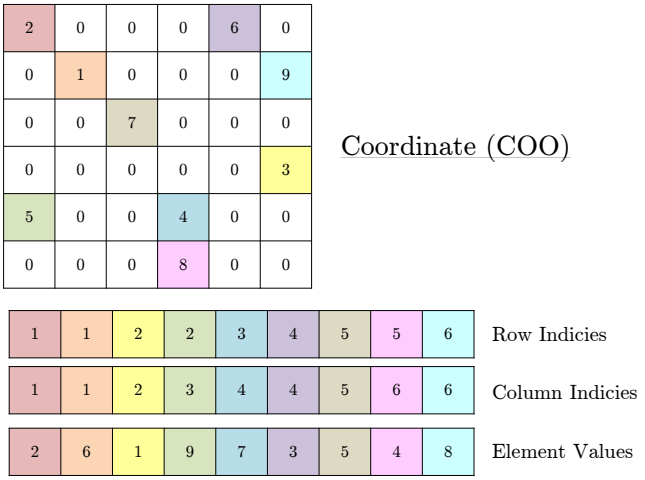



**Figure 2.18:** Coordinate (COO) Format

Compressed Sparse Row (CSR) Format stores three arrays: row offsets, column indicies and element values. Row offsets stores offset values and has dimension of $1\times$ (number of rows$+1$). Figure 2.19 displays CSR Format.

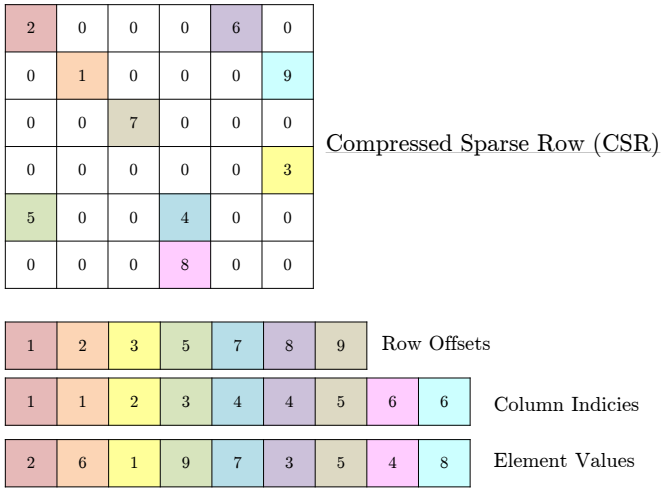



**Figure 2.19:** Compressed Sparse Row (CSR) Format

*Numerical Overview*: Sparse matrices appear commonly when solving PDEs, and in particular when performing CFD analyses. Since sparse matrices consist mostly of zero elements, algorithms are optimized to avoid wasted memory on them. The matrix storage types listed above are the most common for sparse matrices, as they utilize memory efficiently. Generally, DIA and ELL are used for structured matrices, whereas CSR and COO are used for unstructured matrices.

### 1.3.5 Symmetric

*Mathematical Overview*: A matrix $J$ is a **symmetric matrix** if it satisfies the following properties

$$J = J^T \tag{2.46}$$

$$J^{-1}J^T = I \tag{2.47}$$

*Numerical Overview*: Symmetric matrices occur frequent enough in mathematics that their analysis is warranted. In quantum mechanics, a system is described by a time-dependent vector whose evolution is given by Schroedingers equation. Schroedingers equation involves a symmetric matrix $L$ which represents the energy in a system. In chemistry, Huckel theory computes electron density distribution of molecules involving the adjacency matrix $A$ which is symmetric. In statistics, random vectors denoting the expected value of an event produce covariance matrices which are symmetric. [75]

### 1.3.6 Triangular

*Mathematical Overview*: The matrix $L$ and $U$ are defined respectively as the lower and upper triangular matrices.

$$l_{ij} = \begin{cases} l_{ij} & \text{if } j > i \text{ or } j = i \\ 0 & \text{if } j < i \end{cases} \tag{2.48}$$

$$u_{ij} = \begin{cases} u_{ij} & \text{if } j < i \\ 0 & \text{if } j > i \text{ or } j = i \end{cases} \tag{2.49}$$

*Numerical Overview*: Triangular matrices have some interesting applications in many fields. In cryptology, upper triangular matrices are used as a key exchange scheme, private session keys for encrypted communication channels and pseudorandom genertors to generate keys and steam ciphers [2]. In PDEs, lower triangular matrices are used in a reduced-basis discretization procedure for affine computational decompositions [8].

# 2. Graphic Processing Unit

## 2.1 Historical Overview

The first resemblance of the modern graphic cards is seen with the Radio Corporation of America CDP 1861 monochrome video chip in 1976. Television Interface Adapter's 1A which was integrated into the Atari 2600 and Motorola's MC6845 video address generator generated screen displays and sound effects, became the baseline for personal computers in the late 1970s. LSI's ANTIC and CTIA/GTIA (Color/Graphic Television Interface Adaptor) was implemented into the Atari 400 to generate playfield graphics (background) and colors (moveable objects) in 1981. Silicon Graphics popularized three-dimensional graphics in multiple markets (government, defense, scientific visualization) and the graphically driven operation systems (Microsoft Windows), helped create a market for a new graphical processor in the late 1980s [125]. These events alongside others in the history of graphic processors are shown in Figure 2.20.

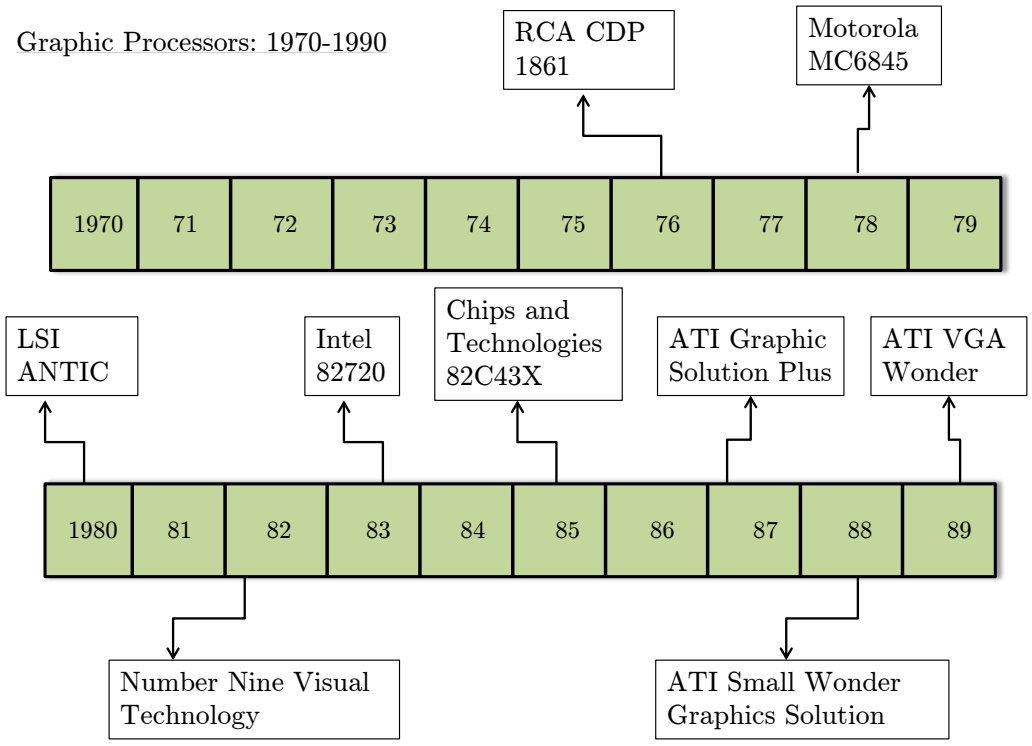



**Figure 2.20:** Graphic Processors: 1970-1990

Silicon Graphics opened programming interface to its hardware with the release of the OpenGL library, and users purchasing two-dimensional display accelerators rose for personal computing in the early 1990s. Realistic first-person graphically driven PC games (Doom and Quake) and companies like NVIDIA, ATI Technologies and 3dfx interface releasing affordable graphic accelerators, drove the demand for computer applications with three-dimensional graphics in the mid 1990s. GeForce 256 releases by NVIDIA in 1998 computed transform and lighting computations on the GPU, enhancing potential for visual applications and marked the beginning of increasingly using the graphics processor. In 2001, NVIDIAs release of the GeForce 3 series marked the first chipset to implement Microsoft's DirectX 8.0 Standard, which required compliant hardware that was programmable vertex and pixel shading stages. This gave developers control over GPU computations [125]. These events in the history of graphic processors are shown in Figure 2.21.

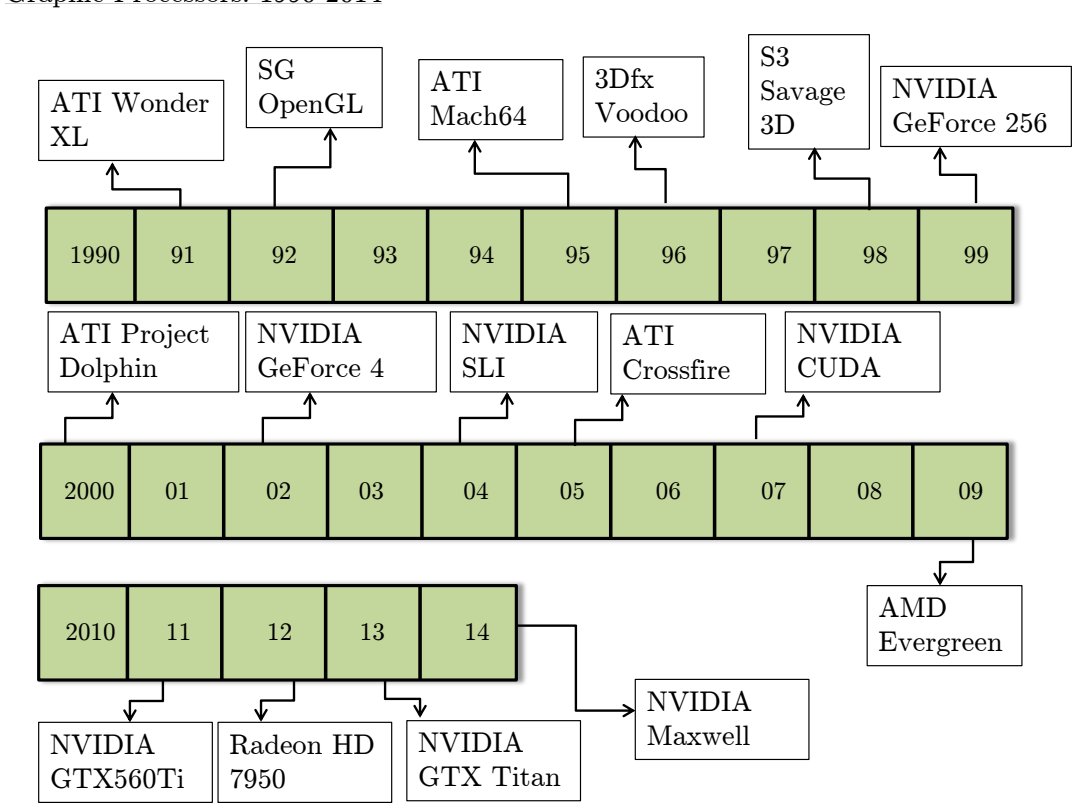



**Figure 2.21:** Graphic Processors: 1990-2014

Graphic processors up until 1997 was hardware designed exclusively to be extremely fast at processing large graphics data sets (polygons, pixels and voxels) for rendering tasks. These tasks involved processing, producing and updating images on a computer screen (or monitor). General purpose programming on graphic processing units (GPGPU) became of significant interest after the creation of OpenGL Shading Language by API in 1997. Early GPGPU can best be described in three words: convoluted, limited and promising.

**Convoluted:** Application Programming Interfaces (APIs), sets of functions, procedures, or classes that an operating system provides support to computer program requests were the only way to interact with a GPU. GPUs are designed to produce colors for every pixel on a computer screen. These pixel colors are generated by programmable arithmetic logic units (ALUs), digital circuits that perform arithmetic and logical operations. Thus to perform GPGPU, a programmer was required to use a pixel shaders $(x, y)$ position on the computer screen as well as additional graphical information (input colors, texture coordinates) to compute output colors. Input colors would represent actual numerical data signifying something other than just a color, and programmers would program pixel shaders to perform arbitrary computations, resulting in an output color.

**Limited:** This early programming model for GPGPU was too restrictive for any mass development. Limitations existed in the number of input colors and texture units a programmer could use for performing computations. Limitations in how and where programmers could write output results to memory were also a concern. Algorithms requiring the ability to write to arbitrary locations (scatter algorithms) were not capable of running on a GPU. Debugging code executed on a GPU was difficult; computing incorrect results, code failing to terminate and simply hung machines were some debugging issues. The most striking limitation of early GPGPU came in the form of memory precision. Most GPUs did not comply with IEEE standards for single precision (SP) and double precision (DP), making it difficult to instrument in research applications.

**Promising:** While the issues in early GPGPU were restrictive and well documented, the acceleration results obtained were very promising. For applications that were computationally

expensive and speed was a main computing concern, the programmable graphics card was an inexpensive parallel computing alternative to supercomputers. Speed up factors of in the ranges of 10-50x were common for a variety of scientific applications. Since the graphics card primary role was to produce computer images, it was already part of the computer architecture and didn't have to be added as separate hardware. With the demand for high-end consumer graphics for gaming purposes, companies that manufactured these graphics cards (NVIDIA, AMD) constantly were forced to develop faster cards, with increased memory and added features. This gave researchers faster clock speeds, more programmable memory and cooling technologies to perform scientific computations. Most importantly, as central processing units (CPUs) experienced halts in technological advancements, their manufacturers (Intel, AMD) could no longer increase computational performance by increasing processor speeds. The physical limitation in cooling chips with increased number of transistors forced researchers to transition to the GPU as a computing device of the future rather than the CPU.

The most important advancement to GPGPU came in 2007 when NVIDIA released its general purpose programming model and developer in Compute Unified Device Architecture, known as CUDA. The CUDA architecture made once limited graphic producing hardware into a massively multithreaded platform capable of achieving GFLOPS and a cheap alternative for HPC computing. The fundamental strength of a GPU is its extremely parallel nature. The CUDA programming model allows developers to exploit this parallelism by writing standard ANSI C code that runs millions of parallel invocations (called threads) simultaneously. Researchers once having to use hardware-hacking techniques such as shader languages and graphic APIs, and having to deal with limitations in the form of available input/outputs and precision, were now capable of performing GPU computations easily in CUDA. The three levels of acceleration in GPU computing have been causally defined by David Kirk as just faster (2-3x), significantly faster (5-10x) and fundamentally different (100x). NVIDIA by opening their GPU architecture to developers allowed all three of these levels to be reached for an assortment of scientific and engineering applications.

## 2.2 Architecture Overview

Flynn's Taxonomy classifies computer architectures into four main categories. These categories are single instruction single data (SISD), single instruction multiple data (SIMD), multiple instruction single data (MISD) and multiple instruction multiple data (MIMD). A single core CPU performing one task at a time would be an example of a SISD architecture. A computer executing multiple data streams over a single core processor, such as a standard GPU is an example of a SIMD architecture. Multiple instructions executing on a single stream of data are the least common architecture of the four, seeing application mainly in space shuttle controls. A dual or quad-core desktop computer processing independent instructions simultaneously is an example of a MIMD architecture. NVIDIA CUDA employs a single instruction multiple thread (SIMT) architecture to GPU computing. Kernels relay instructions to threads of a block, which are executed in warps (32 threads). Each warp executes a single instruction at a time across its threads. The threads within the warp are free to follow any execution path (hardware controls execution divergence automatically), however this process is most efficient when threads follow the same execution path for the entirety of the execution [34]. Kernels, warps and blocks are all entities that are exclusive to the CUDA programming model setup for NVIDIA GPUs employing an SIMT architecture.

Figure 2.22 displays the CUDA (or Parallal Thread Execution (PTX)) programming model as it relates to the two primary computing hardware: the CPU (host) and GPU (device). **Kernels**, parallel portions of an application that are executed on the device are CUDA Single Program Multiple Data (SPMD) functions. Kernels are launched one at a time into **grids**. Grids are two-dimensional entities with a maximum size of $65535 \times 65535$. In Figure 2.22, Kernel 1 will launch into Grid 1 and finish, then Kernel 2 will launch into Grid 2 and finish, and so on until all kernels are launched. Each grid consists of a collection of **blocks** (or cooperative thread array (CTA)), which are three-dimensional entities with a maximum size of $512 \times 512 \times 64$. When a kernel is launched, each block within a grid can perform its job at the same time. CTAs executing the same

kernel can be combined into a grid to increase the number of threads launched in a single kernel invocation, however communication and synchronization between threads is not available between threads in different CTAs.

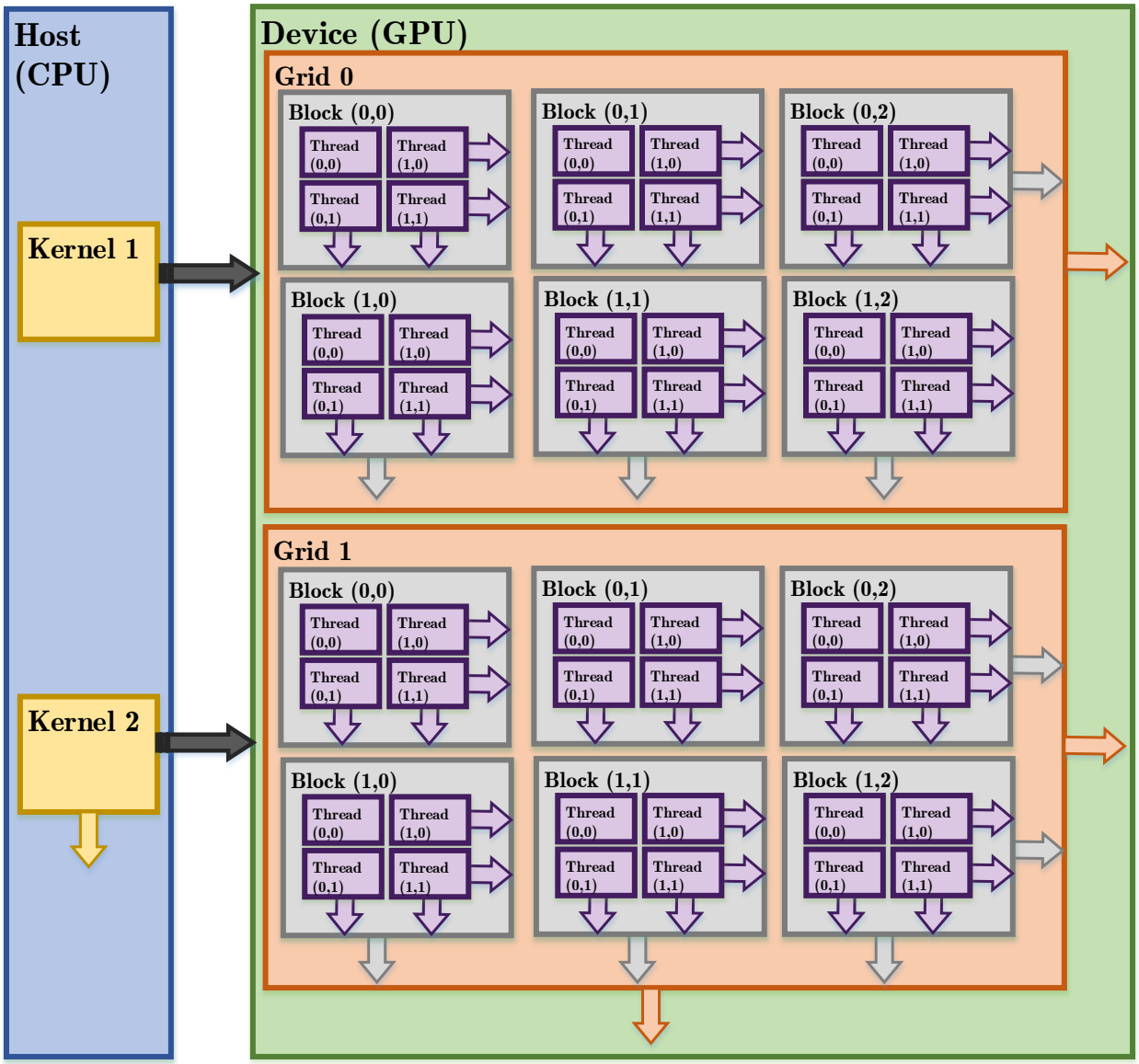



**Figure 2.22:** CUDA Programming Model: : Kernel, Grid and Block

Each block is composed of common **threads**, three-dimensional entities as shown in Figure 2.22. Thread blocks may contain up to 512 threads. Threads within the same block can communicate through shared memory and execution synchronization, whereas threads from different blocks

can't cooperate [71]. To communicate between threads inside of the CTA, synchronization points can be set up where threads wait until every thread in the CTA finishes its process [107]. Threads within the CTA are given a unique thread identifier, which is used to determine roles, compute memory addresses and assign input and output positions. The three-dimensional vector tid is the thread identifier, and ntid is the number of threads in each CTA. Threads execute in warps, which are subsets of 32 threads within a CTA that perform the same command simultaneously [107]. When each block is launched, the threads are filed in the column-major ordering pattern, filling top to bottom and then left to right. With the large amount of CUDA threads available as computing resources, NVIDIA GPUs are able to perform a large amount of computations in a short period of time, referred to as high **throughput**. The CPU being a computing device also possesses threads. CPU threads are generally heavyweight, as context switches (swapping two threads) are slow and computationally expensive. GPU threads are lightweight, with little creation overhead and instant switching, requiring thousands to be running simultaneously in warps to achieve high computing efficiency. Separate registers are allocated to active threads, and as a result no register swapping occurs when threads switch. CPU cores are designed to minimize latency, while a GPU is designed to maximize throughput [98].

A GPU is suited for performing computations that run on numerous data elements in parallel. Matrix arithmetic, where the same operation is being performed across many elements at the same time is a primary example of a optimal application for GPGPU. Adjacent threads should have coherence in memory access, a property known as **coalescing**. Amount of data transfers between CPU and GPU should be minimized, data should be kept on the GPU as long as possible, and multiple kernel calls on the data should be optimized. Information into matrix type (sparse, dense, etc.) can provide further acceleration by simply using predefined libraries and functions optimized for parallel computations made available by NVIDIA. For example, if given a sparse matrix one could easily apply any of the discussed matrix storage formats: DIA, ELL, COO or CSR to avoid storing zero elements that will waste computational time and memory resources.

## 2.3 Memory Overview

Figure 2.23 displays the process of how the four processor cores within the CPU fetch memory. The processor queries the first cache (L1). If the data is present in the L1 cache, the high-speed cache provides the data to the processor. L1 cache are advantageous because they runs at or near processor speed, however their size (16-32 KB) is quite small. If data is not in the L1 cache, the processor makes a fetch for the level two (L2) cache. The L2 cache is slower than the L1 cache, but holds a larger memory (256 KB). If data is not in the L2 cache, the processor makes a fetch for the level three (L3) cache. Similarly, the L3 cache is slower than the L2 cache, but holds a larger memory (multiple MB). Finally, if the data is not in the L3 cache, the processor makes a fetch for main memory (DRAM). While it would be advantageous to have a larger cache, the cache grows proportionally with the physical size of the processor. In other words, a larger processor chip would be more expensive to manufacture, and posses a higher likelihood of containing an error [34].

