

**CUDA OPTIMIZATION OF A CLASS OF ITERATIVE  
LINEAR ALGEBRA SOLVER**

BY

**LUTFI AZIZ FIRDAUS**

A Thesis Presented to the  
DEANSHIP OF GRADUATE STUDIES

**KING FAHD UNIVERSITY OF PETROLEUM & MINERALS**

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the  
Requirements for the Degree of

**MASTER OF SCIENCE**

In

**COMPUTER ENGINEERING**

**MAY 2017**

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS  
DHAHRAN 31261, SAUDI ARABIA


DEANSHIP OF GRADUATE STUDIES

This thesis, written by **LUTFI AZIZ FIRDAUS** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING**.

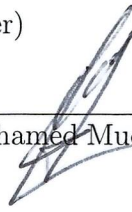
Thesis Committee

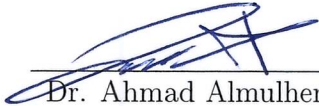


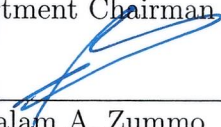
Dr. Mayez Al-Mouhamed  
(Adviser)



Dr. Muhammad Elrabaa  
(Member)

  
Dr. Muhamed Mudawar (Member)

  
Dr. Ahmad Almulhem  
Department Chairman

  
Dr. Salam A. Zummo  
Dean of Graduate Studies

11/6/17  
Date



©Lutfi Aziz Firdaus  
2017

*To my beloved father, may Allah grant him Jannat al-Firdaus*

# ACKNOWLEDGEMENTS

*All the praises is due to Allah alone, who has granted me the ability to finish my Master study in Computer Engineering Department, College of Computer Science and Engineering, King Fahd University of Petroleum and Minerals (KFUPM).*

*After that, I would like to thank my advisor, Dr. Mayez Abdullah Al-Mouhamed, for his patience, guidance, support, and encouragement that help me to finish this thesis. There are so much knowledge and the skills I have obtained under his supervision. Also, to Dr. Muhammad Elrabaa and Dr. Muhamed Mudawar as my thesis committee members, I would like to give my gratitude for helping me to complete my thesis.*

*Special thanks to my beloved father, Ikin Sodikin -may Allah bless him-, the one who support me fully to study and to complete this Master even though at the last moment he could not see me finishing it. Also to my mother, Siti Nurjanah Jamilah, for her presence, kindness, pray, and everything that I could not ever repay.*

*Many thanks and apologize I would like to say to my love of my life, Endang Muthiarini, and my son Fudhail Abdullah Firdaus for everything we have been through together because of this.*

*Last but not least, for Pa Usep and all Indonesian community member that I could*

*not mention name by name, thank you very much for our togetherness. You are just like my second family here.*

*May Allah reward you all with goodness.*



# TABLE OF CONTENTS

<b>ACKNOWLEDGEMENTS</b>	<b>v</b>
<b>LIST OF TABLES</b>	<b>xi</b>
<b>LIST OF FIGURES</b>	<b>xiii</b>
<b>ABSTRACT (ENGLISH)</b>	<b>xvii</b>
<b>ABSTRACT (ARABIC)</b>	<b>xix</b>
<b>CHAPTER 1 INTRODUCTION</b>	<b>1</b>
1.1 Iterative Methods . . . . .	4
1.2 Problem Statement . . . . .	5
1.3 Contributions to this Topic . . . . .	6
<b>CHAPTER 2 BACKGROUND</b>	<b>7</b>
2.1 GPU and CUDA . . . . .	7
2.1.1 GPU Architecture . . . . .	7
2.1.2 CUDA Programming . . . . .	11
2.1.3 Numerical Library . . . . .	13
2.2 Multi-GPU Programming . . . . .	15
2.3 Biconjugate Gradient Stabilized Method (BiCGStab) . . . . .	19
<b>CHAPTER 3 SPARSE MATRIX STORAGE FORMATS</b>	<b>21</b>
3.1 General Sparse Storage Format and Its SpMV . . . . .	21



3.1.1	Coordinate Format (COO) . . . . .	23
3.1.2	Compressed Storage Row Format (CSR) . . . . .	24
3.1.3	Block Compressed Storage Row Format (BSR) . . . . .	26
3.1.4	ELLPACK Format (ELL) . . . . .	26
3.1.5	Hybrid Format (HYB) . . . . .	28
3.1.6	Diagonal Format (DIA) . . . . .	30
3.2	Previous Work Studies . . . . .	32
<b>CHAPTER 4 LITERATURE REVIEW</b>		<b>39</b>
4.1	CUDA Application . . . . .	41
4.1.1	Ocean Modelling . . . . .	41
4.1.2	Artificial Intelligence . . . . .	41
4.1.3	Oil and Gas . . . . .	42
4.1.4	Math Compiler . . . . .	44
4.1.5	RSA Decryption . . . . .	45
4.2	CUDA Multi-GPU . . . . .	46
4.3	CUDA BiCGStab . . . . .	52
<b>CHAPTER 5 METHODOLOGY</b>		<b>55</b>
5.1	General Hepta BDIA (BDIA-GH) . . . . .	55
5.2	Sparse Matrix-Vector Multiplication for BDIA-GH . . . . .	64
5.3	BiCGStab with General Hepta . . . . .	69
5.4	Multi-GPU BiCGStab BDIA-GH . . . . .	70
5.5	Analysis of Deadlock in Inter-Block Synchronization . . . . .	73
5.5.1	Evaluating Number of Active Blocks on Kernel in Kepler K20Xm and Deadlock Condition with Experimental Validation . . . . .	75
5.5.2	General Conclusion . . . . .	82
<b>CHAPTER 6 RESULTS AND DISCUSSION</b>		<b>85</b>
6.1	Sparse Matrix-Vector Multiplication . . . . .	86

6.2	BiCGStab on Single GPU . . . . .	88
6.3	CUDA Memory Transfer . . . . .	91
6.4	BiCGStab on Multi GPU . . . . .	95
6.5	Overhead in Computing BiCGStab . . . . .	103
<b>CHAPTER 7 CONCLUSION</b>		<b>118</b>
<b>REFERENCES</b>		<b>119</b>
<b>VITAE</b>		<b>130</b>

# LIST OF TABLES

2.1	Device Specification for Tesla K80 . . . . .	10
5.1	Matrix Market File configuration . . . . .	65
5.2	BiCGStab Algorithm with Its Function Call kernels . . . . .	71
5.3	Validation Experiment for Analyzing Deadlock on inter-block GPU synchronization using Jacobi Algorithm . . . . .	84
6.1	SpMV Execution Time of Various Storage Formats (Single GPU)	88
6.2	SpMV Speedup of BDIA-GH over Other Storage Formats (Single GPU) . . . . .	89
6.3	SpMV FLOPS of Various Storage Formats (Single GPU) . . . . .	89
6.4	BiCGStab Execution Time on GH Matrices of Various Storage Formats (Single GPU) . . . . .	92
6.5	Speedup BiCGStab BDIA-GH over Other Formats (Single GPU) .	92
6.6	BiCGStab GFLOPS on GH Matrices of Various Storage Formats (Single GPU) . . . . .	94
6.7	Transfer rate between 2 GPUs with various memory access . . . .	94
6.8	Transfer rate between 4 GPUs with various memory access . . . .	94
6.9	Transfer rate between 8 GPUs with various memory access . . . .	95
6.10	Execution time of SpMV with Various Number of GPU(s) . . . .	97
6.11	Execution Time of 2 GPUs BiCGStab with Various Communication Model . . . . .	97
6.12	Speedup of 2 GPUs BiCGStab with Various Communication Model over Single GPU . . . . .	99