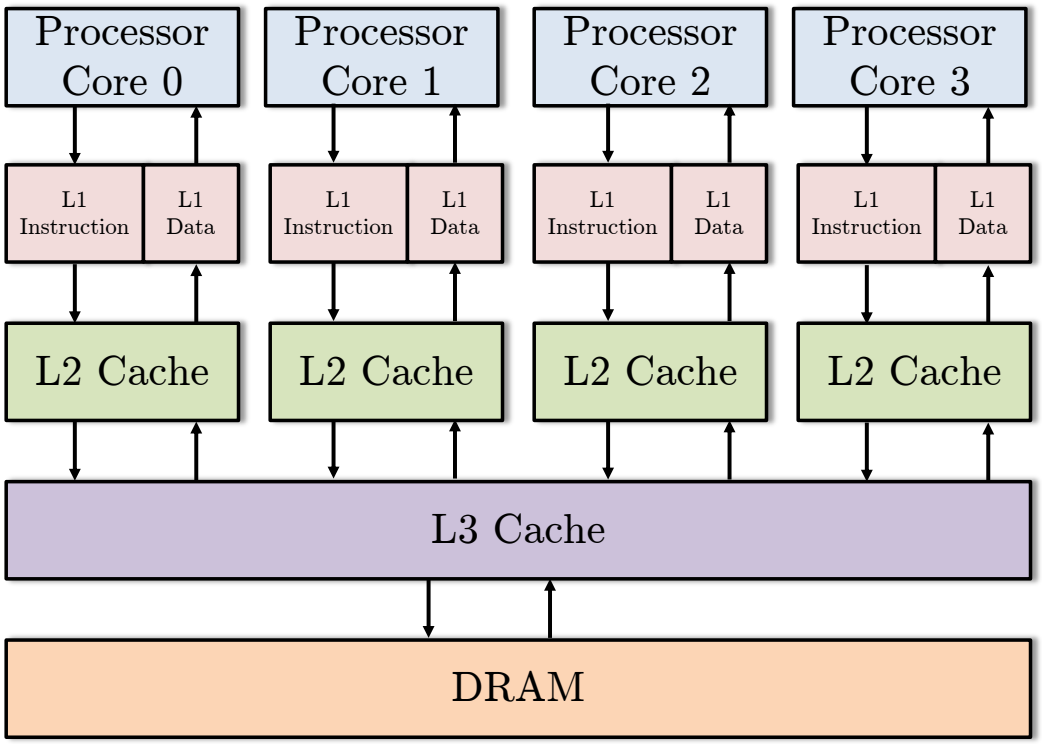



**Figure 2.23:** CPU Memory Breakdown

Figure 2.24 likewise displays the memory hierarchy of the GPU. The device possesses five types of memory: constant, global, local, shared and texture. A thread within a given block has access to its private local memory (purple arrows). A thread block (CTA) has access to shared memory which is usable by all threads within a block (black arrows). Shared memory on the GPU is similar to the L1 cache on the CPU; it's the fastest access memory but also the smallest available. Constant and texture memory are read-only (one-way red arrows). Global memory is read-write (two-way red arrows) and is functionally similar to the DRAM on the CPU. Data transfer between the host and device (blue arrows) occurs within the bottom three memory layers, are computationally expensive and should be limited to achieve optimal acceleration rates.

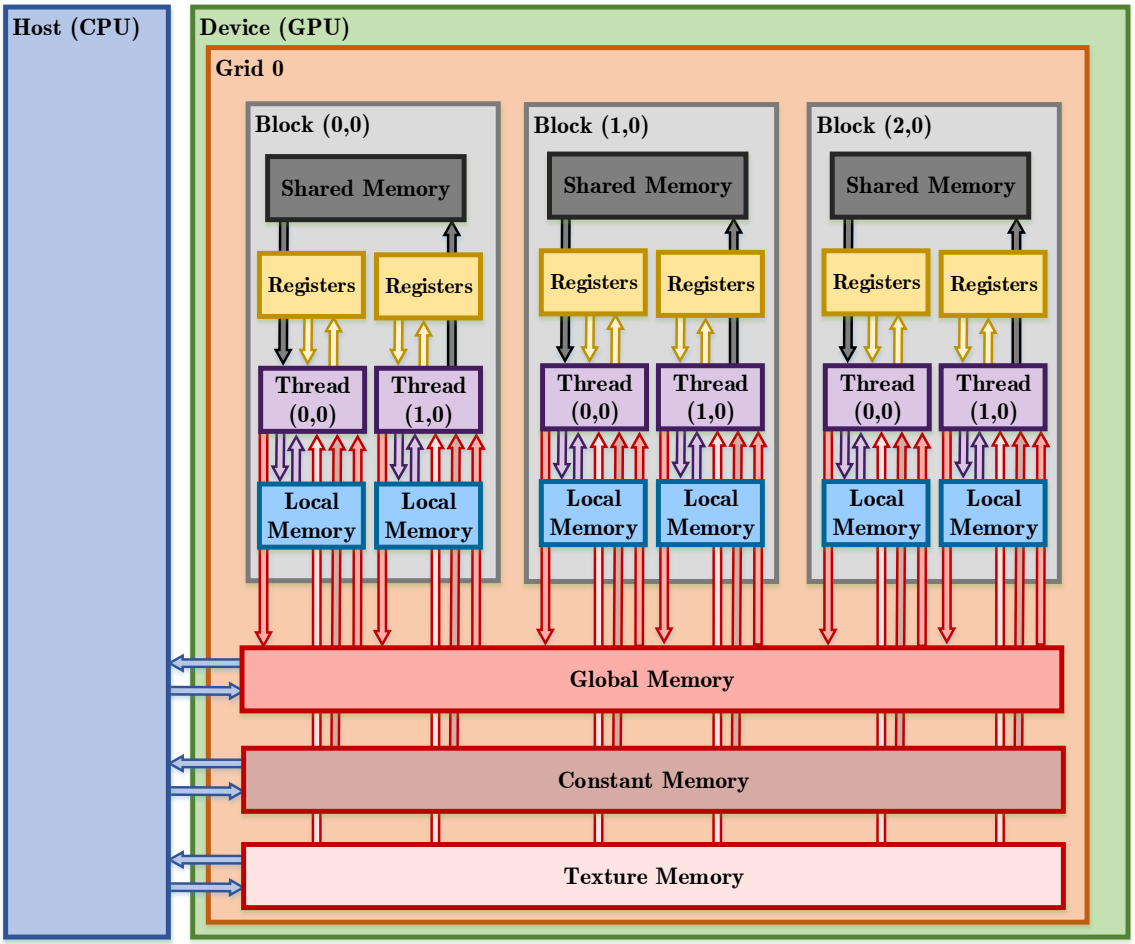



**Figure 2.24:** GPU Memory Hierarchy

## 2.4 Precision & Accuracy Overview

Performing algorithms with floating point values on a computer by their nature approximate exact answers. Thus, it is important that algorithms have the highest precision possible to minimize deviation between these computer approximated values and their exact counterparts. The Institute of Electrical and Electronics Engineers (IEEE) provide technical standards for algorithms performing floating point arithmetic. IEEE 754-2008 provided the current standard for all computing architectures, including NVIDIA's CUDA [100]. In IEEE 754 number are classified in three categories: finite numbers (floating point values), two infinities ($+\infty, -\infty$) and non-finite quantities (NaN). A finite number is described by three integers, $s$ the sign (0 or 1) for polarity, $c$ the significand (any finite number) and $q$ the exponent. Equation 2.50 is an example of how the finite number 192 is represented using this convention.

$$192 = (-1)^s \times b^q \times c = (-1)^0 \times 2^7 \times 1.5 \tag{2.50}$$

Finite integer values, such as 192 can be represented exactly in computer memory as they do not contain a fractional component. However, non-repeating and non-terminating fractions such as $\pi$ and $2/3$ can only be approximated in binary. Approximations of these numbers are found by applying the rules for rounding and rounding modes outlines in IEEE 754. This standard was revised from the 1985 initial release to include the fused multiple-add (FMA) operation, which improves precision in rounding. An example of the advantage of FMA is presented below.

Consider the computation $x^2 - 1$ to four decimal places of precision, where $x = 1.0004$. The exact mathematical value of this simple computation is $8.0016 \times 10^{-4}$. The prior method for rounding developed in IEEE 754-1985 would round twice producing a value of $8.002 \times 10^{-4}$. FMA rounds this computation only once producing a value of $8.000 \times 10^{-4}$. The result of rounding twice for a single multiply-add operation gives a percent error of $2 \times 10^{-2}\%$ vs. $5 \times 10^{-3}\%$ using FMA. Matrix computations such as LU decomposition, Gaussian Elimination, and inversion rely

heavily on multiply-add operations. Thus it was vital to ensure the NVIDIA GPU used in this work could implement FMA. NVIDIA classifies its CUDA GPUs based on compute capacity. Compute capacity value range from 1.0 to 3.5, where any values above 2.0 are IEEE-2008 compatible and implement FMA. The Quadro 2000M GPU has a compute capacity of 2.1 and GeForce GTX 760 has a compute capacity of 3.0 [100]. The GPUs used in this work are not only IEEE-754 compliant, but they possess the highest level of precision currently available for floating point operations [100].

## 2.5 CUDA C Overview

CUDA runtime API, also referred to as CUDA C is the runtime library, driver and language extensions from ANSI C that provide users with a friendly interface for GPU computing. CUDA runtime functions begin with *cuda* (cudaMalloc, cudaFree, cudaMemcpy). The following syntax is used to communicate with the computer for tasking computations to the GPU and CPU. CPU functions are denoted by ____host____ whereas GPU functions are denoted by ____device____. CPU functions can only be called and executed on the host, likewise GPU functions can only be called and executed on the device. Device functions can't have recursion, static variable declarations within functions, or variable number of arguments. Under the circumstance that these elements are present in the numerical algorithm, the user is forced to send these portions to CPU functions or restructure to remove these elements.

NVIDIA CUDAs Compiler Driver NVCC is responsible for separating the device functions from the host code, compiling the device functions using proprietary NVIDIA compilers/assemblers, compiling the host code using Visual Studio compiler, and afterwards embedding the compiled GPU functions as load images in the host object file [100]. In the linking stage, specific CUDA runtime libraries are added for supporting remote SIMD procedure calling and for providing explicit GPU manipulation such as allocation of GPU memory buffers and host-GPU data transfer. NVCC is similar to the GNU compiler GCC in that it accepts a range of conventional compiler op-

tions, and is advantageous as it hides the intricate details of CUDA compilation from developers.

# 3. Computational Fluid Dynamics

## 3.1 Historical Overview

Computational fluid dynamics (CFD) is an engineering analysis method with direct application to many thermal (heat transfer) and fluid (flow behavior and regime) models and simulations. These CFD models require matrix computations discussed in the previous sections. Maximizing accuracy and precision while minimizing computational cost and generation time of the CFD models are of primary interest, and these characteristics can be optimally implemented with GPU accelerated algorithms.

CFD traditionally uses two main methods to model problems, Navier Stokes and Lattice Boltzmann. Navier Stokes methods which relies on continuity and conservation principles (mass, energy, momentum), have been the foundational methods for CFD since the early 1960s [66]. Lattice Boltzmann methods are physics-based particle collision models based on gas particle interaction in local domains, which have gained popularity in the CFD world since its inception in the early 1980s. While the Navier Stokes approach is more founded and mature method [123], the Lattice Boltzmann methods offer robust solutions independent of numerical stability (high non-linear flow), simplicity of algorithm application which can be applied in a parallel manner (GPU programmable), and can solve for pressure terms implicitly [47].

A **fluid** is any matter that deforms continuously under the application of a shear stress. Fluids are assumed to be Newtonian; **Newtoninan fluids** have rate of deformations proportional to applied shear stresses. Fluid dynamics has two main approaches: Eulerian and Lagrangian. The **Eulerian** (or **control volume**) approach describes the fluid as a function of space and time. A **control volume** is a volume in space through which fluid may flow. The **Lagrangian** (or **system**) approach describes the fluid as a function of time. A **system** is any collection of matter through

which fluid may move, flow and interact with its surroundings. Lattice Boltzmann methods (LBM) take a Eulerian approach, whereas Navier Stokes equations (NSE) take a Lagrangian approach.

## 3.2 CFD Analysis Methods

### 3.2.1 Navier-Stokes

The Navier-Stokes equations are derived directly from mass and momentum conservation laws. These conservation laws are an immediate application of Reynolds Transport Theorem. **Reynolds Transport Theorem (RTT)** is derived as follows. Consider unidirectional fluid flow shown in Figure 2.25. Initially at $t = t_0$ the fluid is completely contained inside the control volume (red dashed lines), and at a later time $t = t_0 + \Delta t$ is partially contained within the control volume (blue dashed lines). In other words, at $t = t_0$ the fluid is within regions I and CV - I, and at $t = t_0 + \Delta t$ the fluid is within regions CV - I and II.

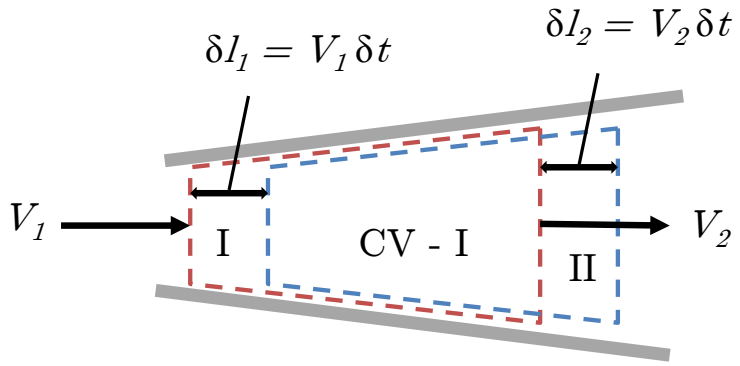



**Figure 2.25:** Reynolds Transport Theorem

The system for this analysis will consist of the area between the two boundaries (gray lines). Take $B_{sys}$ as any **extensive** (or **mass dependent**) **property** and $b_{sys}$ as any **intensive** (or **mass independent**) **property** within the system. The following relationship holds between the extensive and intensive property

$$B_{sys} = b_{sys}\, m = b_{sys}\, \rho\, V. \tag{2.51}$$

Written in summation and integral format the extensive property is

$$B_{sys} = \lim_{\delta V \to 0} \left( \sum_{i=1}^{\infty} b_{sys_i}\, \rho_i\, \delta V_i \right) = \int_{sys} b\, \rho\, \delta V\, dV. \tag{2.52}$$

The rate of change of the extensive property $B_{sys}$ is then the time derivative of Equation 2.52

$$\frac{dB_{sys}}{dt} = \frac{d}{dt}\left( \int_{sys} b\, \rho\, dV \right). \tag{2.53}$$

The rate of change of the extensive property $B_{CV}$ for the control volume is similarly

$$\frac{dB_{CV}}{dt} = \frac{d}{dt}\left( \int_{CV} b\, \rho\, dV \right). \tag{2.54}$$

Applying kinematic relation between distance and velocity, assuming constant fluid flow ($v_1 = v_2$)

$$\frac{\delta\, l_1}{\delta t} = v_1 = v_2 = \frac{\delta\, l_2}{\delta t}. \tag{2.55}$$

Since the control volume was chosen to be the system at $t = t_0$, it is known that $B_{sys}(t) = B_{CV}(t)$. The rate of change of $B_{sys}$ over an infinitesimal time interval is derived from

$$\begin{aligned}
\frac{\delta B_{sys}}{\delta t} &= \frac{B_{sys}(t+\delta t) - B_{sys}(t)}{\delta t} \\
&= \frac{B_{CV}(t+\delta t) - B_1(t+\delta t) + B_2(t+\delta t) - B_{CV}(t)}{\delta t} \\
&= \frac{B_{CV}(t+\delta t) - B_{CV}(t)}{\delta t} - \frac{B_1(t+\delta t)}{\delta t} + \frac{B_2(t+\delta t)}{\delta t} \\
&= \frac{B_{CV}(t+\delta t) - B_{CV}(t)}{\delta t} - \frac{b_1\, \rho_1\ A_1\, \delta l_1}{\delta t} + \frac{b_2\, \rho_2\ A_2\, \delta l_2}{\delta t} \\
&= \frac{B_{CV}(t+\delta t) - B_{CV}(t)}{\delta t} - \frac{b_1\, \rho_1\ A_1\, v_1\, \delta t}{\delta t} + \frac{b_2\, \rho_2\ A_2\, v_2\, \delta t}{\delta t}.
\end{aligned} \tag{2.56}$$

Taking the limit as $t \to 0$

$$\lim_{t \to 0}\left( \frac{\delta B_{sys}}{\delta t} \right) = \lim_{t \to 0}\left( \frac{B_{CV}(t+\delta t) - B_{CV}(t)}{\delta t} - \frac{b_1\, \rho_1\ A_1\, v_1\, \delta t}{\delta t} + \frac{b_2\, \rho_2\ A_2\, v_2\, \delta t}{\delta t} \right), \tag{2.57}$$

we have the following relationship

$$\frac{\partial B_{sys}}{\partial t} = \frac{\partial B_{CV}}{\partial t} - b_1\, \rho_1\ A_1\, v_1 + b_2\, \rho_2\ A_2\, v_2. \tag{2.58}$$

Integrating Equation 2.58 with respect to time

$$\int_t \left(\frac{\partial B_{sys}}{\partial t}\right) = \int_t \left(\frac{\partial B_{CV}}{\partial t} - b_1\,\rho_1\;A_1\,v_1 + b_2\,\rho_2\;A_2\,v_2\right) \tag{2.59}$$

we have the following relationship

$$\frac{DB_{sys}}{Dt} = \int_t \frac{\partial B_{CV}}{\partial t} - \int_t b_1\,\rho_1\;A_1\,v_1 + \int_t b_2\,\rho_2\;A_2\,v_2. \tag{2.60}$$

Simplifying

$$\frac{DB_{sys}}{Dt} = \frac{\partial}{\partial t}\int_{CV} B_{CV} - \int_{CS\,in} b_1\,\rho_1\;A_1\,v_1 + \int_{CS\,out} b_2\,\rho_2\;A_2\,v_2, \tag{2.61}$$

this expression becomes **Reynolds Transport Theorem**

$$\frac{\partial B_{sys}}{\partial t} = \frac{\partial}{\partial t}\int_{CV} b\,\rho\;dV + \int_{CS} b\,\rho\,\mathbf{v}\cdot\mathbf{n}\,dA. \tag{2.62}$$

Figure 2.26 displays a single fluid particle within a infinitesimally small cubical volume.

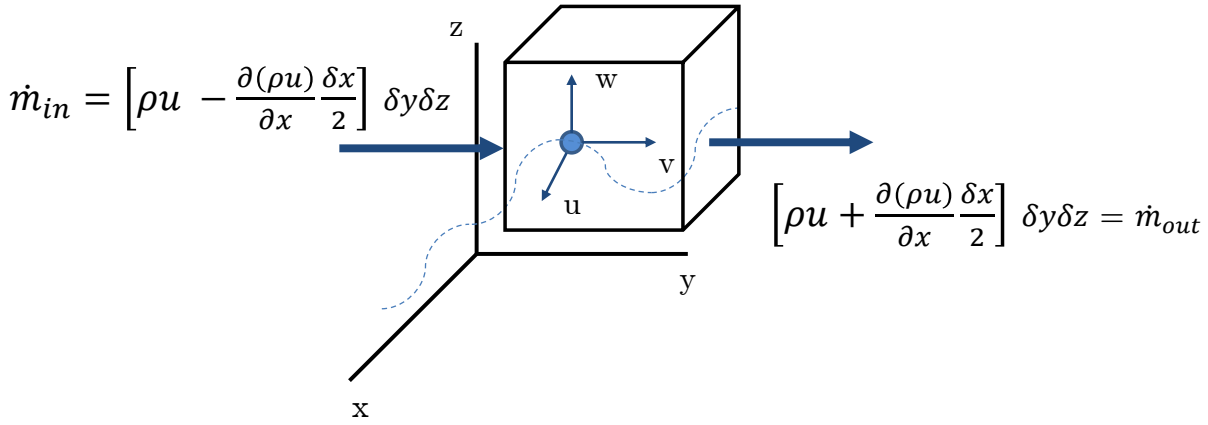



**Figure 2.26:** Conservation of Mass

Applying RTT with $B = m$, $b = 1$ results in the **conservation of mass** equation

$$\frac{\partial m}{\partial t} = \frac{\partial}{\partial t}\int_{CV} \rho\;dV + \int_{CS} \rho\,\mathbf{v}\cdot\mathbf{n}\,dA = 0 \tag{2.63}$$

The mass flow through the control volume in Equation 2.63 can be approximated by

$$\frac{\partial}{\partial t}\int_{CV} \rho \, dV = \frac{\partial \rho}{\partial t}\delta x \delta y \delta z \tag{2.64}$$

Figure 2.26 displays the mass flow rates through in the x-direction. Similar analysis is applied to the y and z directions. The mass flow through the control surface in Equation 2.63 is

$$\int_{CS} \rho \, \mathbf{v} \cdot \mathbf{n} \, dA = \frac{\partial(\rho u)}{\partial x}\delta x \delta y \delta z + \frac{\partial(\rho v)}{\partial y}\delta x \delta y \delta z + \frac{\partial(\rho w)}{\partial z}\delta x \delta y \delta z \tag{2.65}$$

Equation 2.63 then becomes after cancellation of $\delta x \delta y \delta z$

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} + \frac{\partial(\rho w)}{\partial z} \tag{2.66}$$

In vector notation, Equation 2.63 is represented as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \mathbf{v} = 0 \tag{2.67}$$

If the fluid flow is steady ($\rho$ is time-independent), then Equation 2.69 simplifies to

$$\nabla \cdot \rho \mathbf{v} = 0 \tag{2.68}$$

If the fluid is incompressible ($\rho$ is space-independent), then Equation 2.69 simplifies to

$$\nabla \cdot \mathbf{v} = 0 \tag{2.69}$$

Applying RTT with $B = \mathbf{P}$, implies $b = \mathbf{v}$ and results in the **conservation of momentum** equation

$$\mathbf{F} = \frac{\partial \mathbf{P}}{\partial t} = \frac{\partial}{\partial t}\int_{CV} \mathbf{v} \, \rho \, dV + \int_{CS} \mathbf{v} \, \rho \, \mathbf{v} \cdot \mathbf{n} \, dA = 0 \tag{2.70}$$

Forces acting on the fluid particle, represented by $\mathbf{F}$ produce stresses. These stresses are seen in

Figure 2.27 as a normal stress $\sigma_{xx}$ and two shear stresses $(\tau_{xy}, \tau_{xz})$. Relating the forces and the stresses in the positive x-direction

$$\delta F_{xx} = \Big(\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z}\Big)\delta x \delta y \delta z \tag{2.71}$$

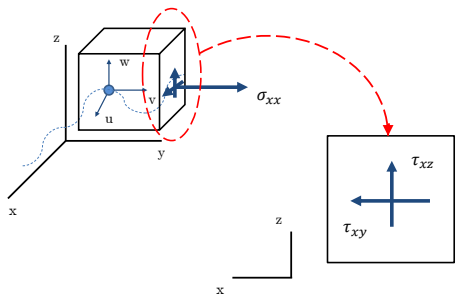



**Figure 2.27:** Conservation of Momentum

From Newton's Second Law $\mathbf{F} = m\mathbf{a}$ in the x-direction where $\delta m = \rho \delta x \delta y \delta z$

$$\rho g_x + \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} = \rho\Big(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial x}\Big) \tag{2.72}$$

Similarly, in the y and z directions

$$\rho g_y + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{zy}}{\partial z} = \rho\Big(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial x}\Big) \tag{2.73}$$

$$\rho g_x + \frac{\partial \sigma_{zz}}{\partial z} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{xz}}{\partial x} = \rho\Big(\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial x}\Big) \tag{2.74}$$

If the fluid flow is inviscid ($\tau$ is negligible), then Equation 2.72 simplifies to

$$\rho g_x + \frac{\partial \sigma_{xx}}{\partial x} = \rho\Big(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial x}\Big) \tag{2.75}$$

Also if the fluid flow is inviscid, the normal stress in Equation 2.75 is represented by a compressive pressure

$$\rho g_x - \frac{\partial p}{\partial x} = \rho\Big(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial x}\Big) \tag{2.76}$$

In vector notation, Equation 2.76 is represented as referred to as **Euler's Equation of Motion**

$$\rho\mathbf{g} - \nabla p = \rho\Big[\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\mathbf{v}\Big] \tag{2.77}$$

For incompressible fluids, the normal stresses are given as

$$\sigma_{xx} = -p + 2\mu\frac{\partial u}{\partial x}\,,\ \sigma_{yy} = -p + 2\mu\frac{\partial v}{\partial y}\,,\ \sigma_{zz} = -p + 2\mu\frac{\partial w}{\partial z}\,, \tag{2.78}$$

and the shear stresses as

$$\tau xy = \tau yx = \mu\Big(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\Big)\,,\ \tau yz = \tau zy = \mu\Big(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\Big)\,,\ \tau zx = \tau xz = \mu\Big(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\Big). \tag{2.79}$$

Plugging these stresses into Equation 2.77 results in the **Navier-Stokes Equations**

$$\rho\Big(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}\Big) = -\frac{\partial p}{\partial x} + \rho g_x + \mu\Big(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\Big). \tag{2.80}$$

Similarly, in the y and z directions

$$\rho\Big(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z}\Big) = -\frac{\partial p}{\partial y} + \rho g_y + \mu\Big(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\Big). \tag{2.81}$$

$$\rho\Big(\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z}\Big) = -\frac{\partial p}{\partial z} + \rho g_z + \mu\Big(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\Big). \tag{2.82}$$

In vector notation, the Navier-Stokes Equations are represented as

$$\rho\Big(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\Big) = -\nabla p + \rho\mathbf{g} + \mu\nabla^2\mathbf{v}. \tag{2.83}$$

The Navier-Stokes Equations (NSE) can also be written as

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{A}(\mathbf{v}) + \nabla p = \frac{1}{Re}\nabla^2\mathbf{v}, \tag{2.84}$$

where $\mathbf{A}(\mathbf{v}) = \nabla(\mathbf{v}\cdot\mathbf{v})$ if the convection term is conservative, and $\mathbf{A}(\mathbf{v}) = (\mathbf{v}\cdot\nabla)\mathbf{v}$ if the convection term is nonconservative. **Reynolds number**, $Re$ is a dimensionless quantity of the fluid flow.