6.13 Execution Time of 4 GPUs BiCGStab with Various Communication Model . . . . .	99
6.14 Speedup of 4 GPUs BiCGStab with Various Communication Model over Single GPU . . . . .	101
6.15 Execution Time of 8 GPUs BiCGStab with Various Communication Model . . . . .	101
6.16 Speedup of 8 GPUs BiCGStab with Various Communication Model over Single GPU . . . . .	102
6.17 GFLOPS of BiCGStab on Various Number of GPU(s) . . . . .	102
6.18 Number of Kernel Calls on BiCGStab . . . . .	106
6.19 Number of Arithmetic Operation in BiCGStab . . . . .	106
6.20 Kernel Calls Execution Time of BiCGStab on various number of GPUs . . . . .	106
6.21 Kernel Calls Execution Time Percentage of BiCGStab . . . . .	107

# LIST OF FIGURES

2.1	Growth of GPU computing [9] . . . . .	8
2.2	SMX on Kepler Architecture [10] . . . . .	9
2.3	Warp Scheduler [10] . . . . .	10
2.4	Hyper-Q on Tesla Kepler K80 [10] . . . . .	11
2.5	RDMA GPUDirect [10] . . . . .	12
2.6	Description of CUDA Architecture [11] . . . . .	13
2.7	CUDA Threads and Memory Hierarchy [10] . . . . .	14
2.8	CPU-GPU communication schemes (a) inside CPU memory (b) between CPUs memory (c) between GPUs memory in same host (d) between GPUs memory in different host (e) CPU memory to GPU memory and vice versa in same system (f) CPU memory to GPU memory and vice versa in the different system [13] . . . . .	17
2.9	Paged host memory scheme (left), Pinned host memory scheme (right) . . . . .	18
2.10	PEX 8747 48-Lane, 5-Port PCI Express . . . . .	18
3.1	Examples of Sparse Matrices [2] . . . . .	22
3.2	COO representation of A with arrays row, col, and data [17] . . . . .	23
3.3	COO SpMV kernel memory access pattern with the arrays row col, and data [17] . . . . .	24
3.4	CSR representation of A with arrays ptr, indices, and data [17] . . . . .	25
3.5	CSR sparse matrix format using serial CPU SpMV kernel [17] . . . . .	25

3.6	CSR SpMV kernel memory access pattern using arrays indices and data [17] . . . . .	25
3.7	ELL representation of A with arrays data and indices [17] . . . . .	27
3.8	Example of the matrix that is suitable with ELL format [17] . . . . .	27
3.9	Example of the matrix that is not suitable for ELL format [17] . . . . .	28
3.10	ELL sparse matrix format SpMV kernel [17] . . . . .	29
3.11	Memory access pattern of ELL SpMV and its data linearization [17] . . . . .	29
3.12	Five diagonals sparse matrix [17] . . . . .	30
3.13	DIA representation of A in arrays data and offsets [17] . . . . .	31
3.14	Sparse matrix type that not suited well with DIA format [17] . . . . .	32
3.15	DIA SpMV memory access pattern and data array linearization [17] . . . . .	32
3.16	DIA sparse matrix format SpMV kernel [17] . . . . .	33
5.1	Example of General Hepta (GH) row . . . . .	56
5.2	Example of GH matrix with parameter J=2, H=4, I=2, Nc=2 . . . . .	57
5.3	Comparison between ELL and DIA storage format for GH . . . . .	58
5.4	Comparison between ELL indices and DIA offset . . . . .	59
5.5	Proposed BDIA-GH storage format that contains two array: data and offset . . . . .	60
5.6	Comparison between BDIA-GH and DIA in one row of block . . . . .	61
5.7	Flowchart of block dispatching order . . . . .	74
5.8	Block dispatching order result . . . . .	75
6.1	SpMV execution Time on Small Size Matrices of Various Storage Formats (Single GPU) . . . . .	90
6.2	SpMV execution Time on Large Size Matrices of Various Storage Formats (Single GPU) . . . . .	91
6.3	Speedup of BDIA-GH SpMV over Other Storage Formats (Single GPU) . . . . .	93
6.4	SpMV FLOPS of Various Storage Formats on Small Matrices (Single GPU) . . . . .	95