### 3.2.2 Lattice Boltzmann

Lattice Boltzmann methods (LBM) are derived directly from the lattice gas automaton (LGA) and the Bhatnagar-Gross-Krook (BGK) approximation. **Cellular automaton** is defined as a collection of colored cells within a grid that evolve with time according to fixed mathematical rules [137]. These mathematical rules determine the states (on and off) of neighboring cells within the grid. **Lattice gas automaton** is a type of cellular automaton constructed of a lattice used to simulate fluid flow. The mathematical rules are carried out in two main areas, propagation and collision. **Propagation** occurs as each fluid particle moves to a neighboring cell determined by the particles velocity. Barring collisions, a fluid particle will travel in the direction of its velocity. That is, if a fluid particle is traveling to the left, it will move one cell to the left in the next time step, if no collisions take place. **Collision** occurs if multiple fluid particles attempt to occupy the same cell during propagation. The conservation laws (mass, momentum) discussed in the previous section are the mathematical rules that determine the results of all collisions. The **on state** corresponds to the fluid particle occupying the given cell, whereas the **off state** corresponds to the fluid particle not being within the given cell. The **lattice gas automaton (LGA)** model evolves on a two-dimensional triangular lattice. The evolution equation for propagation and collision is given by

$$n_a(\mathbf{x}_i + \mathbf{v}_a, t+1) = n_a(x_i, t) + C_a(\{n_b\}), \tag{2.85}$$

where $n_a(x_i, t)$ is the Boolean particle number 0 (off) or 1 (on), and particle velocity $v_a$ where $a, b$ denote velocities at discrete times. The total number of discrete velocities is given by $d$, thus $a, b = 1, 2, \ldots, d$. The formula for the discrete velocities is given by

$$\mathbf{v}_a = [\cos((a-1)\pi/3)\ ,\ \sin((a-1)\pi/3)]\,, a = 1, 2, \ldots, 6. \tag{2.86}$$

The collision operator, $C_a(\{n_b\})$ takes a value of -1 (move in negative direction after collision) or 0 (no collision) or 1 (move in positive direction after collision).

Figure 2.28 shows two examples of fluid particle motion. Discrete particle velocities are found by applying Equation 2.86.

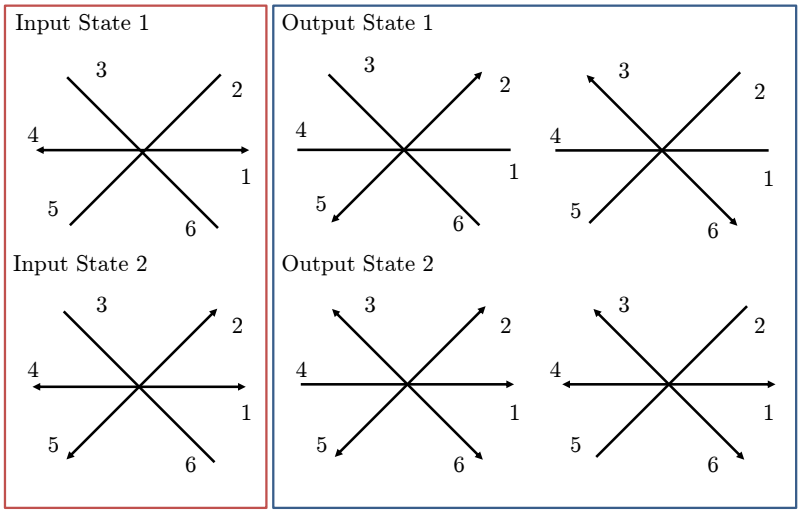



**Figure 2.28:** Lattice Gas Automaton (LGA) Model

$$n_a(\mathbf{x}_i + \mathbf{v}_a, t+1) = n_a(x_i, t) + C_a(\{n_b\}), \tag{2.87}$$

Applying the **Bhatnagar-Gross-Krook (BGK) approximation** to Equation 2.85 yields

$$\frac{\partial f_a}{\partial t} + \mathbf{v}_a \cdot \nabla f_a = \frac{1}{\epsilon\tau}(f_a^{eq} - f_a), \tag{2.88}$$

where $f = f(x, v, t)$ is the distribution function for the fluid particle, $\epsilon$ is a small parameter proportional to Knudsens number, $\tau$ is the relaxation time. The equilibrium distribution function, $f^{eq}$ is given as

$$f_a^{eq} = \omega_a \rho[1 + 3(\mathbf{v}_a \cdot \mathbf{u}) + \frac{9}{2}(\mathbf{v}_a \cdot \mathbf{u})^2 - \frac{3}{2}u^2], \tag{2.89}$$

where $\omega_a$ is the weight along direction $a$, and $\mathbf{u}$ is the fluid velocity, and $\mathbf{v}_a$ is the discrete fluid velocity along direction $a$. Consider the two-dimensional grid shown in Figure 2.29 as an example

of the BGK approximation.

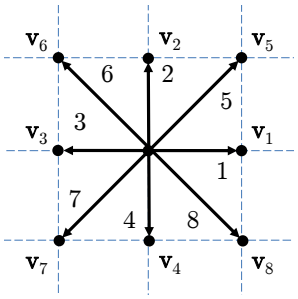



**Figure 2.29:** Bhatnagar-Gross-Krook (BGK) Approximation

Figure 2.29 displays a two-dimensional square lattice with four elements and nine discrete fluid velocities ($a = 9$). The fluid velocities, given by $\mathbf{v}_a$ are

$$\mathbf{v}_0 = 0 \tag{2.90}$$

$$\mathbf{v}_a = \frac{\Delta x}{\Delta t}\left[[cos((a-1)\pi/2)\,,\ sin((a-1)\pi/2)]\,, a = 1,3,5,7\right. \tag{2.91}$$

$$\mathbf{v}_a = \sqrt{2}\frac{\Delta x}{\Delta t}\left[[cos((a-1)\pi/2) + \pi/4\,,\ sin((a-1)\pi/2) + \pi/4]\,, a = 2,4,6,8\right. \tag{2.92}$$

Where $\Delta x$ is the grid size and $\Delta t$ the time step. The weights are

$$\omega_0 = 4/9 \tag{2.93}$$

$$\omega_a = 1/9\,, a = 1,3,5,7 \tag{2.94}$$

$$\omega_a = 1/36\,, a = 2,4,6,8 \tag{2.95}$$

These fluid velocities and weights are then substituted back into Equations 2.88 and 2.89 to solve the LBE. It should be noted, one can apply algebraic manipulations in the form of a Chapman-Enskog Expansion to Equation 2.88 to derive the Navier-Stokes equations. A complete derivation of LBE from LGA and BGK approximations is found in the work carried out by He and Luo [62].

# CHAPTER 3

# LITERARY REVIEW

## 1. GPU Computing for Numerical Linear Algebra and Matrices

*Dense Algebra Overview:* Ltaief et. al [82] implemented a Cholesky decomposition routine MAGMA, that achieved 1.189 TFLOP/s in single precision and 282 GFlops in double precision on 4 NVIDIA GPUs (Tesla S1070). The algorithm optimized by keeping reused data for future computations, rather than having a constant data transfer between CPU and GPU. This technique implemented over half of the matrix on the GPU. Ozcan et. al [111] implemented a CUDA based LU decomposition algorithm with 16 and 32 CUDA cores being utilized to solve system of linear equations (6, 12, 32 and 64 equations). Data for the system was copied first from the CPU to GPU, the algorithm was ran to compute $L$ and $U$ to find the solution vector. The speed-up factors were between 1.77x-4.31x for the systems considered, they were largest when running 16 CUDA cores on the small sets of equations (6 and 12). Volkov et. al [136] implemented CUDA based decomposition algorithms on four NVIDIA GPUs (GeForce and Quadro series). Optimization was developed from SIMD accesses, since the GPU memory bandwidth of the most efficient of the four GPUs was only 88% of the pin-bandwidth (76 GB/s). CPU-GPU data transfer rates were also optimized to obtain up to 90% of the arithmetic peak for the two base operations. Two types of system of equations were solved, a large dense (4096x4096) and small dense (64x16 and 32x32), with emphasis placed on the smaller dense matrices due to possible memory latencies and not fully using the GPU. Speed-up factors of 7.4-8.3x (179-192 Gflops/s) were achieved on dual processor and 3.0-5.5x (34.9-59.2 Gflops/s) on a quad processor. Compared to other works [5, 7], Volkov et. al achieved larger speed-up factors on processors with slower clock speeds and lower pin-bandwidth, attaining 80%-90% of peak speed for large dense matrices. Table 3.1 outlines prior research, with GPU(s) used and corresponding performance metrics for dense linear algebra.

**Table 3.1:** Dense Linear Algebra Selected Prior Work

| **Author [Reference]** | **GPU(s) Utilized** | **Performance Metrics** |
|---|---|---|
| Ltaief [82] | Tesla S1070 | (1.189 TFLOP/s) Cholesky<br>DP(282 GFLOP/s) Cholesky |
| Ozcan [111] | NVIDIA GPU with 16 CUDA cores | 4.31x Speed-Up |
| Volkov [136] | GeForce GTX 8800 | 183 GFLOP/s, 5.5x Speed-Up |

*Sparse Algebra Overview:* Castao-Dez et. al [26] implemented a matrix addition algorithm of two vectors of 256 components that executed in serial on a CPU and in parallel on a GPU. The CUDA thread manager was used to automatically distribute computing tasks to 512 threads. The goal was to maximize global performance by keeping all threads utilized throughout the whole computation. This computation, while low in arithmetic intensity achieved a speed-up factor of 15x. Bell et. al performed sparse matrix-vector multiplication in CUDA for structured and unstructured matrices in single and double precision, achieving up to 81% of the maximum theoretical bandwidth of the GTX 280 without memory caching, with up to 114.78 GFLOP/s. Bell et. al achieved acceleration factors of up to 10.85x and 5.33x compared to Intel Clovertown Xeon and Dual Xeon, respectively [11]. Bell et. al [12] worked on optimizing the Symmetric Matrix Vector product (SYMV) kernel for dense matrices using NVIDIA GTX280 GPUs. Speed-up factors of 35x in single precision were realized using a hybrid algorithm for SYMV that implemented blocking and coalesced memory access. The matrices analyzed in this work consisted of symmetric (which was subjected to memory allocation overhead), and a standard dense matrix. Yan et. al optimized sparse linear algebra algorithms using compressed sparse row format with thread communication and shared memory synchronization, proper memory exploitation ,and thread management with proper choice of warp size. Using matrices from [40], Yan et. al achieved acceleration factors of up to 4x on 4884x4884 sparse matrices (59.45 average nonzeros/row) [141]. Table 3.2 outlines prior research, with GPU(s) used and corresponding performance metrics for sparse linear algebra.

**Table 3.2:** Sparse Linear Algebra Selected Prior Work

| **Author [Reference]** | **GPU(s) Utilized** | **Performance Metrics** |
|---|---|---|
| Castao-Dez [26] | Quadro FX5600 | 15x Speed-Up with CUBLAS |
| Bell [11] | GeForce GTX 280 | SP: 155.1 GB/s with DIA<br>SP: 156.3 GB/s with ELL<br>DP: $\approx$ 140 GB/s with DIA<br>DP: $\approx$ GB/s with DIA |
| Bell [12] | GeForce GTX 280 | 35x Speed-Up |
| Yan [141] | 2 Tesla C1062X | 4x Speed-Up |

*Regular Algebra Overview:* Beliakov et. al [9] performed determinant computations using an NVIDIA Tesla C2070 GPU on large systems (up to 12,000x12,000). Using Gaussian elimination to compute the minors in the determinant expansion, each minor was stored on a separate thread which were stored in non-sequential locations (misaligned) at times. The matrices used in this analysis were large and ill-conditioned, and this degenerate behavior caused questions on the accuracy lost during computation. Altering the algorithm used would improve the accuracy and reduce the error in this work, and it was mentioned that not keeping full accuracy of intermediate values would improve storage requirements and computational cost, allowing the later iterations to have the full precision. Bosilca et. al [19] similarly implemented a Cholesky decomposition hybrid algorithm with MAGMA, but instead focused on small matrices (512x512, 768x768, 1024x1024, and 2048x2048). They achieved speed-up factors of 3x-4x for single-precision and 2x-3x for double-precision algorithms on a single NVIDIA GTX480 GPU. Ezzatti et. al [48] implemented GPU accelerated algorithms, through LU decomposition and Gauss-Jordan elimination to perform matrix inversion. The algorithms were run on a standard multi-core CPU, NVIDIA Tesla C1060 GPU and utilized BLAS and LAPACK as well as a hybrid algorithm through MAGMA. Matrix sizes from 1,000x1,000 up to 14,000x14,000 were computed, with variable block sizes (32 up to

512). The hybrid algorithms developed achieved speed-up factors of 3x-10x, however they were not optimized for small matrices (data transfer between CPU and GPU is not efficient) and double precision was not implemented fully in these works. Song et. al implemented scalability tests (both weak and strong) to analyze algorithm efficiency for precision types (single and double) as well as for core-GPU combination. By introducing a measure for the load imbalance, as a ratio of the maximum load over the average load received by the GPU Song et. al were able to determine runtime system efficiency measures for each setup [126]. Wolf et. al [139] implemented a matrix addition algorithm with CUDA on a GPU cluster (5 machines with 2 Tesla M1060 GPUs each). Similar to Castao-Dez et. al, this work used a middleware (JaMP Runtime System) to ensure all kernels were performing computations in parallel. JaMP partitioned the data over the ten devices, and kernels were co-located automatically to avoid communication between CPU and GPU (array elements were local). The computation of 16 million vector elements being added using two algorithms, *ArraySum* and *VectorAddition* 256 times each resulted in a speed-up factor of 4.4x. Table 3.3 outlines prior research, with GPU(s) used and corresponding performance metrics for regular linear algebra.

**Table 3.3:** Regular Linear Algebra Selected Prior Work

| **Author [Reference]** | **GPU(s) Utilized** | **Performance Metrics** |
|---|---|---|
| Beliakov et. al [9] | Tesla C2070 | 3.95x Speed-Up |
| Bosilca [19] | Tesla S1070 | $\approx$ 2000 GFLOP/s, 4x (SP) & 3x (DP) Speed-Up |
| Ezzatti [48] | Tesla C1060 | 3x-10x Speed-Up |
| Song [126] | 3 Fermi M2070 | SP(1.44 TFLOP/s) Cholesky<br>DP(972 GFLOP/s) Cholesky |
| Wolf [139] | 10 Tesla M1060 | 4.9x Speed-Up |

## 2. GPU Computing for CFD Applications

*Navier-Stokes Overview:* Cohen et. al [33] implemented a Navier-Stokes based CFD solver using an NVIDIA Tesa C1060 GPU for convection problems in 2D and 3D. To compute the Rayeigh values for the convection, Cohen used a sparse matrix solver and developed up to resolutions of 128x64x128. To optimize the performance of the solver, serial bottlenecks were removed, small computations were performed on the CPU, unnecessary data transfers between CPU/GPU were removed and kernels were stored for maximum memory throughput. Speed-up factors of 2.0x-8.5x were realized for single and double precision computations. This work did not optimize for memory bandwidth and with only a single GPU, it was mentioned that these two factors could lead to a much higher acceleration of the solver. Goddeke et. al [56] performs driven cavity and channel flow around a cylinder for unmodified finite-element NS equations. Thibualt et. al [132] also implemented a Navier-Stokes based CFD solver using an NVIDIA Tesla S870 server to solve 3D incompressible flow problems. This work took three-dimensional data and stored it in two-dimensional matrices in global memory, with different matrices storing different part of the N-S equation (pressure, velocity) as a function of time. This was implemented for a single GPU, and then repeated for a multi-GPU approach. Memory allocation was done using two different domain decompositions, with only up to 22% of shared memory being updated at all times. Solving a lid-cavity problem, speed-up factors of 16x-33x using a single GPU. Table 3.4 outlines prior research, with GPU(s) used and corresponding performance metrics for Navier-Stokes.

**Table 3.4:** Navier-Stokes Selected Prior Work

| **Author [Reference]** | **GPU(s) Utilized** | **Performance Metrics** |
|---|---|---|
| Cohen [33] | Tesa C1060 | 2.0x-8.5x Speed-Up |
| Goddeke [56] | GeForce GTX 8800 | 2.25x Speed-Up |
| Thibualt [132] | Tesla S870 | 183 GFLOP/s, 16x-33x Speed-Up |

*Lattice Boltzmann Methods Overview:* Bernaschi et. al [15] introduced a multi-physics simulation with CUDA to optimize MUPHY for irregular grid domains. Obrecht et. al [109] implemented a LBM based CFD solver that used 6 NVIDIA Tesla C1060 GPUs to solve a lid-cavity problem. To optimize for performance emphasis was placed on groups of threads into single transactions of 32, 64 and 128 bits, and allocating the rest of the threads to be placed in blocks using shared memory. The analysis was carried out in single precision, mixed precision and double precision, with an absolute error less than 0.5%. While this work had much computational power, performance gains usually seen on multi-GPU systems were not present in this instance. The data throughput showed only 48.% of the maximum value for a single GPU showing that the bound placed was computational and not memory based. Tolke [133] introduced a LBM based CFD solver that utilized a single NVIDIA G80 GPU to compute flow through a generic porous medium. In this setup, three main kernels are introduced to compute collision, synchronize thread distribution and store boundary conditions and are stored in 12x12 matrices. Performance gains of up to 1x faster for mesh sizes (512, 1024, 2048 and 3072) for thread counts (32, 64, 128, 192, 256) were realized. Shared memory was not used for the propagation, and the amount of solid nodes present in the work were more than desired and it was mentioned that a smaller decomposed domain could fix these issues. Table 3.5 outlines prior research, with GPU(s) used and corresponding performance metrics for Lattice Boltzmann Methods.

**Table 3.5:** Lattice Boltzmann Methods Selected Prior Work

| **Author [Reference]** | **GPU(s) Utilized** | **Performance Metrics** |
|---|---|---|
| Bernaschi [15] | GeForce GT200 | 955 MLUP/s |
| Obrecht [109] | 6 Tesa C1060 | $\approx$ 200 MLUP/s |
| Tolke [133] | GeForce 8800 Ultra | 670 LUP/s |

# CHAPTER 4

# METHODOLOGY

In this chapter the methodology for this work is discussed. The numerical algorithmic development of the matrix computations corresponding to the two base operations is outlined. Matrix parameters (matrix types, element types) will be overviewed. Methods to measure precision and accuracy, performance and speed-up factors of algorithms are discussed. Prior work and test matrices to ensure validity of the algorithms are mentioned. Computing resources used in this work will be listed, and the CFD applications to numerical linear algebra will be stated.

Overall this work had three primary application and verification stages. Application stage one began with building matrix algorithms for GPU implementation in CUDA C. Verification stage one introduced small test matrices which ensured validity of these algorithms. Application stage two continued with analysis of performance and optimization techniques for the developed algorithms in stage one. Verification stage two followed with comparison of acceleration rates and bandwidth analysis. Application stage three concluded with Navier-Stokes and Lattice Boltzmann Methods applied to structured and unstructured grids, using matrix methods of stage one and performance techniques of stage two. Verification stage three followed with comparison to accepted prior results and measuring GPU impact. Figure 4.1 outlines the overall thesis process flow.

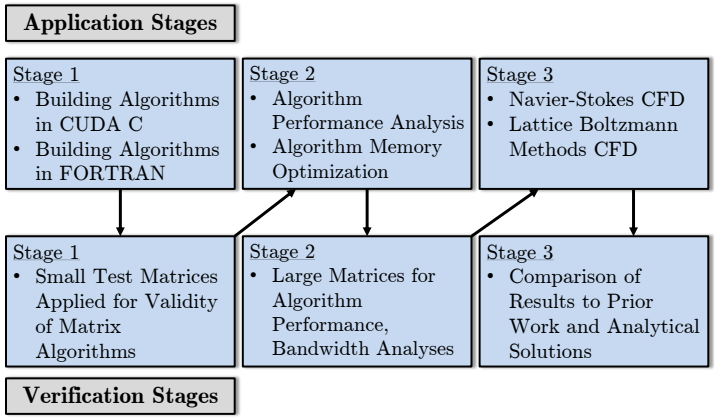



**Figure 4.1:** Thesis Outline

### 1. Computational Linear Algebra

The primary focus of the computational linear algebra field was to collect desired test matrices from the NIST Matrix Market. The matrices used in this work come from three primary types: dense, banded and sparse. Dense and banded matrices were generated through the Java applets available on the NIST website. Since these matrices were generated and do not possess any special characteristics or distinctions, their information is not presented. The sparse matrices however came from test data and real applications, and can be found on the main NIST mathematics website [91]. The NIST Matrix Market presents the matrix name (which can be used to determine the matrix family collection), the size of matrix (the row and column number) as well as the type (Real Unsymmetric, Real Symmetric, Pattern Symmetric Indefinite), nonzeros (number of nonzero elements in the matrix), column (average number of nonzero elements per column), row (average number of nonzero elements per row), bandwidth (average of the distance of nonzero elements from the main diagonal), and the conditioning information (condition number and diagonal dominance). A level of sparsity in a given matrix can easily be determined by taking the number of nonzeros and dividing it by the square of the size (since all matrices considered are square). Additional information is available for each matrix such as the ten most important (heaviest) diagonals, lower and upper bandwidths, weight of longest and shortest column/row, and Frobenius norm [40, 91].

Certain matrix groups display unique characteristics that allow for ideal analysis scenarios. These groups can be classified into two types: variable matrix size and variable matrix conditioning number. The former is seen with the BCSSTK, BSCPWR, DWT, SHERMAN and JAGMESH grops. The latter is seen with the E05, E20, E30, E40 and E50 groups. These matrices come from an array of unique scientific applications: BCSSTK (finite element), DWT (structural engineering), SHERMAN (oil and petroleum numerical analysis), JAGMESH (finite element).

All matrices considered are of the real type, instead of the complex type, with certain matrices having symmetric properties. This was done to eliminate complex roots in solutions for certain

matrix operations and applications, as well as provide a measure for comparison of prior work.

A quick visualization tool used to determine the sparsity and pattern of a matrix is found from cityplots. **Cityplots** are graphs that display the elements of a matrix in colors based on the relative magnitude of the matrix entries. The largest values in magnitude are denoted red and represented with the largest height from the surface, whereas the smallest values in magnitude are denoted blue and are represented with the smallest height near the surface. Zero values are left white. Thus, sparse matrices display large amounts of white space and dense matrices displays small amounts of white space. Figure 4.2 displays an example of a dense, banded and sparse matrix.

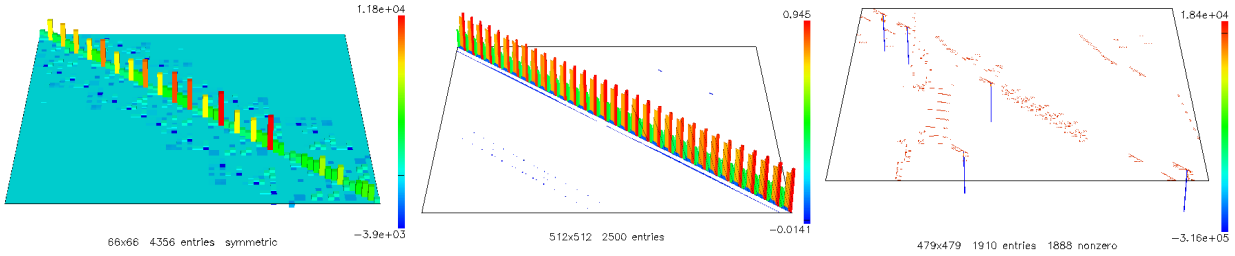



**Figure 4.2:** CityPlots: Dense (Left), Banded (Middle) and Sparse (Right)

Virtual Reality Modeling Language (VRML) software enables 3D interactive options for matrices. FreeWRL is an free open source software that was used to view the given matrices. The advantage of using VRML is that any user has the capability to view a given matrix at any angle, a possibility that is not possible with a cityplot image. Figure 4.3 displays an example of an interactive matrix.

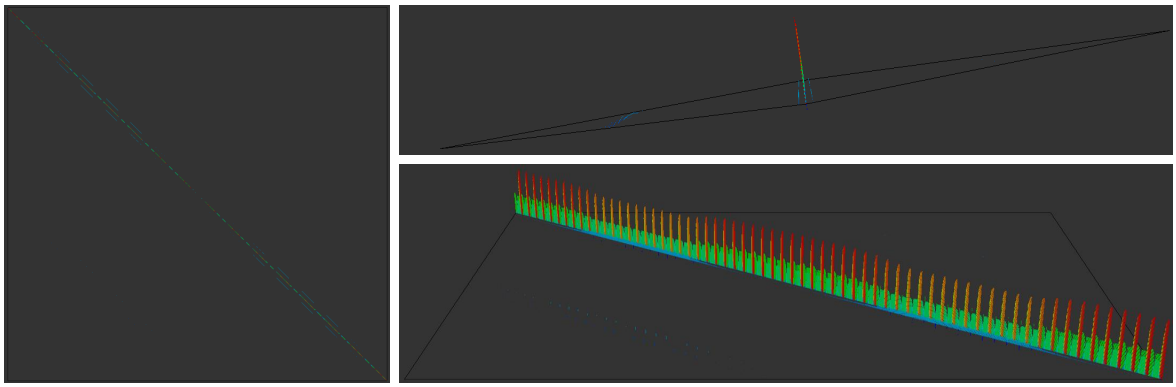


**Figure 4.3:** 3D Interactive Matrix Plot: Top (Left), Size Views (Right)

## 2. GPU Integration

Initially the computer which GPGPU is being carried out is verified to have a proper NVIDIA CUDA-enabled GPU by using the Device Manager as shown in Figure 4.4. Graphic processors that are not CUDA-enabled, such as Intel(R) HD Graphics 4000 are not to be utilized for GPGPU and have no effect on the performance of programmed algorithms.

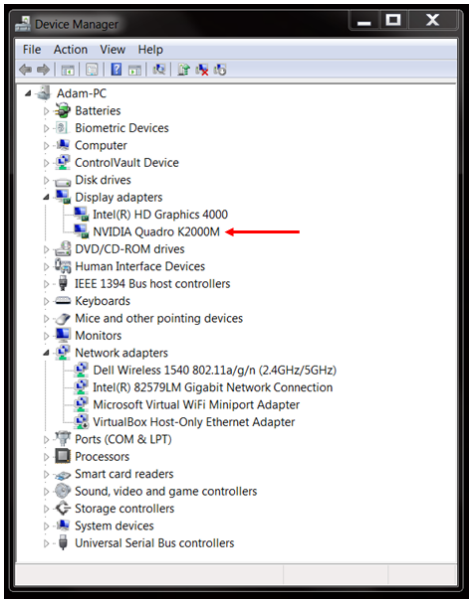



**Figure 4.4:** GPU Detection K2000M: Device Manager

After verification of a programmable CUDA-enabled GPU, installation of CUDA 5.5 which includes the CUDA Toolkit, SDK code samples, Nsight Visual Studio Edition and developer drivers is necessary. This installation is available for free online in NVIDIA's CUDA ZONE. The CUDA Toolkit provides the development environment to build GPU-accelerated algorithms in C and C++. It provides a compiler, math libraries, and debugging and optimization tools for the developed algorithms. The CUDA Toolkit also comes equipped with programming guides, user manuals and API reference documentation. The SDK (Standard Development Kit) code samples include applications for simple reference (CUDA key concepts and runtime APIs), utilities (query device capabilities, measure CPU/GPU bandwidth), graphics, imaging (processing and data analysis), fi-

nance, simulations, and Cudalibraries (CUBLAS, CUFFT, etc.). A detailed explanation of the CUDA samples can be found in the Reference Manual [99]. Lastly, Nsight Visual Studio provides GPGPU integration into Microsoft Visual Studio. Nsight Visual Studio is currently the industry's only GPU hardware debugging solution, has application and system trace for a kernel timeline, and has a graphics profiler that determines automatic bottleneck and performance measurements. Figure 4.5 shows the CUDA sample deviceQuery which is run through the Windows 7 Command Prompt. CUDA sample deviceQuery provides the basic capabilities of the GPU, with memory specifications for kernels, blocks, and threads as well as clock rates and cache sizes. The version of CUDA (5.5) installed is also shown with the driver and runtime versions.

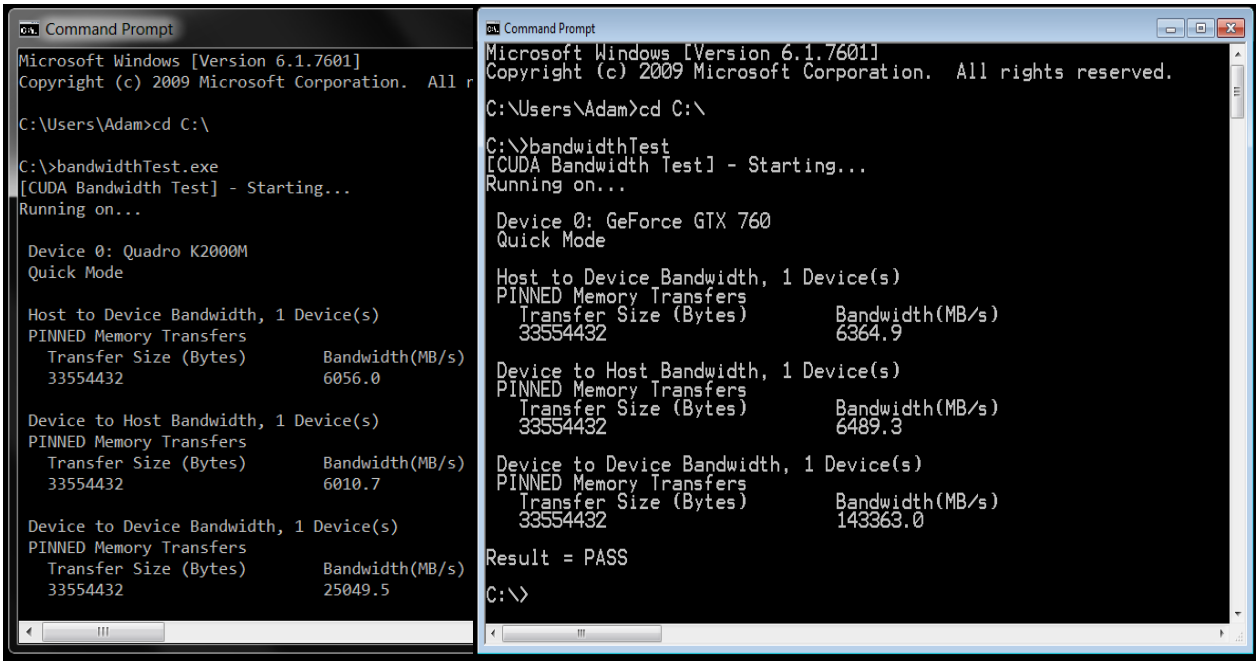



**Figure 4.5:** CUDA Samples: deviceQuery K2000M (Left), GTX760 (Right)

Figure 4.6 shows the CUDA sample bandwidthTest which is also run through the Windows 7 Command Prompt. CUDA sample bandwidthTest provides the optimal memcopy bandwidth of the GPU and the memcpy bandwidth across the PCI-e 3.0 expansion bus. The memory bandwidth of data transfer from CPU to GPU, 6056.0 MB/s is almost identical to the memory bandwidth of

data transfer from GPU to GPU, 6010.7 MB/s. These values are about four times smaller than the internal memory bandwidth of the GPU, 25049.5 MB/s. The discrepancy in memory bandwidth is one of the main reasons behind keeping the communication and data transfer between the CPU and GPU at a minimum, and performing most computations solely on the GPU. Overall the CUDA Toolkit targets parallel algorithms using one or more GPUs as coprocessors. GPU algorithms executed entirely on the GPU without intervention of the host achieve maximum acceleration rates if implemented properly using this toolkit. Communication between the CPU and GPU is possible through a process called remote procedure calling. In remote procedure calling, the host can dispatch GPU jobs through the CUDA Toolkit.

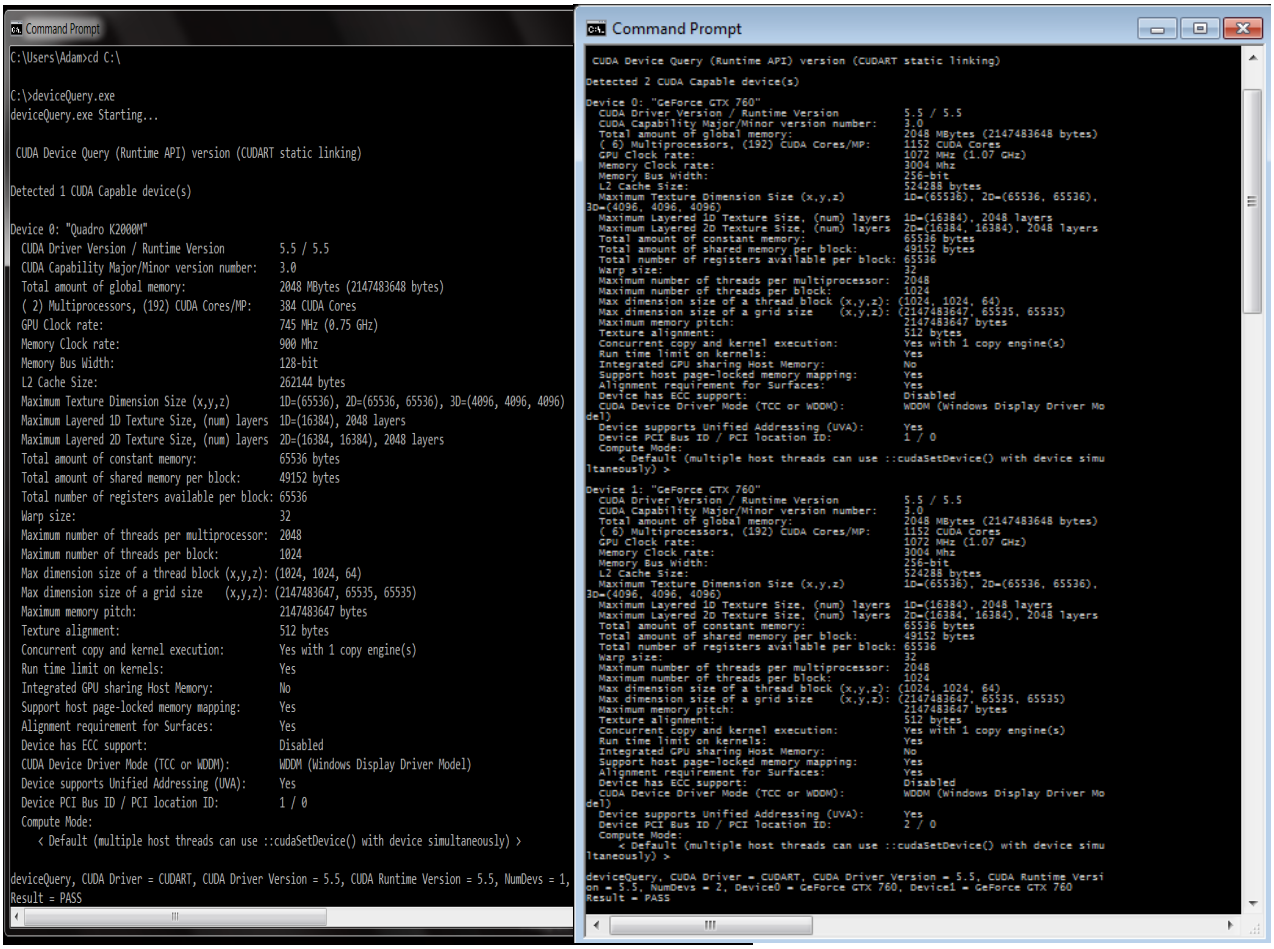



**Figure 4.6:** CUDA Samples: bandwidthTest K2000M (Left), GTX760 (Right)

Figure 4.7 displays the CUDA design cycle: assess, parallelize, optimize, and deploy (APOD). APOD is an iterative process. Code is first parallized, initial accelerations are achieved and deployed and assessments are made. Optimization are realized, code is modified, additional accelerations are achieved, and faster algorithms are deployed. The first step in APOD is to access the algorithm for routines that take the majority of the code compilation time. These bottlenecks are used with Amdahl's and Gustafon's laws to determine to what extent the code is being compromised. Parallelized code is then constructed of the desired algorithms, using optimized NVIDIA libraries. Optimization of the code is next step in APOD, where speedup strategies are incrementally applied and analyzed for effectiveness. Deployment of the code occurs last, if the accelerated code is accepted the process ends and if greater acceleration rates are desired the process begins again [98].

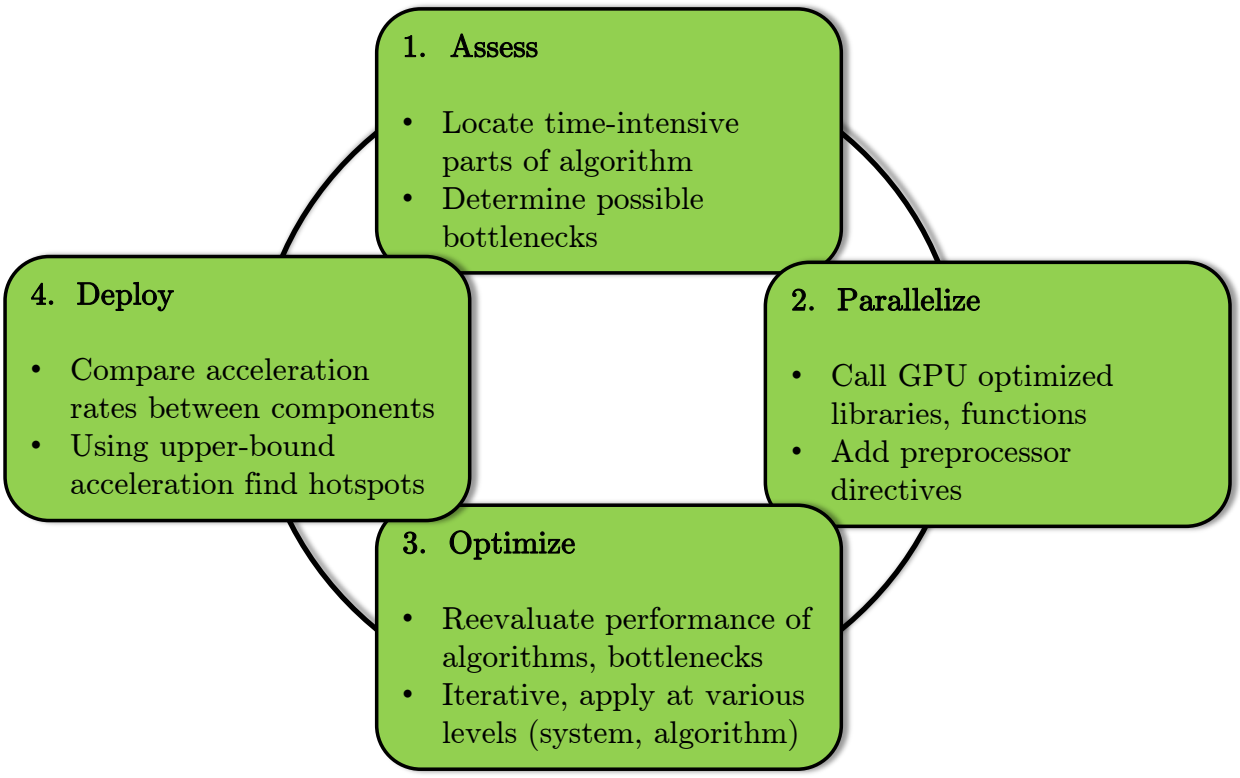



**Figure 4.7:** CUDA Design Cycle

## 2.1 Scalable Link Interface (SLI)

Scalable Link Interface (SLI) is an NVIDIA developed technology for multiple GPU inputs to produce a single output. SLI is used to increase computing power for parallel applications, showing a linear trend between number of GPUs and processing power [42]. This phenomenon of increased computing benefits from additional GPUs is known as scaling. The requirements to utilize the advantages of an SLI GPU system are the following, first all graphics cards must have an identically named GPU. This is the primary reason behind the choice to use two GeForce GTX 760s. Second, all GPUs must have the same amount of VRAM and an identical bus width. This is the primary reason behind the choice for two GTX 760's with 2GB of VRAM. Lastly, one needs an SLI bridge and compatible motherboard.

SLI works ideally when the setup being used is GPU limited, rather than CPU limited. Figure 4.8 displays these two states. The GPUs needs to be the bottleneck of the system, as shown in the GPU limited setup, the device waiting from instructions from the CPU to perform the necessary calculations. If the GPUs are positioned in the CPU limited setup, the GPUs are not properly being utilized and the advantages of scalability and multiple GPU setups are being lost.

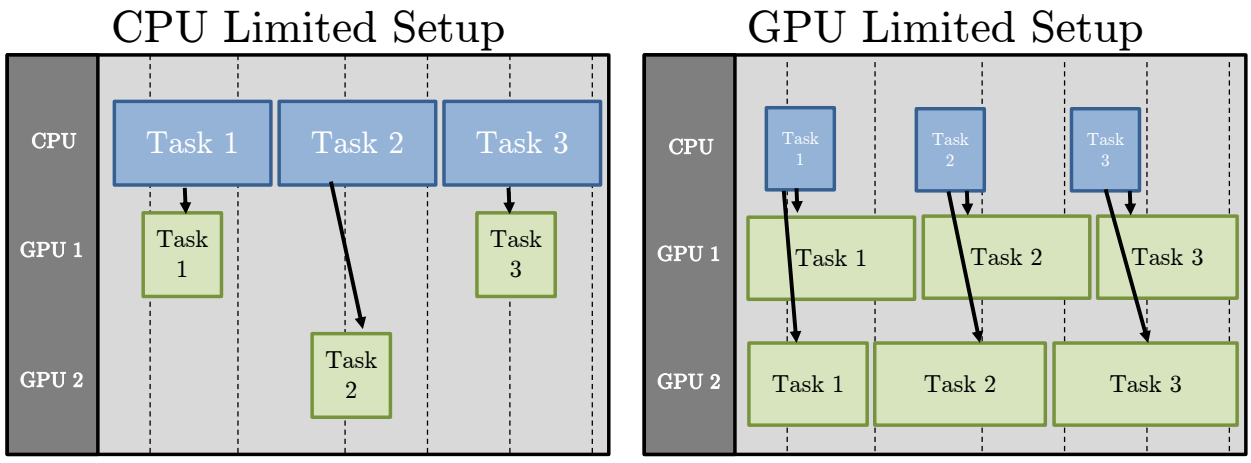
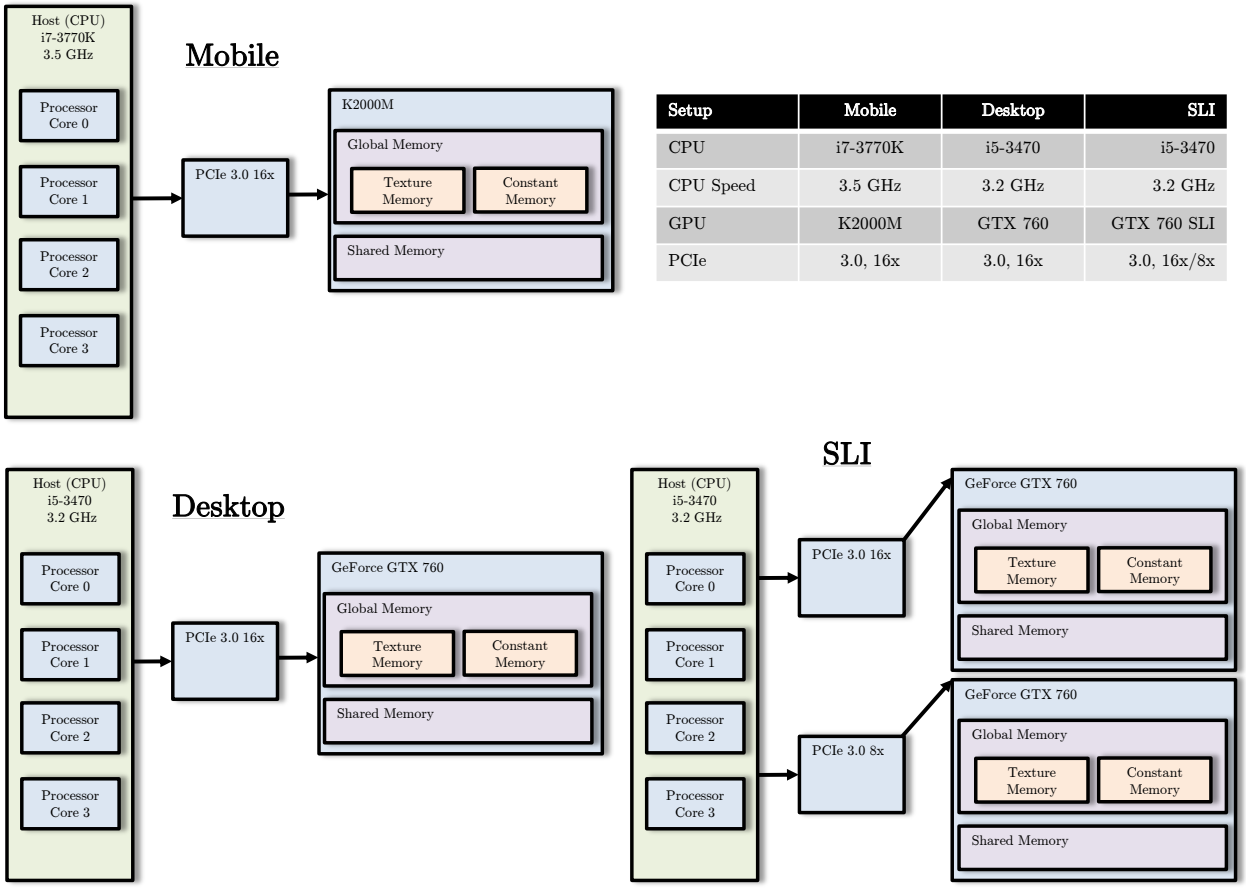



**Figure 4.8:** CPU (Left) and GPU (Right) Limited Setups

SLI connected GPUs are placed in the motherboard through Peripheral Component Interconnect Express (PCI Express) ports. These ports are classified based on number of lanes and the transfer speed. PCIe slots contain one to thirty-two lanes, in powers of two (1, 2, 4, 8, 16, and 32). GeForce GTX 760 graphic cards occupy 8 lanes, thus they can be placed in either 8, 16 or 32 slot ports. More importantly, PCIe slots possess data transfer speeds of 4, 8 and 16 GB/s for the 8, 16 or 32 slot ports, respectively. To ensure that a GPU is not limited by the motherboard, a GPU should be placed in the PCIe slot with the largest number of ports and the fastest transfer speed. This is the primary reason behind choosing the ASUS P8Z77-V LK Z77 Motherboard. Figure 4.9 displays the three setups and their respective PCIe connections, with CPU name and speed.




| Setup | Mobile | Desktop | SLI |
|---|---|---|---|
| CPU | i7-3770K | i5-3470 | i5-3470 |
| CPU Speed | 3.5 GHz | 3.2 GHz | 3.2 GHz |
| GPU | K2000M | GTX 760 | GTX 760 SLI |
| PCIe | 3.0, 16x | 3.0, 16x | 3.0, 16x/8x |

**Figure 4.9:** PCIe GPU Setups: K2000M (Top Left), GTX 760 (Bottom Left), GTX 760 SLI (Bottom Right)

## 2.2 Algorithm Verification

While accelerating algorithms is of interest in this work, obtaining correct results and proper verification methods takes precedent over any realized speedup factors. Also issues arise when performing parallel computations that are not traditionally encountered when performing serial-oriented programming on the CPU. Some of these issues are unexpected floating-point computations, threading complications, and differences in operation techniques of the CPU and GPU. Established mechanisms take previous accepted output values from representative inputs, and compare them to new output values. After each change is made in the process, these mechanisms are applied and analysis is performed to decide if the algorithm is sound. It is important to note that algorithms involving floating-point values may not exactly compare to the reference output values, as an accepted level of computer precision error is determined to answer for the soundness of the algorithm. Another technique used for code verification is unit testing. Unit testing is writing CUDA kernels such that they can be verified individually and on both hardware. Writing short __device__ kernels rather than one large __global__ function and __host__device__ rather than __device__ functions are examples of these two types of unit testing.