6.5	SpMV FLOPS of Various Storage Formats on Large Matrices (Single GPU) . . . . .	96
6.6	BiCGStab Execution Time on Small Size GH Matrices of Various Storage Formats (Single GPU) . . . . .	98
6.7	BiCGStab Execution Time on Large Size GH Matrices of Various Storage Formats (Single GPU) . . . . .	100
6.8	Speedup of BiCGStab BDIA-GH over Other Storage Formats (Single GPU) . . . . .	103
6.9	FLOPS on Small Size GH Matrices of BiCGStab in Various Storage Format (Single GPU) . . . . .	104
6.10	FLOPS on Large Size GH Matrices of BiCGStab in Various Storage Format (Single GPU) . . . . .	104
6.11	Memory transfer routes on 4 GPUs implementation . . . . .	105
6.12	Peer-to-Peer GPUDirect 4 GPUs implementation. (a) right communication, (b) left communication . . . . .	107
6.13	Transfer rate between 2 GPUs with various memory access . . . . .	108
6.14	Transfer rate between 4 GPUs with various memory access . . . . .	108
6.15	Transfer rate between 2 GPUs with various memory access . . . . .	109
6.16	Execution time of SpMV on various number of GPU(s) (Small Size)	109
6.17	Execution time of SpMV on various number of GPU(s) (Large Size)	110
6.18	Execution Time of 2 GPUs BiCGStab with Various Communication Model (Small Size) . . . . .	110
6.19	Execution Time of 2 GPUs BiCGStab with Various Communication Model (Large Size) . . . . .	111
6.20	Speedup of 2 GPUs BiCGStab over Single GPU with Various Communication Model (Small Size) . . . . .	111
6.21	Speedup of 2 GPUs BiCGStab over Single GPU with Various Communication Model (Large Size) . . . . .	112
6.22	Execution Time of 4 GPUs BiCGStab with Various Communication Model (Small Size) . . . . .	112

6.23 Execution Time of 4 GPUs BiCGStab with Various Communication Model (Large Size) . . . . .	113
6.24 Speedup of 4 GPUs BiCGStab over Single GPU with Various Communication Model (Small Size) . . . . .	113
6.25 Speedup of 4 GPUs BiCGStab over Single GPU with Various Communication Model (Large Size) . . . . .	114
6.26 Execution Time of 8 GPUs BiCGStab with Various Communication Model (Small Size) . . . . .	114
6.27 Execution Time of 8 GPUs BiCGStab with Various Communication Model (Large Size) . . . . .	115
6.28 Speedup of 8 GPUs BiCGStab over Single GPU with Various Communication Model (Small Size) . . . . .	115
6.29 Speedup of 8 GPUs BiCGStab over Single GPU with Various Communication Model (Large Size) . . . . .	116
6.30 FLOPS of BiCGStab on Various Number of GPU(s) (Small Size)	116
6.31 FLOPS of BiCGStab on Various Number of GPU(s) (Large Size)	117
6.32 Kernel Calls Execution Time of BiCGStab on various number of GPUs . . . . .	117



# THESIS ABSTRACT

**NAME:** Lutfi Aziz Firdaus  
**TITLE OF STUDY:** CUDA OPTIMIZATION OF A CLASS OF ITERATIVE  
LINEAR ALGEBRA SOLVERS  
**MAJOR FIELD:** COMPUTER ENGINEERING  
**DATE OF DEGREE:** May 2017