Regression testing, writing small portion of code and performing verification before adding additional code is a smart programming technique for easy debugging and ensuring proper results [50]. Regression testing is implemented throughout this work to verify the correct output of all matrix operations and applications, as well as the CFD application results. As with all cases of programming, however there are instances that codes do not deliver the desired results. In these cases, debugging is accomplished through Microsoft Visual Studio's Parallel Nsight. Breakpoints are first set in critical sections of the code, next the sample is built and the debugger launched. Values for variables and values in memory are inspected for validity. The NVIDIA Nsight User Guide provides more details into debugging process and code optimization [101].

## 2.3 Algorithm Performance Metrics

Measuring algorithm performance through code execution time can be accomplished with CPU and GPU timers. CPU timers measure the elapsed time of CUDA calls and kernel executions. CUDA API functions then return back to the CPU thread prior to their completion, thus to properly assess algorithm times CPU threads must be synchronized with the GPU by using cudaDeviceSynchronize() before and after the CPU timer. Theoretical Bandwidth of a GPU is found by applying Equation 4.1.

$$\text{Theoretical Bandwidth = Memory Clock Rate x Bus Width,} \tag{4.1}$$

where standard units for memory clock rate is Hz, bus width is bytes and theoretical bandwidth is GB/s. The theoretical bandwidths for the Quadro K2000M, and GeForce GTX 760 are

$$[1800 \times 10^6 \times (128/8)]/(10^9) = 28.8 \text{ GB/s.} \tag{4.2}$$

$$[6008 \times 10^6 \times (256/8)]/(10^9) = 192.256 \text{ GB/s} \tag{4.3}$$

This is found by taking the memory clock rate (1800 MHz, 6008 MHz) and converting it to Hz, taking the bus width (128-bit, 256-bit) and converting it to bytes, and then dividing by $10^9$ for standard units of GB/s. These values are found in Table 4.2. Effective Bandwidth of a GPU is found by applying Equation 4.4.

$$\text{Effective Bandwidth} = [(B_r + B_w]/(10^9)/t, \tag{4.4}$$

where $B_r$ is the number of bytes read per kernel, $B_w$ is the number of bytes written per kernel and $t$ is the amount of time to read and write a kernel.

Ideally, effective bandwidth is equal to theoretical bandwidth, however this is not the case for most GPUs. The following steps can be taken to maximize bandwidth and minimize the difference between two type calculations. These steps are using as much fast-memory and as little slow-access memory as possible. Since CPU/GPU data transfers are much slower than GPU/GPU data transfers, kernels are kept on the GPU even if they don't demonstrate any speedup compared to running them on the CPU. This avoids unnecessary data transfers that slows down execution time and reduces the advantages of the GPU. Small data transfers are combined into one large transfer to eliminate any overhead associated with each small transfer, even in the case that non-contiguous regions of memory are combined into a contiguous buffer and then unpacked after the transfer. Page-locked (or pinned) memory is used as it provides high bandwidth, but used acutely as it limited and difficult to determine prior to running the algorithm the proper amount to use. Operations in different streams (sequence of operations that are performed on the GPU) are interleaved and overlapped, allowing data transfer latencies between the CPU and GPU to be hidden [98].

Another form of analyzing the performance of GPU algorithms is found in the Compute Command Line profiler in the CUDA Toolkit. The command line profiler can be used to assess algorithm speeds and locate possible bottlenecks, and is accomplished through environmental variables displaying kernel execution and memory transfer information. If environmental variable COMPUTE_PROFILE is set to 1, the command line profiler will log records of timing information for each kernel launch and memory operation performed. Figure 4.10 shows the standard log file after running code through the command line profiler. The four default columns are method, gputime, cputime and occupancy. Method is a label specifying the name of the memory copy method or kernel execution. Gputime is a label specifying the actual chip execution time (in ms) of the memory copy method or kernel execution. It is computed by taking [gpuendtimestamp - gpustarttimestamp]/1000.0. Cputime is a label specifying the actual driver execution time (in ms). It is computed two different ways, depending on the type of method (non-blocking or blocking)

chosen for the algorithm. For non-blocking methods: walltime = cputime + gputime and for blocking methods: walltime = cputime. Occupancy is a label specifying the ratio of number of active warps per MP to the maximum number of active warps per MP. Occupancy is a performance measure that determines efficiency of GPU kernels, ranging from 0.0 (totally inefficient) to 1.0 (totally efficient) [92].

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2075
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6de60e24570
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 80.640 ] cputime=[ 278.000 ]
method=[ memcpyHtoD ] gputime=[ 79.552 ] cputime=[ 237.000 ]
method=[ _Z6VecAddPKfS0_Pfi ] gputime=[ 5.760 ] cputime=[ 18.000 ]
occupancy=[ 1.000 ]
method=[ memcpyDtoH ] gputime=[ 97.472 ] cputime=[ 647.000 ]
```

**Figure 4.10:** Default Log File for Compute Command Line Profiler

Additional command line profiler commands are found in Table 4.1. Each command is used for launching kernels, and are not inclusive of the four commands shown in Figure 4.10.

**Table 4.1:** Comprehensive Command Line Profiler Commands

| **Command** | **Description** |
|---|---|
| cacheconfigureexecuted | Cache configuration used for kernel launch |
| cacheconfigurerequested | Requested cache for kernel launch |
| countermodeaggregage | Aggregate counter for multiple devices |
| conckerneltrace | Computes gpu start/end timestamps for concurrent kernels |
| dynsmemperblock | Size of dynamically allocated shared memory per block |
| gpustarttimestamp | Time stamp when kernel execution/memory transfer begins |
| gpuendtimestamp | Time stamp when kernel execution/memory transfer ends |
| gridsize | number of blocks per grid along x and y directions |
| gridsize3d | number of blocks/grid along x, y and z-directions |
| memtransferdir | memory transfer direction |
| memtransfersize | memory transfer size between source and target |
| memtransferhostmemtype | host memory type (pageable or page-locked) |
| regperthread | Number of registers used per thread |
| stasmemperblock | Size of statically allocated shared memory per block |
| streamid | Stream id |
| threadblocksize | number of threads per block along x, y and z-directions |
| timestamp | Similar to gpustarttimestamp, less accurate |

## 2.4 GPU Comparison

Table 4.2 displays a comparison of specifications between the two NVIDIA GPUs used in this work: Quadro K2000M and GeForce GTX 760. The K2000M is exclusively for mobile workstations, whereas the GTX 760 is exclusively for desktop workstations. Both GPUs operate on NVIDIAs Kepler architecture, which is optimized for efficiency, programmability with CUDA and performance. This was an update from the penultimate NVIDIA architecture, Fermi which was focused on increasing pure computing performance for both compute and tessellation. Being a desktop GPU, the GTX 760 is larger (9.5 in x 4.376 in) than the K2000M. This results in the GTX 760 having approximately three times as many resistors as the K200M. The increased amount of resistors allows the GTX 760 to operate at a much high memory clock rate (6008 MHz) and memory bandwidth (192.26 GB/s) than the K2000M. When considering memory type, GDDR5 memory has bandwidth advantages that comes with latency concerns whereas DDR3 memory has latency advantages however is much slower. An address bus is a computer bus architecture used to transfer data between hardware (CPU and GPU) on a computers motherboard. The width of the address bus determines the amount of memory that a computer can address. The 128-bit address bus on the K2000M can address $2^{128}$ memory locations, similarly the 256-bit address bus on the GTX 760 can address $2^{256}$ memory locations. CUDA Cores are related to the computing capacity available in a GPU, as the GTX 760 has a higher ability to accelerate algorithms. A ROP (Raster Operator) are units that deal with pixel generation and coloring on the screen, and relate directly to the pixel and texture fill rates. The increase from 16 ROPs on the K2000M to 32 ROPs on the GTX 760 is responsible for the increase in the two fill rates. The PCIe (Peripheral Component Interconnect Express) is the bus located on the motherboard which provides the connection to the GPU. PCIe 3.0 (Base Clock Speed: 8 GHz, Data Rate: 1000 MB/s, Total Bandwidth: 32 GB/s and Data Transfer Rate: 8 GT/s) have architectural improvements to the PCIe 2.0 (Base Clock Speed: 5 GHz, Data Rate: 500 MB/s, Total Bandwidth: 16 GB/s and Data Transfer Rate: 5 GT/s).

**Table 4.2:** CPU/GPU Specifications

| **GPU Name** | Quadro K2000M | GeForce GTX 760 |
|---|---|---|
| **Architecture** | NVIDIA Kepler | NVIDIA Kepler |
| **Transistors** | 1.3 Billion | 3.54 Billion |
| **Launch Date** | 06/2012 | 06/2013 |
| **GPU Clock Rate** | 745 MHz | 980 MHz |
| **Global Memory** | 2048 MB | 2048 MB |
| **Memory Type** | DDR3 | GDDR5 |
| **Memory Clock Rate** | 1800 MHz | 6008 MHz |
| **Memory Bus Width** | 128-bit | 256-bit |
| **Memory Bandwidth** | 28.8 GB/s | 192.26 GB/s |
| **CUDA Cores** | 384 | 1152 |
| **ROPs** | 16 | 32 |
| **Texture Mapping Units** | 32 | 96 |
| **Maximum Power Draw** | 55 W | 170 W |
| **Pixel Fill Rate** | 11.92 Gpixels/s | 33 Gpixels/s |
| **Texture Fill Rate** | 23.84 Gtexels/s | 94 Gtexels/s |
| **SP Compute Power** | 527.16 GFLOPS | 2258 GFLOPS |
| **Max. Threads/Block** | 1024 | 1024 |
| **32-bit Registers/Thread** | 63 | 63 |
| **Max. Shared Memory/MP** | 48 KB | 48 KB |
| **Shared Memory Banks** | 32 | 32 |
| **Local Memory/Thread** | 512 KB | 512 KB |
| **Surfaces/Kernel** | 8 | 16 |
| **SPs** | 48 | 192 |

## 3. CFD Applications

CFD develops numerical solutions for differential equations governing mass, momentum and energy conservation in fluid mechanics. Prior to the advent of CFD, engineering design used to rely exclusively on empirical data in the form of correlations and handbook tables. Empirical data is only applicable to the limited range of scales for which they are collected, thus groundbreaking designs were forced to rely solely on scaling laws for approximations. Scaling laws become critical as electronic equipment becomes miniaturized and architectural structures grow in size. Ideally, designers desire scale-neutral design tools that fit expansive ranges of physical phenomena.

Overall, there exists two solution types for mathematical, scientific and engineering problems. These two types are referred to as analytical and numerical solution methods. Analytical (or exact) solutions are desired as they contain no error or approximations, and are found for simple but not complex analyses. Analtyical solutions are continuous in both space and time. Numerical (or approximate) solutions are more common in analyses, expansive in methodology for solution but contain error. Numerical solutions are discrete in both space (mesh size) and time (time step). Numerical solutions are sought in CFD because conservation equations are multi-dimensional, strongly coupled or nonlinear, and contains complex solution domains. Numerical solutions become analytical solutions when simplified through approximations. [38]

The numerical solution process is as follows, first given flow problem, define the physical (space and time) domain of interest. Second, apply relevant conservation equations, define boundary conditions and fluid properties. Next, develop grid within the domain with nodes at discrete points. Following, apply approximations to convert partial differential equations into a system of linear equations. Furthermore, decide on discretization method to solve (finite element, difference, volume). Additionally, develop algorithms to implement chosen discretization method with given problem and boundary conditions. Then, analyze the results obtained from running the algorithms. Finally, determine if results are acceptable, if needed change grid size or time step. [38]

The components of a numerical solution method are presented in Figure 4.11. First, a mathematical model, a set of equations and boundary conditions defines the problem. Based on the desired application, simplifications of equations are applied. A general solution is not found, rather a simplified solution for a given application. A numerical solution method possesses a discretization method, a method of approximating the differential equations by algebraic equations (finite element, finite difference, finite volume). A coordinate and basis vector system is established (rectangular, cylindrical, spherical) for the problem domain. Lastly, a grid pattern is established for the problem. Structured and unstructured grid patterns are the two primary types used in CFD.

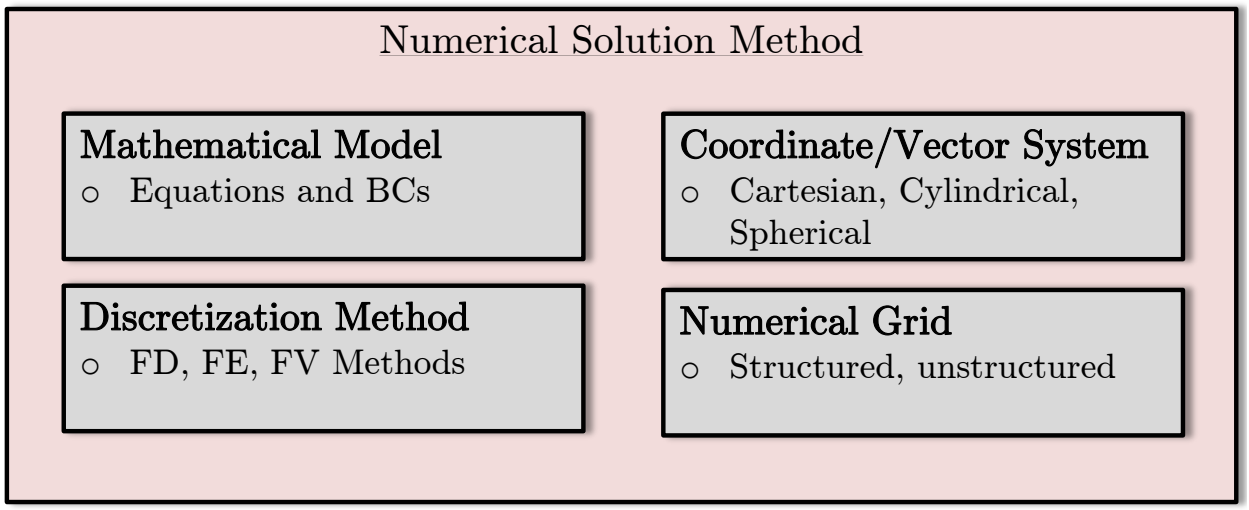



**Figure 4.11:** Numerical Solution Method

For this work the mathematical models were derived from the Navier-Stokes equations and Lattice-Boltzmann methods for two dimensional flows. The boundary conditions were arrived at based on experimental data that relates to the physics of the setup. The coordinate systems were chosen such that they made the analysis of the problem the easier to implement numerically, as well as arrive at a valid solution efficiently computationally. The discretization method chosen to analyze the problem relates again to the type of problem, geometry and physics of the problem. Numerical grids are structured in cases that the whole problem of interest is important, whereas unstructured grid patterns are used when certain regions retain higher importance than other regions.

## 3.1 Steady Flow Past a Cylinder

This work studied two CFD applications, the first being steady flow past a cylinder shown in Figure 4.12. The diameter of the cylinder, $d$, was kept constant at 1 m. The inlet velocity, $v_x$ was also kept constant at 1 m/s, as well as the density, $\rho$ = 1 kg/m$^3$. The kinematic viscosity, $\nu$ was varied to obtain desired values for the Reynolds number, Re using

$$Re = \frac{\rho\, v_x\, d}{\nu}. \tag{4.5}$$

Experimental data from flow visualizations show the formation of vortices and transformation from steady and symmetric to unsteady and asymmetric with increasing Reynolds number [35, 128]. A no-slip boundary condition is applied at the wall of the cylinder, and uniform free stream conditions are applied to the fluid at both the inlet and outlet boundaries [130]. Mapped face meshing as well as circumferential and radial edge sizings were applied to create the mesh. In total there were 192 divisions and 96 divisions along the along the circumferential and radial directions, respectfully.

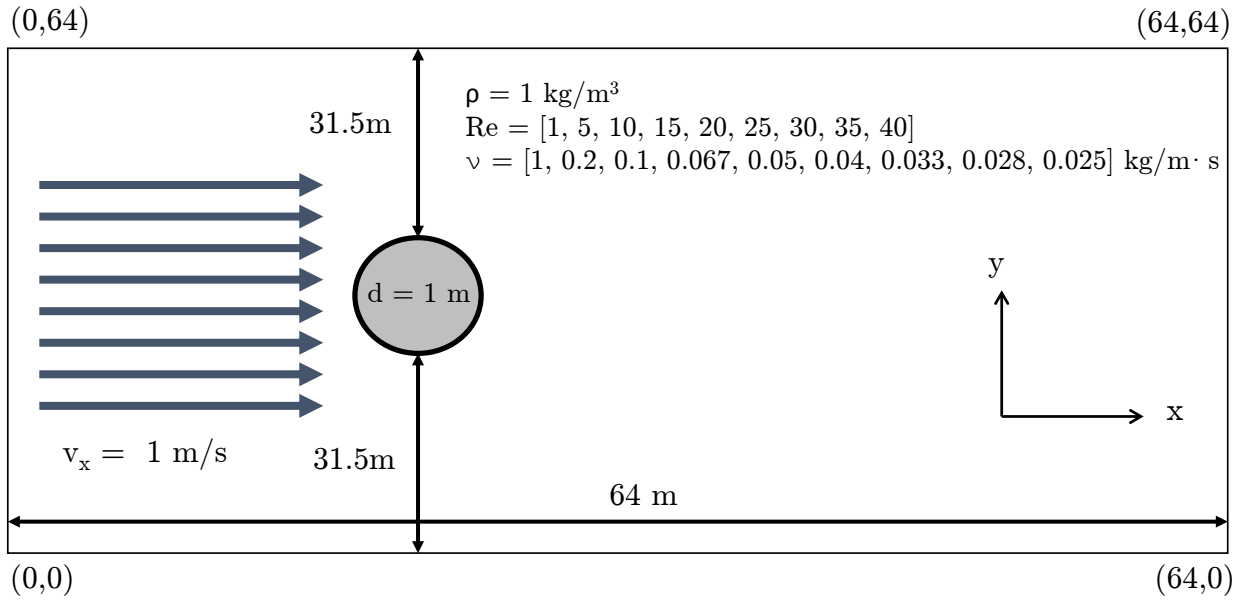



**Figure 4.12:** CFD Application: Steady Flow Past a Cylinder

## 3.2 Flat Plate Boundary Layer

This work studied two CFD applications, the second being flow past a flat plate shown in Figure 4.13. Fluid flowing past a flat plate creates a thin boundary layer as a result of the no-slip condition at the plates surface. At low Reynolds numbers, the boundary layer encompasses a relatively large area. This region of viscous effects becomes smaller with increasing Reynolds numbers, to the extent that at very high Reynolds numbers the boundary layer consists of only a thin layer near the surface. This boundary layer that forms on an semi-infinite plate is referred to as a Blasius boundary layer. The boundary layer thickness as a function of Reynolds number and distance

$$\delta = \begin{cases} \frac{5}{\sqrt{Re}}x & : Re < 5 \times 10^5 \\ \frac{0.38}{Re^{1/5}}x & : 5 \times 10^5 < Re < 10^7 \end{cases} \tag{4.6}$$

where $x$ is the distance along the primary axis. Similarly the drag force is found to be

$$C_D = \begin{cases} \frac{1.328}{\sqrt{Re}} & : Re < 5 \times 10^5 \\ \frac{0.074}{Re^{1/5}}x & : 5 \times 10^5 < Re < 10^7 \end{cases} \tag{4.7}$$

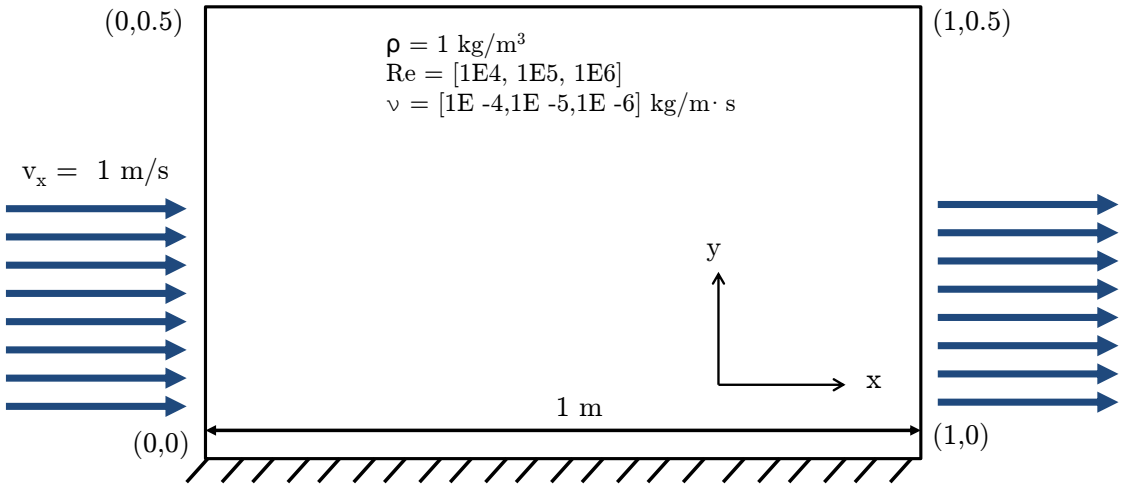



**Figure 4.13:** CFD Application: Flat Plate Boundary Layer

# CHAPTER 5

# LINEAR ALGEBRA RESULTS

This section outlines the numerical linear algebra results obtained using the three graphical processing unit setups. For each of the matrix operations (addition and multiplication), as well as the matrix applications (determinant, inverse, LU Decomposition) studied in this work, the basic outline of the algorithms is presented. This algorithm outline can be comparable to a pseudocode, and doesn't contain all the syntax in its entirety. Comments, denoted by //, are placed in the algorithm outline to present the large operations and help explain the process in a concise manner. Full syntax can be found in programming guides and manuals provided by NVIDIA, and full source code can be found on the authors website.

Acceleration plots are given for each operation and application as functions of matrix dimension. Plots are also given for the total execution time for each algorithm as a function of matrix dimension. Overall trends are analyzed for both plots, and key parameters such as maximum acceleration and the dimension size for which the GPU outperforms the CPU are mentioned. Comparisons are also made between the three GPU setups: K2000M, GTX760 and GTX760 SLI as they relate the overall execution time and acceleration factors compared to the CPU performance.

To determine the execution time of the algorithms, the CPU clock() function was used alongside with the GPU cudaEvent(). These clocks were started after the data was copied from the host to the device(s) and right become the algorithm began. These clocks were then ended after the algorithm was completed, from which the data was then copied back from the device(s) to the host. The acceleration factors were then determined by taking the ratio of the execution time of the CPU to that of the GPU for the same algorithm. For all algorithms, the matrices considered were chosen to be square such that the dimension represents the number of rows as well as the number of columns for a given matrix. Each algorithm was executed 10 times to get a time distribution.

## 0.3 Addition

Algorithm 1 outlines the GPU implemented matrix addition code for standard $m \times n$ matrices.

**Algorithm 1** Addition

```
1:  # include < stdio.h >
2:  # include < cuda.h >
3:  int n; // Number of Rows of Matrices
4:  int m; // Number of Columns of Matrices
5:  float *a, *b; // Matrix A and B on Host
6:  float *d_a, *d_b; // Matrix A and B on Device
7:  // Allocate Memory on Host and Device
8:  // Input Matrices from Text File
9:  // Copy Data from Host to Device
10: for (i = 1; i < n; i ++) do
11:     for (j = 1; j < m; j ++) do
12:         then
13:             C[i][j] = A[i][j] + B[i][j];
14:         end
15:         // Copy Data from Device to Host
16:     // Output Resultant Matrix C
17:         return C
18:     end for
```

Figure 5.1 displays the acceleration factor as a function of matrix dimension, and Figure 5.2 displays the overall execution time for Algorithm 1 as a function of matrix dimension. The GPUs began to outperform the CPU after the matrix dimension becomes greater than 80, from this point onward the acceleration factor from all three GPUs is greater than one. The GTX760 and

GTX760 SLI outperformed the K2000M, and the multiple GPU setup began to outperform the single GTX760 after the matrix dimension exceeded 350. The largest acceleration factors for the three GPU setups were 5.23x for the K2000M, 9.18x for the GTX 760, and 10.46x for the GTX760 SLI.

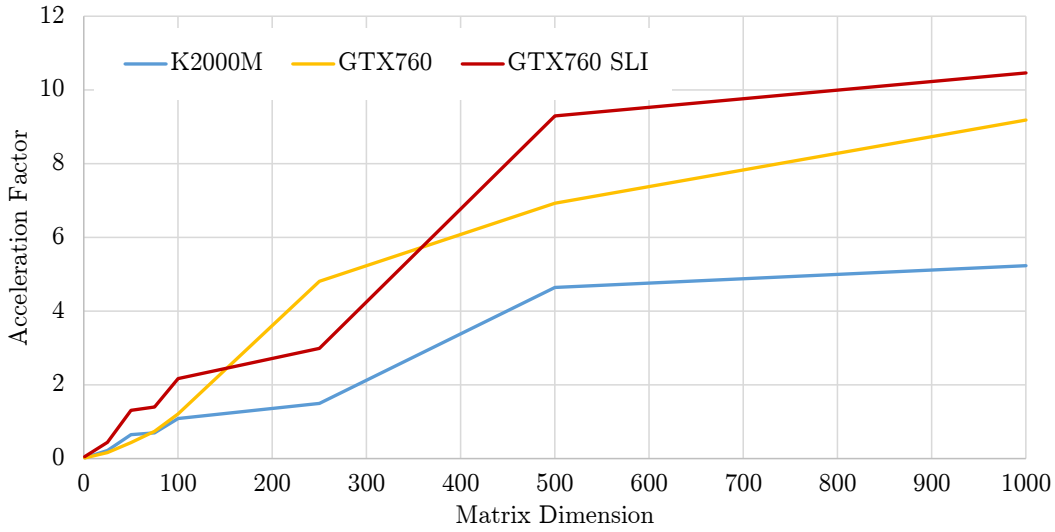



**Figure 5.1:** Addition: Acceleration Factor as a Function of Matrix Dimension

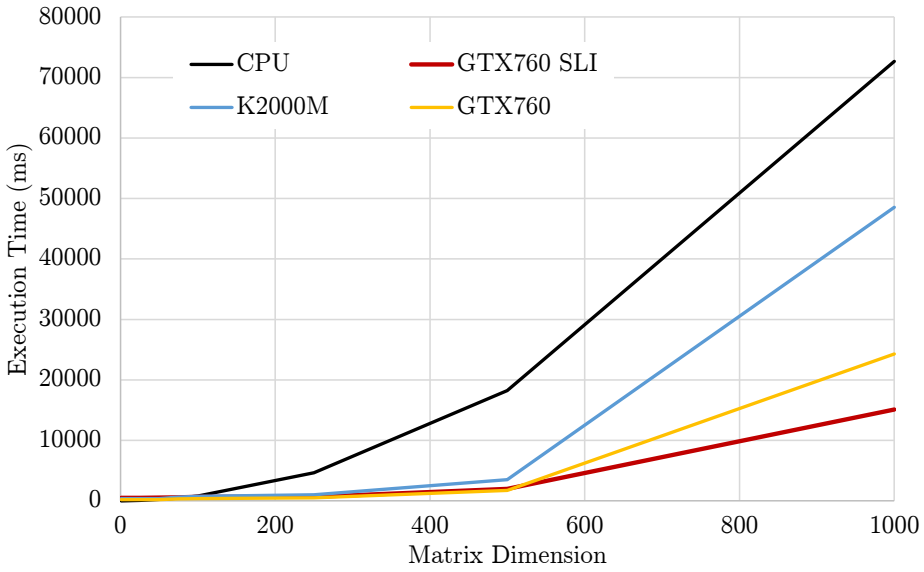



**Figure 5.2:** Addition: Execution Time as a Function of Matrix Dimension

## 0.4 Multiplication

Algorithm 2 outlines the GPU implemented matrix multiplication code for standard $m \times n$ matrices.

**Algorithm 2** Multiplcation

```
1:  # include < stdio.h >
2:  # include < cuda.h >
3:  int m; // Number of Rows of Matrix A
4:  int n; // Number of Columns of Matrix A/Number of Rows of Matrix B
5:  int p; // Number of Columns of Matrix B
6:  float *a, *b; // Matrix A and B on Host
7:  float *d_a, *d_b; // Matrix A and B on Device
8:  // Allocate Memory on Host and Device
9:  // Input Matrices from Text File
10: // Copy Data from Host to Device
11: for (k = 1; k < m; i ++) do
12:     for (i = 1; i < p; i ++) do
13:         then
14:             for (j = 1; j < n; j++) do
15:             F[k][i] = A[k][j] + B[j][i];
16:         end
17:         // Copy Data from Device to Host
18:     // Output Resultant Matrix F
19:         return F
20:     end for
```

Figure 5.3 displays the acceleration factor as a function of matrix dimension, and Figure 5.4 displays the overall execution time for Algorithm 2 as a function of matrix dimension. The GPUs

began to outperform the CPU after the matrix dimension becomes greater than 50, from this point onward the acceleration factor from all three GPUs is greater than one. The GTX760 SLI outperformed the other two GPU setups (K2000M and GTX760), and it becomes most evident when the size of the matrix was greater than 500. The largest acceleration factors for the three GPU setups were 5.04x for the K2000M, 5.60x for the GTX 760, and 10.09x for the GTX760 SLI.

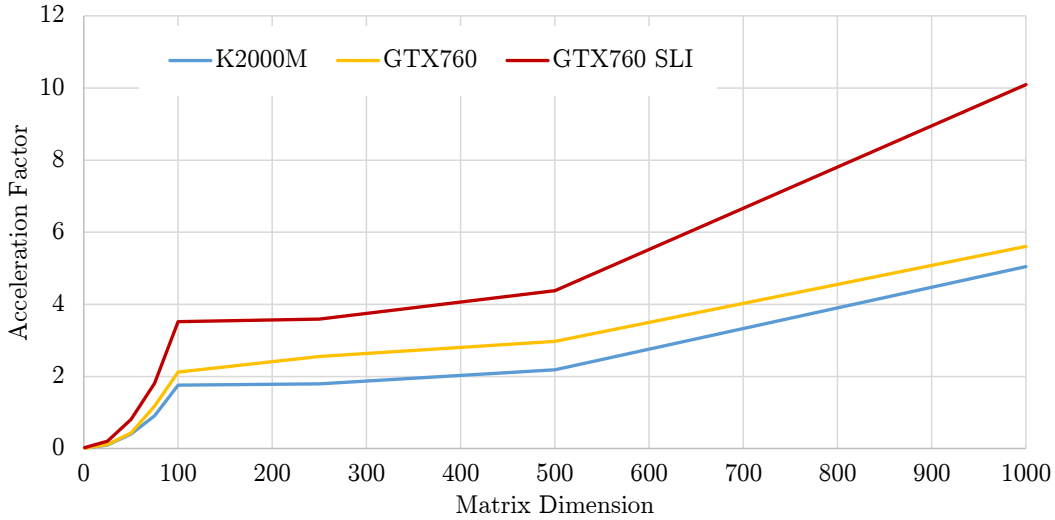



**Figure 5.3:** Multiplication: Acceleration Factor as a Function of Matrix Dimension

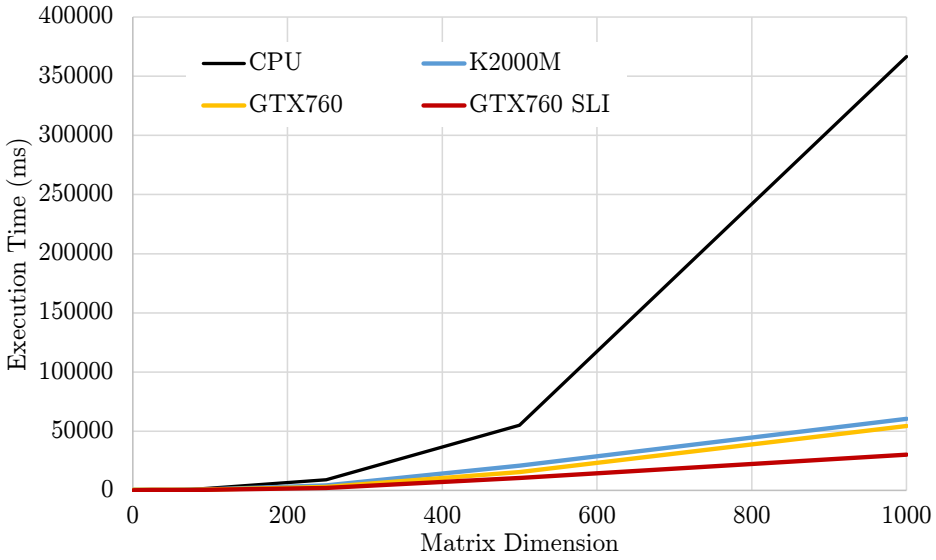



**Figure 5.4:** Multiplication: Execution Time as a Function of Matrix Dimension

## 0.5 Determinant

Algorithm 3 outlines the GPU implemented matrix determinant code for a square $n \times n$ matrix.

**Algorithm 3** Determinant

```
1:  # include < stdio.h >
2:  # include < cuda.h >
3:  int n; // Dimension of Matrix A
4:  float *a // Matrix A on Host
5:  float *d_a; // Matrix A on Device
6:  // Allocate Memory on Host and Device
7:  // Input Matrices from Text File
8:  // Copy Data from Host to Device
9:  for (i = 1; i < n; i ++) do
10:     for (j = 1; j < n; i ++) do
11:         then if (j > i)
12:             ratio = A[j][i]/A[i][i];
13:             for (k = 1; k < n; k++) do
14:             A[j][k] - = ratio*A[i][k];
15:         end
16:         // Copy Data from Device to Host
17:     // Output Determinant of Matrix A
18:         for (i = 1; i < n; i++) do
19:         return det * = A[i][j]
20:     end for
```

This algorithm was developed to test if the matrices being analyzed were invertible. Recalling from the Background section of this work, if det($A$) is nonzero, the matrix $A$ possesses an inverse.

## 0.6 Inverse

Algorithm 4 outlines the GPU implemented matrix inversion code for a square $n \times n$ matrix.

**Algorithm 4** Inverse

```
1: # include < stdio.h >, < cuda.h >
2: int n; // Dimension of Matrix A
3: float *a, *d_a // Matrix A on Host and Device
4: for (k = 1; k < n; k ++) do // Copy Data from Host to Device
5:     L[k][k] = 1;
6:     for (i = k+1; i < n; i ++) do
7:         L[i][k] = A[i][k] / A[k][k];
8:         for (j = k+1; j < n; j++) do
9:         A[i][j] - = L[i][k]*A[k][j];
10:        for (j = k; j < n; j++) do
11:        U[k][j] = A[k][j]
12:        for (o = 1; o < n; o++) do // Forward Substitution
13:        y[o] = b[o]
14:        for (p = 1; p < i+1; p++) do
15:        y[o] + = -l[o][o]*y[j]
16:        for (q = n; q < 1; q–) do // Backward Substitution
17:        x[q] = y[q]
18:        for (r = q+1; r < n; r++) do
19:            x[q] + = -u[q][r]*x[r]
20:        x[q] = x[q]/u[q][q]
21:    U[k][j] = A[k][j] // Copy Data from Device to Host and Output Inverse Matrix
22:    end for
```

Figure 5.5 displays the acceleration factor as a function of matrix dimension and Figure 5.6 displays the overall execution time for Algorithm 4 as a function of matrix dimension. The performance for the matrix inversion algorithm was the highest for the GTX760 SLI at 2.68x faster, with the GTX760 peaking at 2.01x and the K2000M at 1.27x. Being a combination of the two matrix operations and two matrix applications, the matrix inversion algorithm was expected to deliver smaller speed-up factors and take overall more execution time.

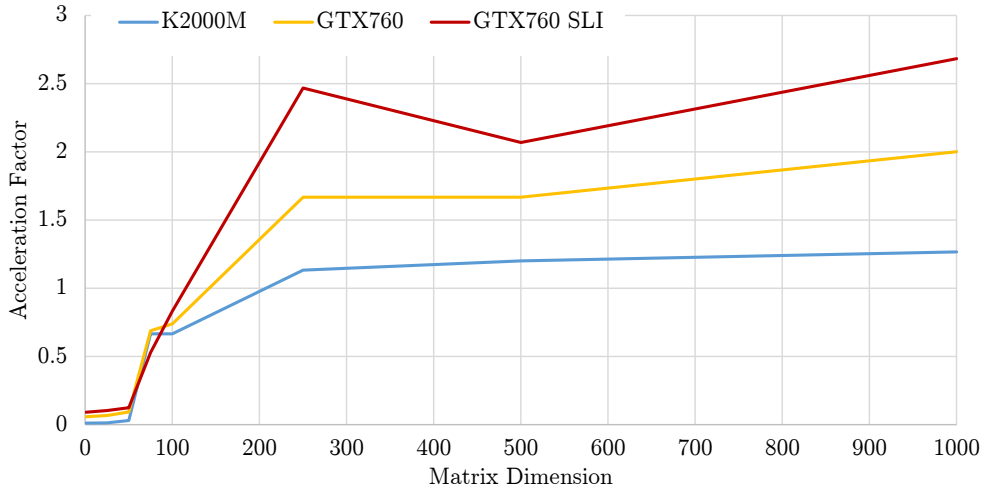



**Figure 5.5:** Inverse: Acceleration Factor as a Function of Matrix Dimension

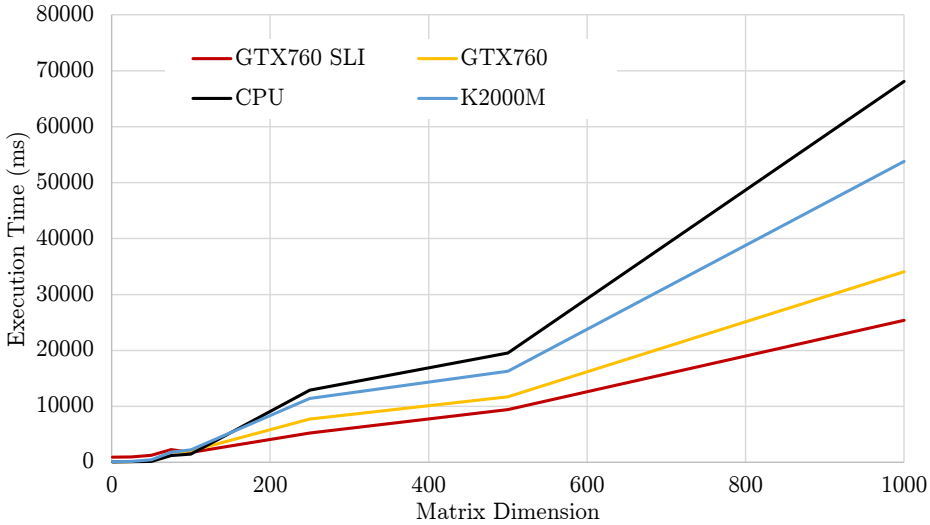



**Figure 5.6:** Inverse: Execution Time as a Function of Matrix Dimension

## 0.7 LU Decomposition

Algorithm 5 outlines the GPU implemented LU Decomposition code for a square $n \times n$ matrix.

**Algorithm 5** LU Decomposition

```
1: # include < stdio.h >
2: # include < cuda.h >
3: int n; // Dimension of Matrix A
4: float *a // Matrix A on Host
5: float *d_a; // Matrix A on Device
6: // Allocate Memory on Host and Device
7: // Input Matrices from Text File
8: // Copy Data from Host to Device
9: for (k = 1; k < n; k ++) do
10:     L[k][k] = 1;
11:     for (i = k+1; i < n; i ++) do
12:         L[i][k] = A[i][k] / A[k][k];
13:         for (j = k+1; j < n; j++) do
14:         A[i][j] - = L[i][k]*A[k][j];
15:         // Copy Data from Device to Host
16:     // Output Matrix L and U
17:         for (j = k; j < n; j++) do
18:         return U[k][j] = A[k][j]
19:     end for
```

Algorithm 5 was used for the matrix determinant and inverse algorithms. Performing LU Decomposition instead of cofactor expansion for determinants and using the fact that $\det(A) = \det(L)\det(U)$, the order of magnitude for performing a determinant operation is reduced.

# CHAPTER 6

# CFD RESULTS

## 1. Flow Around a Cylinder

Figures 6.1 through 6.5 display the stream lines on the top, velocity vectors on the left and vorticity on the right for Reynolds numbers ranging from Re = 1 to Re = 50. These results were obtained in FLUENT, with the parameters and geometry as described previously in Section 3.1. For small Reynolds numbers, the streamline is symmetrical about the cylinder's upper and lower side as well as its front to rear evidenced in Figure 6.1. The vorticity is more variant than the velocity vectors, as evidenced the change in output from a Reynolds number of 10 to 20 in Figure 6.2. Increasing Reynolds number produces a loss in the front to rear symmetry, and a closed streamline region generates at the rear as evidenced in Figure 6.3 and 6.4. These regions are known as vortices in the flow, and increase in length as the Reynolds number increases. Reynolds number was kept under 50 to maintain steady-state conditions and avoid turbulent flow.

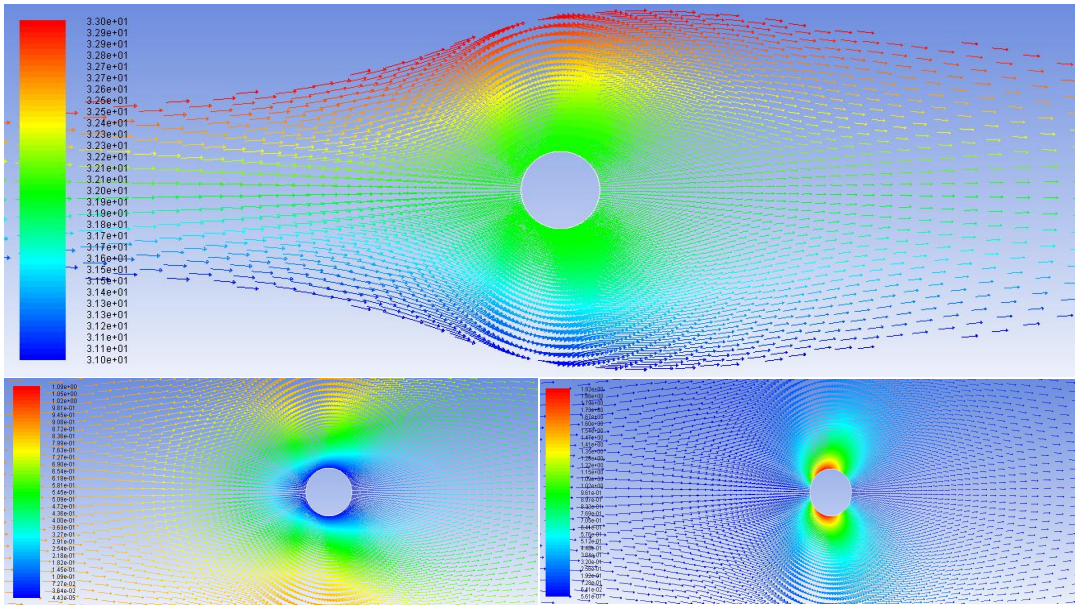


**Figure 6.1:** Flow Around Cylinder: Re = 1

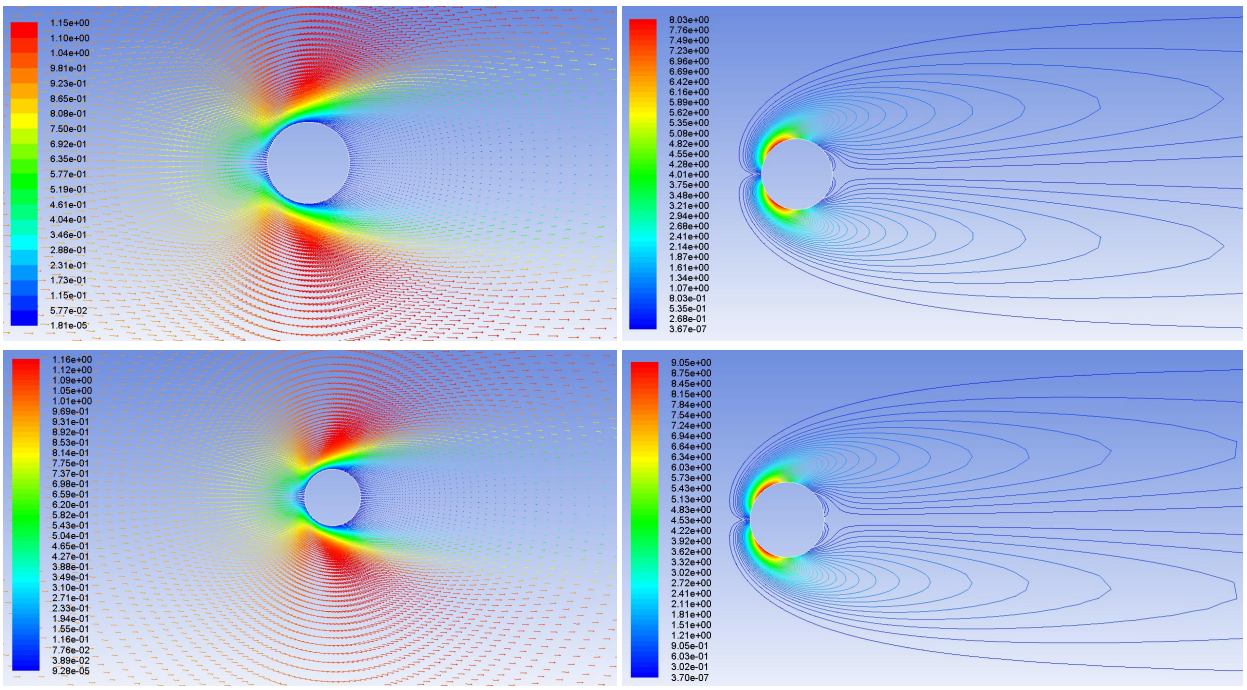


**Figure 6.2:** Flow Around Cylinder: Re = 10, 20

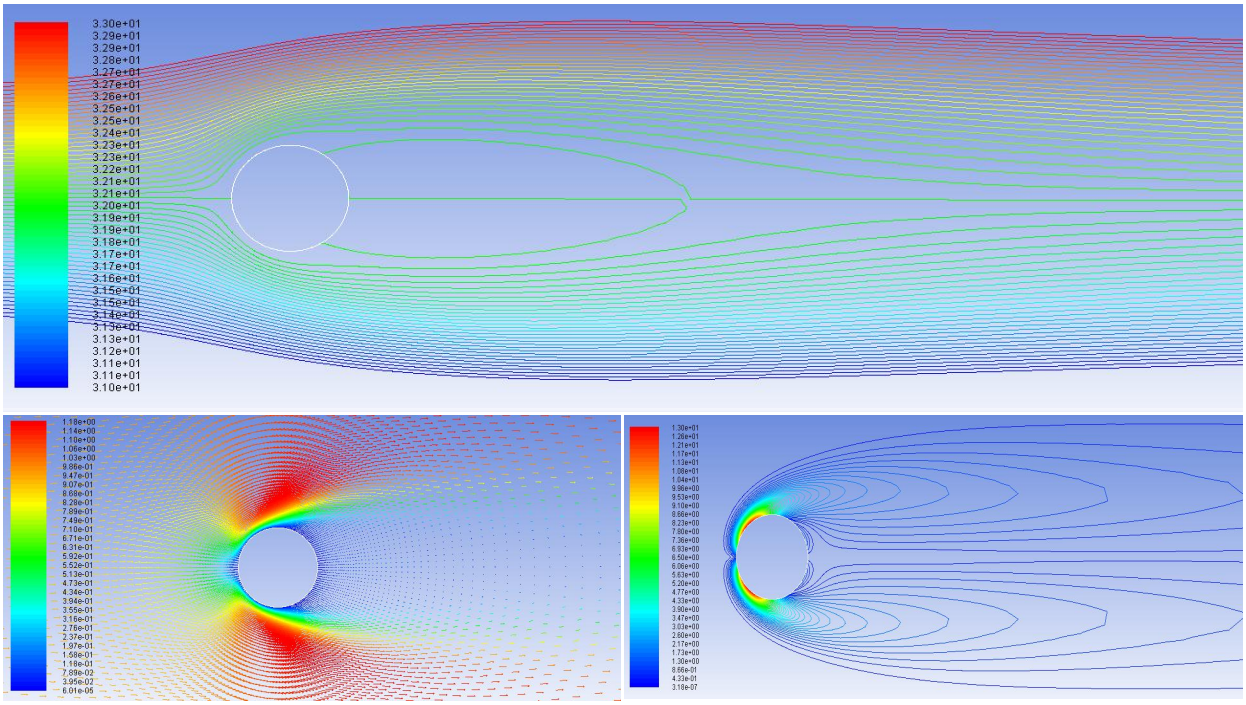


**Figure 6.3:** Flow Around Cylinder: Re = 25

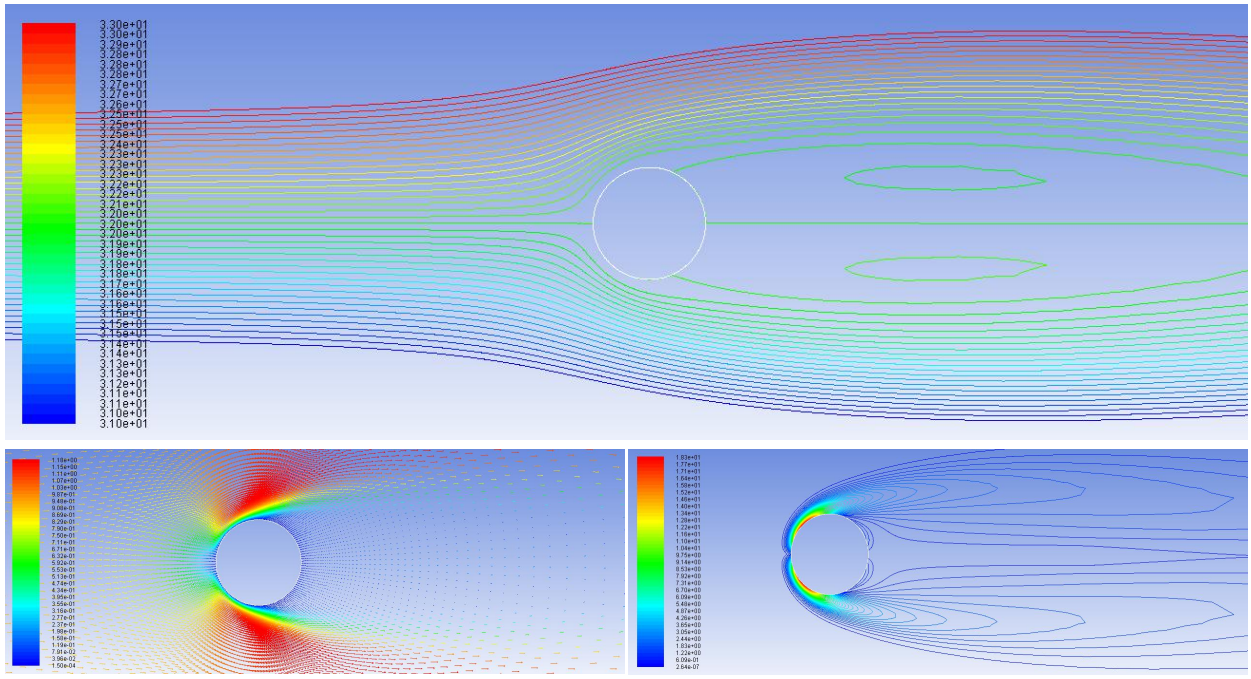


**Figure 6.4:** Flow Around Cylinder: Re = 40

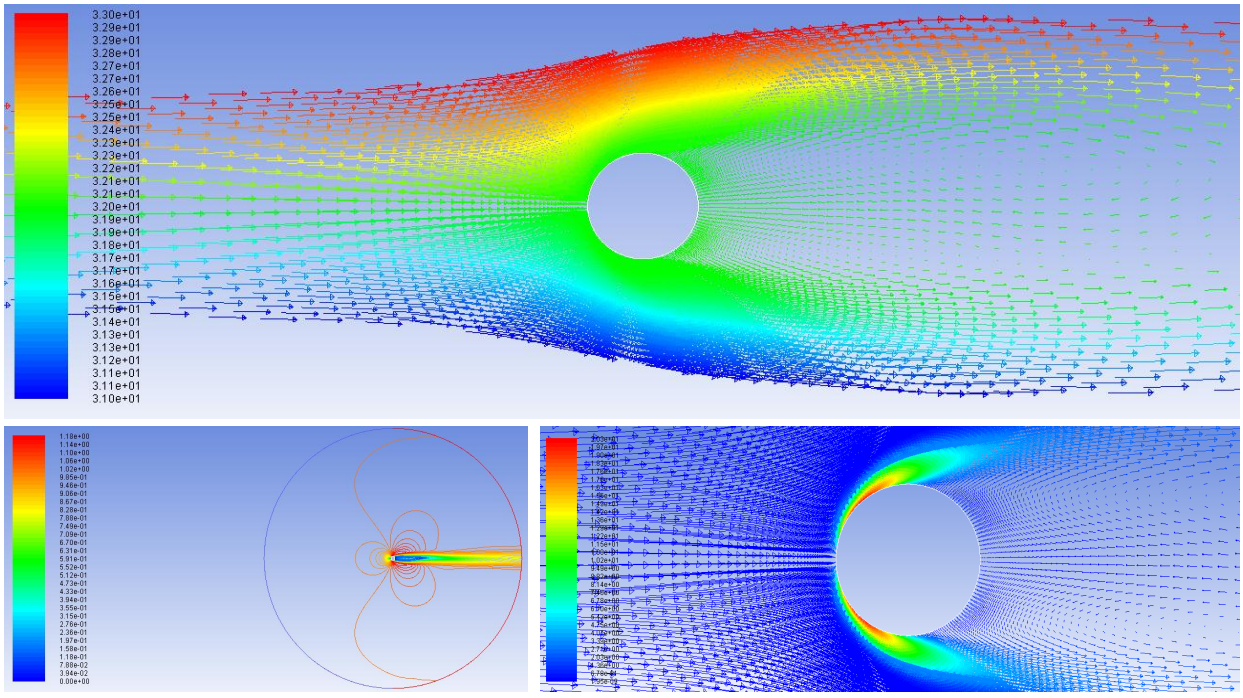


**Figure 6.5:** Flow Around Cylinder: Re = 50

Table 6.1 displays the drag coefficients along with the drag forces of the cylinder for the selected Reynolds numbers.

**Table 6.1:** Flow Around a Cylinder Variables

| **Reynolds Number,** $Re$ | **Drag Coefficient,** $C_D$ | **Drag Force,** $F_D$ |
|---|---|---|
| 1 | 2.042 | 0.864 N |
| 5 | 1.853 | 0.614 N |
| 10 | 1.728 | 0.572 N |
| 15 | 1.399 | 0.508 N |
| 20 | 1.015 | 0.470 N |
| 25 | 1.008 | 0.379 N |
| 30 | 0.887 | 0.369 N |
| 35 | 0.855 | 0.360 N |
| 40 | 0.810 | 0.325 N |
| 45 | 0.803 | 0.319 N |
| 50 | 0.793 | 0.285 N |

The drag coefficient can be found analytically from

$$C_D = \frac{2F_D}{\rho v_x^2 A_r}, \tag{6.1}$$

where $F_D$ is the component of the drag force acting on the cylinders surface, and $A_r$ is the reference area that is the area projected onto the cylinders surface perpendicular to the fluid flow. For small Reynolds numbers the Oseen approximation is applicable as

$$C_D = \frac{8\pi}{Re[2.002 - \ln(Re)]}. \tag{6.2}$$

Figure 6.6 displays the Lattice-Boltzmann coded flow profile as a function of time for selected Reynolds numbers in Table 6.1. Source code applying the LBM method with the BGK approximation was obtained and parallelized with MATLAB's Parallel Computing Toolbox seeing acceleration factors up to 1.9x [78]. The acceleration factors were fairly consistent across Reynolds numbers, seeing its highest rate at the highest Reynold number tested, Re = 50. Drag forces computed appear within a tolerance range of 10% for the values found in Table 6.1 of the Reynolds numbers displayed in Figure 6.6.

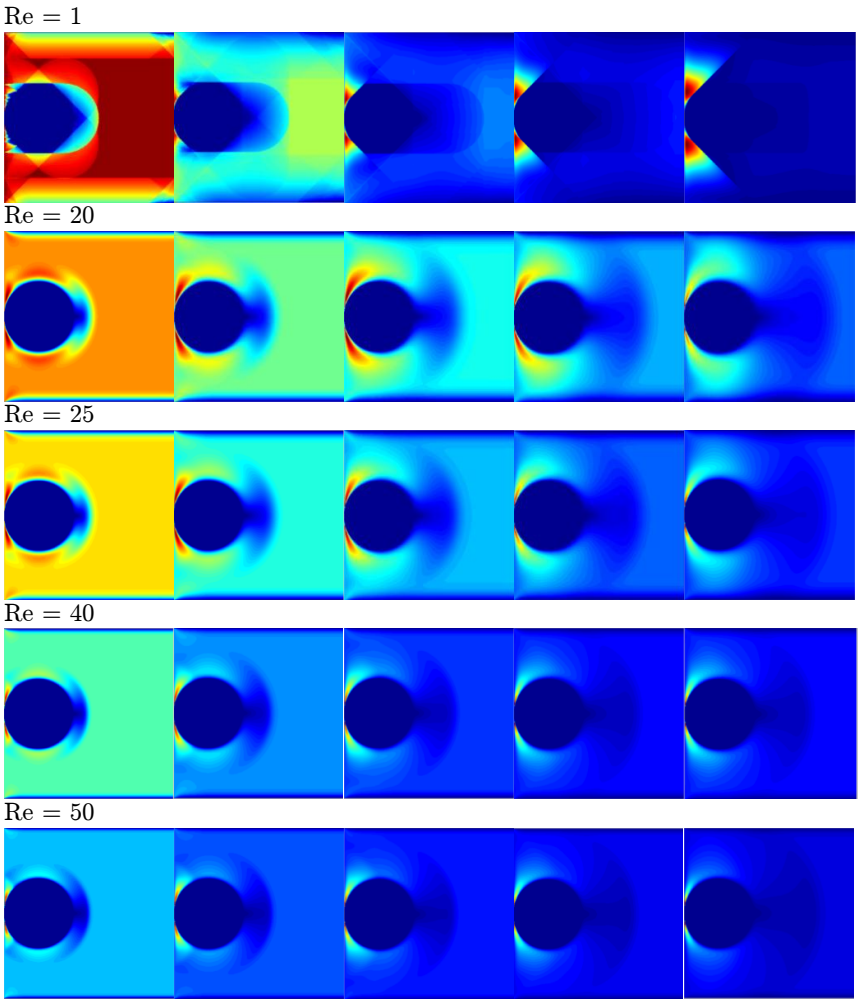



**Figure 6.6:** Flow Profile at t = 10, 20, 30, 40, 50 seconds

## 2. Flat Plate Boundary Layer

Figures 6.7 through 6.11 display the velocity profiles on the top and pressure profiles on the bottom for Reynolds numbers ranging from Re = 1 to Re = 10000. These results were obtained in FLUENT, with the parameters and geometry as described previously in Section 3.2. For small Reynolds numbers, the size of the boundary layer is much larger and is evidenced by the lightly colored yellow arrows seen in Figure 6.7. Increasing Reynolds numbers produce smaller boundary layers evidenced by smaller lightly colored yellow arrows and the increasing darker colored red arrows. As a result of the boundary layer becoming thinner, the pressure decreases as a function of the direction of the flow evidenced for all Reynolds numbers. Reynolds numbers were chosen such that flow separation, the phenomenon of the velocity at the plate becoming an inflection point, did not occur.

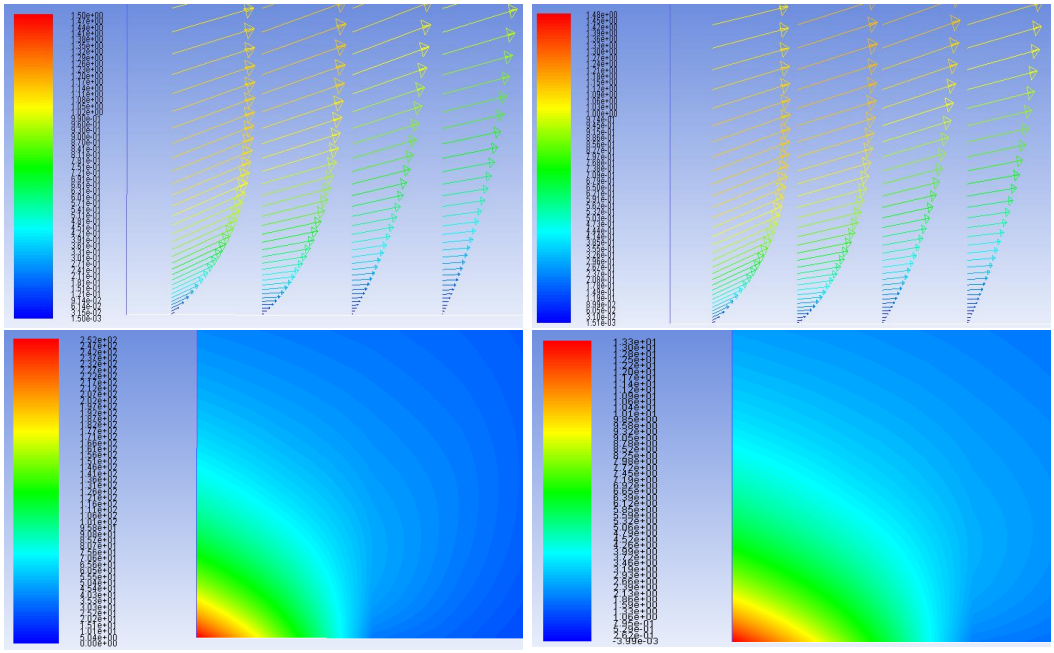


**Figure 6.7:** Velocity Profile (top) and Pressure Distribution (bottom): Re = 1 (left) and Re = 20 (right)

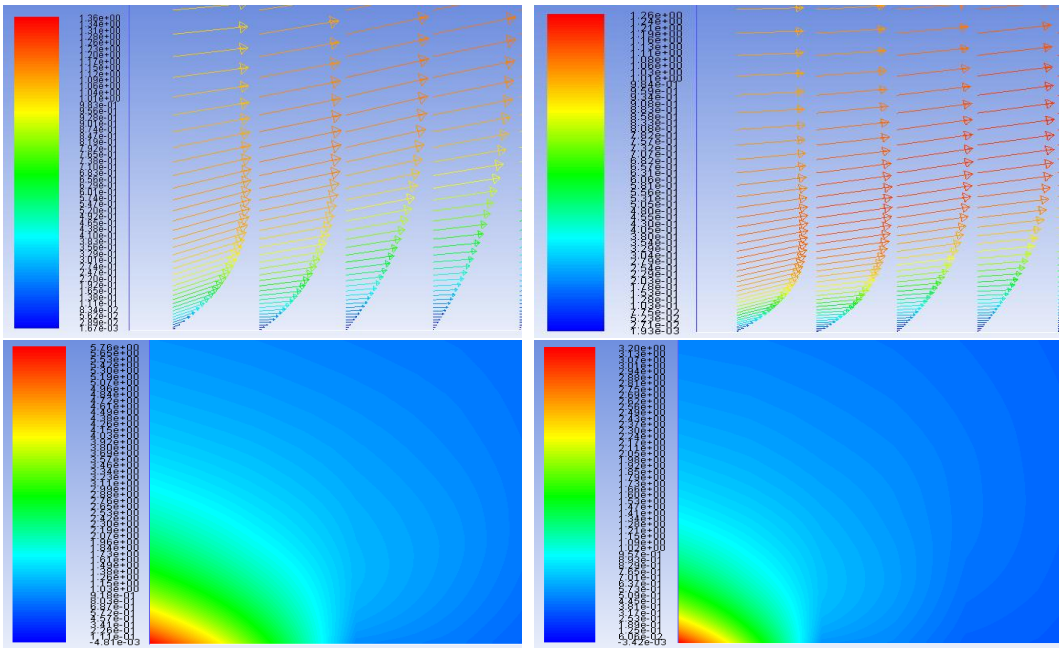


**Figure 6.8:** Velocity Profile (top) and Pressure Distribution (bottom): Re = 50 (left) and Re = 100 (right)

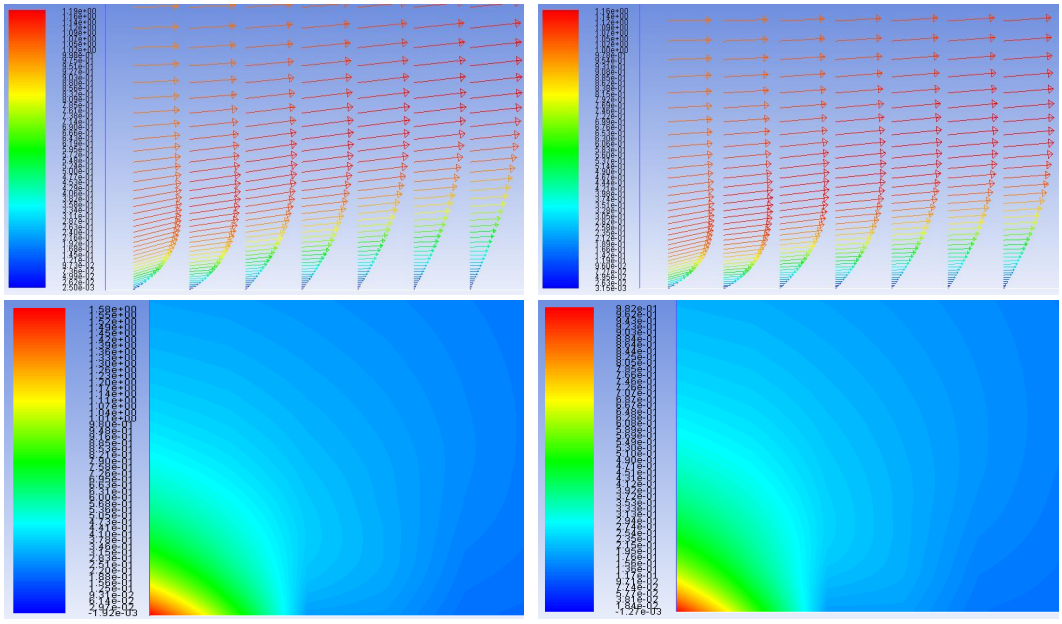


**Figure 6.9:** Velocity Profile (top) and Pressure Distribution (bottom): Re = 250 (left) and Re = 500 (right)

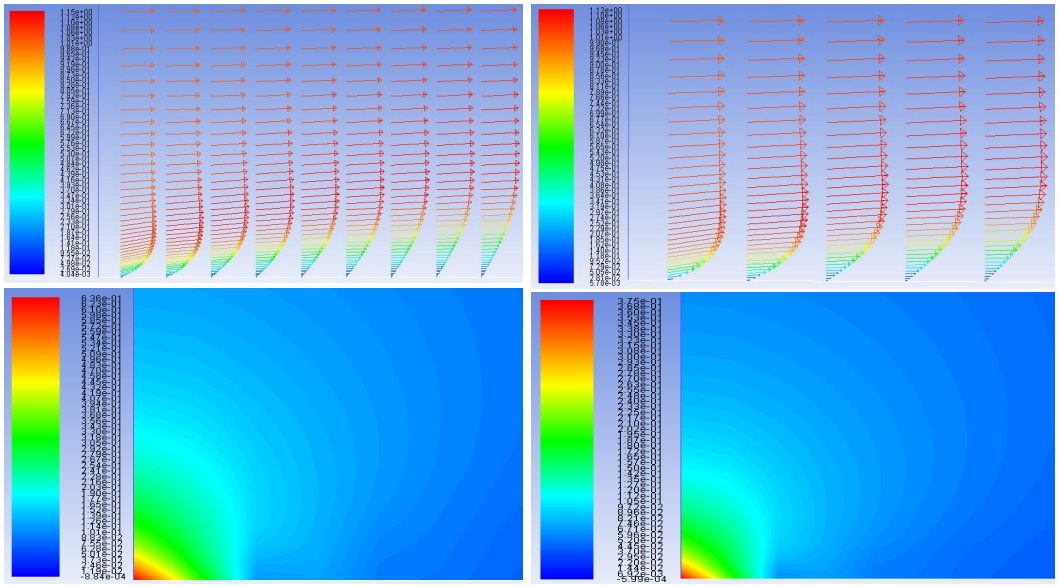


**Figure 6.10:** Velocity Profile (top) and Pressure Distribution (bottom): Re = 1000 (left) and Re = 2500 (right)

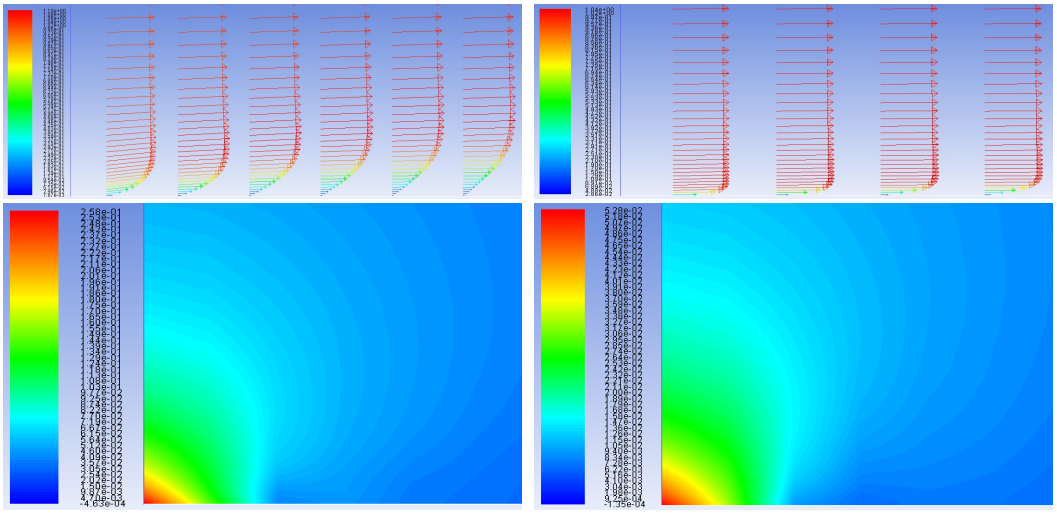


**Figure 6.11:** Velocity Profile (top) and Pressure Distribution (bottom): Re = 5000 (left) and Re = 10000 (right)

# CHAPTER 7

# CONCLUSIONS

## 1. Concluding Statements

In summary, the research objectives outlines previously are overviewed and extensively reviewed.

**• Develop stable and efficient algorithms using CUDA for NVIDIA GPUs**

To determine if this condition was satisfied, it is imperative that first *stable* and *efficient* are first defined. Stable algorithms do not propagate error and execute with desired and expected results. Efficient algorithms execute in minimal runtime, and optimize memory and computing resources. Small test matrices and known geometric setups ensured the validity of the results obtained for the numerical linear algebra and CFD results, respectively. Increasing the size of the matrices and physical fluid parameters did not result in the propagation of error, and the algorithms were determined to be stable. Likewise, best practices laid out by NVIDIA and others conducting research were followed to optimize the algorithms for efficiency.

**• Optimize linear algebra algorithms for performance**

The algorithms developed in the Results section of this thesis were analyzed for limiting regions and bottlenecks. Using NVIDIA Visual Profiler, the number of kernels along with the block and grid sizes were chosen to maximize the acceleration factors of the algorithms. Proper load assignment to each kernel (and each GPU in the case of the SLI setup) is also another area of consideration to maximize algorithm performance. Coalescing computations, utilizing available shared GPU memory, zero copy, maximizing bandwidth and minimizing host/device data transfers are other considerations related to GPU computations that also play a role optimizing the linear algebra algorithms.

• **Develop parallel CFD algorithms**

From the two applications analyzed, steady flow around a cylinder with a Lattice-Boltzmann method and Blasius flat plate boundary flow with a Navier-Stokes method, parallel CFD algorithms were developed. The results obtained from these algorithms match consistently with the FLUENT models, both visually and numerically. Implementation through MATLAB's Parallel Computing Toolbox ran smoothly for the three GPU setups, and acceleration results were promising.

• **Implement single vs. multiple GPU approach**

With multiple companies (NVIDIA, AMD) possessing programmable graphics cards for scientific computing purposes, it was critical to select the best company that provided the most computing power, resources and guides, and promise for future expansion. After selecting NVIDIA as the primary option, the next step was to determine which family architecture (GeForce, Quadro, Tesla) to choose. Seeing the opportunity to compare multiple families, a Quadro K2000M and GeForce GTX 760 were chosen for their capabilities and price. The GeForce GTX 760 also was attractive because it offered the ability for SLI, enabling a further comparison between a single and dual GPU setup. Prior research has been conducted on multiple GPU setups for Tesla configurations, but to the authors knowledge this appears to be the first to implement a GeForce GTX 760 in this similar fashion.

• **Introduce GPU computing to the general audience**

GPU computing is a field of great promise with increasing consumer desire for accelerated computational performance and advancing complex algorithms. While this work outlined just two applications that benefit from parallel computing with GPUs, there are many other fields that could seek immediate computational gains. Within the next year, the primary goal is to open workshops and seminars to disseminate knowledge about GPU computing at UCF and create open access tutorial videos and example files available on the authors personal website.

## 2. Recommendations for Future Work

**• Parallelize additional linear algebra and CFD applications**

While the applications covered in this work were some of the most common, it doesn't encompass all possible applications that are useful in numerical linear algebra and CFD. Additional linear algebra applications are eigenvalue problems, Chlokesky decomposition and QR factorization. Additional CFD applications are cavity flow, pipe flow and flow through a nozzle.

**• Study memory instruction classification**

Memory instruction classification deals with the different types of GPU memory: shared, global, texture, constant. The use of shared memory was maximized in this work based on the suggestions from the NVIDIA programming guide and past research. Global memory can also be used, and future work can look to determine the proper balance between the amount of data sent to global memory and shared memory.

**• Comprehensive power and energy study**

Power consumption and energy measures are important measures for GPUs and computers in general. Studies into the power consumed as a function of performance (FLOP/s and acceleration factor) as well as power consumed as a function of the number of threads are important relationships studied in prior works. NVIDIA's newest architecture, Maxwell was released with emphasis placed on its efficiency in terms of power consumption and energy. Jen-Hsun Huang, the CEO of NVIDIA during a Q&A session at a investor day conference in 2013 expressed that Maxwell-based GPUs will give improved graphics capabilities, simplified programmability, and increase its efficiency over Kepler-based GPUs [124].

# APPENDIX A

# EXAMPLE MATRICES

$$A^* = \begin{bmatrix} 1 & -2 & 0 \\ 2 & -6 & 3 \\ 0 & 1 & 1 \end{bmatrix}, \; |A^*| = 3 \times 3 \tag{A.1}$$

$$B^* = \begin{bmatrix} 3 & 0 & 2 \\ 1 & -4 & 3 \\ 2 & -8 & 6 \end{bmatrix}, \; |B^*| = 3 \times 3 \tag{A.2}$$

$$C^* = \begin{bmatrix} 0 & 2 & 1 \\ 3 & -1 & 2 \\ 4 & 0 & 1 \end{bmatrix}, \; |C^*| = 3 \times 3 \tag{A.3}$$

$$E^* = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 0 & -1 \\ -6 & 2 & 3 \end{bmatrix}, \; |E^*| = 3 \times 3 \tag{A.4}$$

$$F^* = \begin{bmatrix} 1 & -2 & 3 \\ -1 & 3 & 0 \\ 2 & -5 & 5 \end{bmatrix}, \; |F^*| = 3 \times 3 \quad \& \quad b^* = \begin{bmatrix} 9 \\ -4 \\ 17 \end{bmatrix}, \; |b^*| = 3 \times 1 \tag{A.5}$$

$$G^* = \begin{bmatrix} 1 & -2 & -1 \\ 2 & 8 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \; |G^*| = 3 \times 3 \tag{A.6}$$

# REFERENCES


[1] S. Abbasbandy, R. Ezzati, and A. Jafarian. "LU Decomposition Method for Solving Fuzzy System of Linear Equations", *Applied Mathematics and Computation*, 172(1): 633-643, Jan. 2006.

[2] R. Alvarez, F. Martinez, J.-F. Vicent, and A. Zamora. "Cryptographic Applications of 3x3 Block Upper Triangular Matrices", *Hybrid Artificial Intelligence*, 7209: 97-104, 2012.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hardi, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects", *Journal of Physics: Conference Series*, 180(1): 1-5, 2009.

[4] G.B. Arfken, H.-J. Weber, and L. Ruby. *Mathematical Methods for Physicists*. Academic Press, 1985.

[5] M. Baboulin, J. Dongarra, and S. Tomov. "Some Issues in Dense Linear Algebra for Multi-core and Special Purpose Architectures", *Centro de Matemtica da Universidade de Coimbra*, 1-12, 2008.

[6] J. Barbosa, J. Tavares, and A.J. Padiha. "Optimizing Dense Linear Algebra Algorithms on Heterogeneous Machines", *Algorithms and Tools for Parallel Computing on Heterogeneous Clusters*, 17-31, 2006.

[7] S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Orti. "Solving Dense Linear Systems on Graphics Processors", *Euro-Par 2008Parallel Processing*, Springer, 739-748, 2008.

[8] M. Barrauly, Y. Maday, N. Nguyen, and A. Patera. "An Empirical Interpolation Method: Application to Efficient Reduced-Basis Discretization of Partial Differential Equations", *Comptes Rendus Mathematique*, 339(9): 667-672, 2004.

[9] G. Beliakov, and Y. Matiyasevich. "A Parallel Algorithm for Calculation of Large Determinants with High Accuracy for GPUs and MPI Clusters", *arXiv*, 1-17, 2013.

[10] N. Bell. *Sparse Matrix Representations and Iterative Solvers: Lesson 1*. NVIDIA, 2008.

[11] N. Bell, and M. Garland. "Efficient Sparse Matrix-Vector Multiplication on CUDA", *NVIDIA Technical Report NVR-2008-004*, 1-32, Dec. 11, 2008.

[12] N. Bell, and M. Garland. "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors", *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009.

[13] P. Benacerraf, and H. Putnam. *Philosophy of Mathematics: Selected Readings*. Cambridge University Press, 1983.

[14] P. Benner, P. Ezzatti, E. Quintana-Orti, and A. Remon. "Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function", *Euro-Par 2009Parallel Processing Workshops*. Springer, 132-139, 2010.

[15] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, and E. Kaxiras. "A Flexible HighPerformance Lattice Boltzmann GPU code for the Simulations of Fluid Flows in Complex Geometries", *Concurrency and Computation: Practice and Experience* 22(1): 1-14, 2010.

[16] M. Berry, S. Dumais, and G. O'Brien. "Using Linear Algebra for Intelligent Information Retrieval", *SIAM Review* 37(4), 573-595, 1995.

[17] E. Anderson. *LAPACK Users' guide*. SIAM, Vol. 9, 1999.

[18] J. Boltz, I. Farmer, E. Grinspun, and P. Schroder. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid", *ACM Transactions on Graphics*, 22(3): 917-924, 2003.

[19] G. Bosilca, A. Bouteiler, T. Herault, P. Lemarinier, N. Saengpatsa, S. Tomov, and J. Dongarra. "Performance Portability of a GPU Enabled Factorization with the DAGuE Framework", *2011 IEEE International Conference on Cluster Computing*, 395-402, 2011.

[20] C.B. Boyer, and U.C. Merzbach. *A History of Mathematics*. John Wiley & Sons, 1976.

[21] O. Bretscher. *Linear Algebra with Applications*. Prentice Hall, 1997.

[22] R. Brown, "Bottlenecks", 05-24-2004.

[23] I. Buck. GPU Computing with NVIDIA CUDA. *SIGGRAPH 2007*, 2007.

[24] D. Burton. *The History of Mathematics.* McGraw-Hill, 1991.

[25] J. Carthy, "History of Computers", Comp 1001: History of Computers at UCD School of Computer Science at Informatics.

[26] D. Castano-Diez, D. Moser, A. Schoenegger, S. Pruggnaller, and A. Frangakis. "Performance Evaluation of Image Processing Algorithms on the GPU", *Journal of Structural Biology*, 164(1): 153-160, 2008.

[27] Y. Censor. "Parallel application of block-iterative methods in medical imaging and radiation therapy", *Mathematical Programming* 42(1-3): 307-325, Apr. 1988.

[28] Center for Academic Support. *Elementary Row Operations for Matrices*. Missouri Western State University. 25 May 2008.

[29] H. Chang. The Hidden History of Phlogiston. *HYLEInternational Journal for Philosophy of Chemistry* 16(2): 47-79, 2010.

[30] E.W. Cheney, and D.R. Kincaid. *Numerical Mathematics and Computing*. Cengage Learning, 2012.

[31] J.W. Choi, A. Singh, and R.W. Vudoc. " Model-driven autotuning of sparse matrix-vector multiply on GPUs", *ACM Sigplan Notices*. 45(5), 2010.

[32] C.T. Chong. *Some Remarks on the History of Linear Algebra*. National University of Singapore.

[33] J.M. Cohen, and M.J. Molemaker. "A Fast Double Precision CFD Code using CUDA", *Proceedings of Parallel CFD 2009*. 414-429, 2009.

[34] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.* Newnes, 2012.

[35] M. Coutanceau, and J.-R. Defaye. "Circular cylinder wake configurations: A flow visualization survey." *Applied Mechanics Reviews* 44.6, 255-305, 1991.

[36] J.D. Cresser. *Lecture Notes on Special Relativity*. Department of Physics. Macquarie University. 8 August 2005.

[37] EM Photonics, Inc. *CULA Programmers Guide*, 2009-2013.

[38] A.W. Date. *Introduction to Computational Fluid Dynamics*. Cambridge University Press, 2005.

[39] A. Davidson, and J. Hall. "Using a Graphics Processor Unit (GPU) for Feature Extraction from Turbulent Flow Datasets. *20th National Conference on Undergraduate Research*. Volume 99, 2006.

[40] T.A. Davis, and Y. Hu. "The University of Florida Sparse Matrix Collection", *ACM Transactions on Mathematical Software*. 38(1), 2011.

[41] A. DePalma. "Beating The Heat", from ASME.org, February 2012.

[42] A. Dolicho. "GeForce SLI Technology: An Introductory Guide", *GeForce: Guides*, Web.

[43] T. Doom, M.L. Raymer, D. Krane, and O. Garcia. "Crossing the Interdisciplinary Barrier: a Baccalaureate Computer Science Option in Bioinformatics", *IEEE Transactions on Education*, 46(3): 387,393, Aug. 2003.

[44] S. Du, C. Tu, and M. Sun. "High Accuracy Hough Transform Based on Butterfly Symmetry", *Electronics Letters* 48(4): 199-201, 2012.

[45] S. Easterbrook. *What is Engineering?: Lecture 3*. University of Toronto. Department of Computer Science. 2004-5.

[46] L. Euler, J. Hewlett, F. Horner, J. Bernoulii, and J.L. Lagrange. *Elements of Algebra*. Longman, Orme and Co., 1840.

[47] EXA Corporation. *Frequently Asked Questions Physics*. 2007.

[48] P. Ezzatti, E.S. Quintana-Orti, and A. Remon. "Using graphics processors to accelerate the computation of the matrix inverse", *The Journal of Supercomputing*, 58(3): 429-437, 2011.

[49] J.D. Faires, and R. Burden. *Numerical Methods*. Brooks Cole, 1998.

[50] R. Farber. *CUDA application design and development*. Elsevier, 2011.

[51] G. Fasshauer. *Gaussian Elimination and LU Factorization*. Illinois Institute of Technology. Numerical Linear Algebra. Fall 2006.

[52] I. Friedberg, A.J. Insel, and L.E. Spence.*Linear Algebra*. Prentice Hall, 2003.

[53] G.P. Galdi. *An Introduction to the Mathematical Theory of the Navier-Stokes Equations: Linearised Steady Problems*. Springer, 1994.

[54] N. Galoppo, N.K. Govindaraju, M. Henson, and D. Manocha."LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware", *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[55] J.R. Gilbert, C. Moler, and R. Schreiber. "Sparse Matrices in MATLAB: Design and Implementation", *SIAM Journal on Matrix Analysis and Applications*, 13(1): 333-356, 1992.

[56] D. Goddeke, S.H.M. Buijssen, H. Wobke, and S. Turek."GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver", *High Performance Computing & Simulation*, 2009.

[57] G. Golub, and C.F. Van Loan. *Matrix Computations(Johns Hopkins Studies in Mathematical Sciences)*. John Hopkins University Press, 1996.

[58] W.D. Gropp, D.K. Kaushik, D.E. Keyes, and B.F. Smith. "High-Performance Parallel Implicit CFD", *Parallel Computing*, 27(4): 337-362, 2001.

[59] F.B. Hanson. *CAUTION: Cramser's Rule is Computationally Expensive: Special Notes from MSC 471*. University of Illinois at Chicago. Fall 2004.

[60] S.A. Haque, and M.M. Maza. "Determinant Computation on the GPU using the Condensation Method", *Journal of Physics: Conference Series.* 341(1): 1-10, 2012.

[61] W. D. Hart. *The Philosophy of Mathematics*. Oxford University Press, 1996.

[62] X. He, and L.-S. Lou. "A Priori Derivation of the Lattice Boltzmann Equation", *Physical Review E* 55(6): R6333, June 1997.

[63] M, Headrick. “Summary of Linear Algebra and Its Applications in Physics”, *Linear algebra in physics*, Brandeis University.

[64] A.S. Householder. *The Theory of Matrices in Numerical Analysis*. Courier Dover Publications, 2013.

[65] N. Jacobson. *Basic Algebra I*. Dover Publications, 2012.

[66] N.L. Johnson. *The Legacy of Group T-3*. Los Alamos National Laboratory. 26, June 2006.

[67] V.J. Katz. *A History of Mathematics*. Addison-Wesley, 1993.

[68] V.J. Katz, and B. Barton. “Stages in the History of Algebra with Implications for Teaching”, *Educational Studies in Mathematics* 66(2): 185-201, 2007.

[69] V.J. Katz. “Algebra and Its Teaching: An Historical Survey”, *The Journal of Mathematical Behavior* 16(1): 25-38, 1997.

[70] G.R. Kaye. *Indian Mathematics.*, JSTOR, 1919.

[71] D. Kirk. *NVIDIA CUDA Software and GPU Parallel Computing Architecture*. 2008.

[72] L.B. Kish. “End of Moore’s Law: Thermal (Noise) Death of Integration in Micro and Nano Electronics”, *Physics Letters A* 305(3): 144-149, 2002.

[73] I. Kleiner. *A History of Abstract Algebra*. Springer, 2007.

[74] W. Knight, “Two Heads Are Better Than One”, *IEEE Review*, September 2005.

[75] O. Knill. *Symmetric Matrices: Linear Algebra and Differential Equations. Mathematics* Math21b. Harvard University. Spring 2008.

[76] E. Kreyszig. *Advanced Engineering Mathematics*. Wiley, 1999.

[77] R.E. Larson, and B. Edwards. *Elementary Linear Algebra*. Cengage Learning, 2012.

[78] J. Latt. ”Palabos: CFD and Complex Physics” from www.lbmethod.org.

[79] D. Lay. *Linear Algebra and its Applications*. Addison Wesley, 1997.

[80] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. “NVIDIA Tesla: A Unified Graphics and Computing Architecture”, *IEEE Mirco* 28(2): 39-55, Apr. 2008.

[81] R. Lhner. *Applied CFD Techniques*. J. Wiley & Sons, 2001.

[82] H. Ltaief, S. Tomov, R. Nath, and J. Dongarra. “Hybrid Multicore Cholesky Factorization with Multiple GPU Accelerators”, *IEEE Transaction on Parallel and Distributed Systems*, 2010.

[83] L.-S. Luo. “Theory of Lattice Bolzmann Equation”, *China Center of Advanced Science and Technology*, October 9 - 13, 2000.

[84] P. Lutus. *Is Mathematics a Science?* N.p., 2008. Web. 13 Dec. 2013.

[85] J.A. Miller, R.J. Kee, and C.K. Westbrook. “Chemical Kinetics and Combustion Modeling”, *Annual Review of Physical Chemistry* 41(1): 345-387, 1990.

[86] B.B. Mandelbrot. *The Fractal Geometry of Nature*. Macmillan, 1983.

[87] R. Nath, S. Tomov, T. Dong, and J. Dongarra. “Optimizing Symmetric Dense Matrix-Vector Multiplication on GPUs”, *High Performance Computing*, 2011.

[88] D. Negrut. *High Performance Computing for Engineering Applications.* Course ME964, University of Wisconsin, 2008.

[89] J. Nickolls, and W.J. Dally. “The GPU Computing Era”, *Micro, IEEE* 30(2): 56-69, 2010.

[90] M.A. Nielsen, and I.L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2010.

[91] NIST Matrix Market. “Browse the Harwell-Boeing Collection.” from http://math.nist.gov/MatrixMarket/

[92] NVIDIA, *Compute Command Line Profiler: User Guide*. DU-05982-001_v03. November 2011.

[93] NVIDIA, *cublas Library: User Guide*. DU-06702-001_v5.5. May 2013.

[94] NVIDIA, *CUDA Compiler Driver NVCC: Reference Guide*. TRM-06721-001_v5.5. May 2013.

[95] NVIDIA, *CUDA Driver API: API Reference Manual*. TRM-06703-001_v5.5. May 2013.

[96] NVIDIA, *CUDA Developer Guide for NVIDIA Optimus Platforms: Reference Guide*. DG-06715-001_v5.5. May 2013.

[97] NVIDIA, *CUDA Runtime API: API Reference Manual.* v5.5. May 2013.

[98] NVIDIA, *CUDA C Best Practices Guide: Design Guide*. DG-05603-001_v5.5. May 2013.

[99] NVIDIA, *CUDA Samples: Reference Manual*. TRM-06704-001_v5.5. July 2013.

[100] NVIDIA, *CUDA C Programming Guide: Design Guide.* PG-02829-001_v5.5. May 2013.

[101] NVIDIA, *NVIDIA Nsight Visual Studio Edition 3.2 User Guide*

[102] NVIDIA, *CUPTI: User’s Guide.* DA-05679-001_v5.5. May 2013.

[103] NVIDIA, *cusparse Library*. DU-06709-001_v5.5. May 2013.

[104] NVIDIA, *Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cusparse and cublas*. WP-06720-001_v5.5. May 2013.

[105] NVIDIA, *NVIDIA CUDA Getting Started Guide for Microsoft Windows: Installation and Verification on Windows*. DU-05349-001_v5.5. May 2013.

[106] NVIDIA, *NVIDIA Performance Primitives (NPP)*. Version 5.5. March 27, 2013.

[107] NVIDIA, *Parallel Thread Execution ISA: Application Guide*. v3.2. May 2013.

[108] NVIDIA, *Thrust Quick Start Guide*. DU-06716-001_v5.5. May 2013.

[109] C. Obrecht. "Multi-GPU Implementation of the Lattice Boltzmann Method", *Computers and Mathematics with Applications* 65(2): 252-261, 2013.

[110] J.D. Owens, M. Houston, D. Luebke, and S. Green. "GPU Computing", *Proceedings of the IEEE* 96(5): 879-899, 2008.

[111] C. Ozcan, Caner, and B. Sen. "Investigation of the Performance of LU Decomposition Method using CUDA",. *Procedia Technology* 1(1): 50-54, 2012.

[112] P.-O. Persson. *Sparse Matrix Algorithms: Lecture 20*. MIT 18.335J/6.337J. Introduction to Numerical Methods. 26 November 2007.

[113] A.L. Porter, and J. Youtie. "How Interdisciplinary is Nanotechnology?", *Journal of Nanoparticle Research* 11(5): 1023-1041, 2009.

[114] W.H. Press, B.P. Flannery, S.A. Teuklosky, W.T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1992.

[115] P.J. Pritchard. *Fox and McDonald's Introduction to Fluid Mechanics.* Wiley, 2011.

[116] L. Puig.*History of Algebraic Ideas and Research on Educational Algebra.* 2004.

[117] A. Ralston, and P. Rabinowitz. *A First Course in Numerical Analysis*. Dover Publications, 1965.

[118] J. Richard, S. Girimaji, D. Yu, and H. Yu. *Lattice Boltzmann Method for CFD*. Department of Aerospace Engineering. Texas AM. 1 December 2003.

[119] J.F. Rutherford, and A. Ahlgren. *Science for all Americans*. Oxford University Press, 1991.

[120] L.L. Scharf. *Statistical Signal Processing*. Addison-Wesley, 1991.

[121] H. Schneider, and G.P. Barker. *Matrices and Linear Algebra*. Dover Publications, 1973.

[122] A. Shen, and N.K. Vereshchagin. *Basic Set Theory*. American Mathematical Society, 2002.

[123] M. Shenwei. “Navier-Stokes vs Lattice Boltzmann: Will it Change the Landscape of CFD?”, Sep. 22, 2011.

[124] A. Shilov. “Nvidia: Next-Generation Maxwell Architecture Will Break New Grounds,” from www.xbitlabs.com.

[125] G. Singer. *The History of the Modern Graphics Processor.*TechSpot, 27 Mar. 2013.

[126] F. Song, S. Tomov, and J. Dongarra.“Efficient Support for Matrix Computations on Heterogeneous Multi-Core and Multi-GPU Architectures”, *LAPACK Working Note*, 2011.

[127] I.S. Sokolnikoff, R.M. Redheffer, and J. Avents. “Mathematics of Physics and Modern Engineering”, *Journal of The Electrochemical Society* 105(9), 1958.

[128] J. Soria. “An investigation of the near wake of a circular cylinder using a video-based digital cross-correlation particle image velocimetry technique.” *Experimental Thermal and Fluid Science*, 12.2, 221-233, 1996.

[129] R.R. Stoll. *Set Theory and Logic*. Dover Publications, 1979.

[130] B.M. Sumer, and J. Fredsoe. *Hydrodynamics around cylindrical structures.* No 12. World Scientific, 1997.

[131] P. Suppes. *Axiomatic Set Theory*. Dover Publications, 1960.

[132] J.C. Thibault, and I. Senocak. “CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows”, *Proceedings of the 47th AIAA Aerospace Sciences Meeting.* 2009.

[133] J. Tolke. “Implementation of a Lattice Boltzmann Kernel using the Compute Unified Device Architecture Developed by NVIDIA”, *Computing and Visualization in Science* 13(1): 29-39, 2010.

[134] L. Vandenberghe. *Cholesky Factorization*. UCLA Electrical Engineering: EE103. Applied Numerical Computing. Fall 2011-2012.

[135] T. Vicsek, M. Cserz, and V.K. Horvth. ”Self-Affine Gowth of Bacterial colonies”, *Physica A: Statistical Mechanics and its Applications* 167(2): 315-321, 1990.

[136] V. Volkov, and J. Demmel. “LU, QR and Cholesky factorizations using Vector Capabilities of GPUs”, EECS Department, University of California, Berkeley, May 2008.

[137] E.W. Weisstein. *Cellular Automaton.* MathWorld.

[138] F. White *Fluid Mechanics* McGraw-Hill Higher Education, 1998.

[139] C. Wolf, G. Dotzler, R. Veldema, and M. Philippsen. “Object Support for OpenMP-Style Programming of GPU Clusters in Java”, *Advanced Information Networking and Applications Workshops*, 2013.

[140] W. Xian, and A. Takayuki. “Multi-GPU Performance of Incompressible Flow Computation by Lattice Boltzmann Method on GPU cluster”, *Parallel Computing* 37(9): 521-535, 2011.

[141] D. Yan, H. Cao, X. Dong, B. Zhang, and X. Zhang. “Optimizing Algorithm of Sparse Linear Systems on GPU”, *2011 Sixth Annual ChinaGrid Conference*, 2011.

[142] M. Yang. *Matrix Decomposition*. Electrical and Computer Engineering, Northwestern University.

[143] Y. Zhao. “Lattice Boltzmann Based PDE Solver on the GPU”, *The Visual Computer* 24(5): 323-333, 2008.

[144] D.G. Zill, and W.S. Wright. *Advanced engineering mathematics*. Jones & Bartlett Publishers, 2009.