*Modern scientific computing focuses on developing applications using GPU (Graphics Processing Unit) because of the rapid growth in computing power and drop in the price of massively parallel accelerators. The growth in emerging architectures is also supported with powerful programming languages like the Compute Unified Device Architecture (CUDA) released by NVIDIA. The abundant parallel arithmetic hardware in GPUs largely exceeds that available with multi-core CPU (Central Processing Unit). Science simulations heavily depend on Iterative linear algebra solvers (ILAS). ILAS is one of the algorithms that could exploit this GPU massive parallelism provided by GPGPU (General Purpose Graphics Processing Unit) to accelerate science simulation. BiCGStab (Biconjugate Gradient Stabilized) is a relatively general solver for solving sparse systems of*

linear equations. *BiCGstab* is selected on this thesis to solve on implementing the reservoir simulation. The characteristic matrices that come from reservoir simulation allow defining the pattern and properties of the above matrix. The General Hepta (GH) sparse matrix is found to be the matrix pattern for reservoir simulation. GH must be handled effectively and efficiently especially for Sparse Matrix-Vector Multiplication (SpMV) that takes most of the computation time in the simulation. To optimize the storage, a new sparse matrix storage format called Block Diagonal General Hepta (BDIA-GH) is proposed in this work to accelerate the corresponding SpMV. Evaluation shows that the speedup of using BDIA-GH in the SpMV on single GPU is up to 3.0 compared to other storages like CSR (Compressed Storage Row) format, 2.8 compared to BSR (Block Compressed Storage Row) format, and 1.4 compared to HYB format. The optimization of SpMV, leads to the improvement on *BiCGStab* on a single GPU by 2.64 speedup over CSR, 2.4 over BSR, and 1.35 over HYB. With multi-GPU, the SpMV almost scaled perfectly. Speedup of 2 is gained by 2 GPUs over single GPU, 3.97 for 4 GPUs, and 7.97 for 8 GPUs. Multi-GPU *BiCGStab* also proposed as a novel approach to improve the computing performance. Speedup with 1.88 factor gained by using 2 GPUs over single GPU, and 3.28 by using 4 GPUs, and 4.38 by using 8 GPUs. This work contributes to development of scalable SpMV and *BiCGstab* for reservoir simulation on many-core.

## ملخص الرسالة

الاسم الكامل: لطفى عزيز فردوس  
عنوان الرسالة: كودا أوبتيميزاتيون أوف كلاس أوف إيتيراتييف لينير ألبير سولفرس  
التخصص: هندسة الحاسوب  
تاريخ الدرجة العلمية: شعبان ١٤٣٨ هـ

تركز الحوسبة العلمية الحديثة على تطوير التطبيقات باستخدام وحدة معالجة الرسومات (GPU) بسبب التطور السريع في قوة الحوسبة وانخفاض سعر المسرعات المتوازية بشكل كبير. التطور في عمارة الحاسب الناشئة مدعوم أيضا من لغات برمجة متطورة مثل معمارية حوسبة الاجهزة الموحدة (كودا) (CUDA) الصادرة عن نيفيديا. الاجزاء المسؤولة عن العمليات الحسابية المتوازية المتوفرة في وحدات معالجة الرسومات تتجاوز إلى حد كبير تلك المتوفرة في وحدة المعالجة المركزية متعددة النواة. تعتمد عمليات المحاكاة العلمية بشكل كبير على الطريقة التكرارية في حل معادلات الجبر الخطي (ILAS). (ILAS) هي واحدة من الخوارزميات التي يمكن أن تستغل هذا التوازي الضخم المقدم من وحدة معالجة الرسومات التي تقدمها وحدة معالجة الرسومات للاغراض العامة لتسريع المحاكاة العلمية. طريقة المرافق الثنائي المتدرج الثابت BiCGStab هو حل عام نسبيا لحل المعادلات الخطية للانظمة الخفيفة. اختيارنا طريقة المرافق الثنائي المتدرج الثابت BiCGStab في هذه الأطروحة لحل ينفذ على محاكاة الخزانات المائية الكامنة. خصائص المصفوفات المرتبطة بمحاكاة الخزانات المائية الكامنة تسمح بتحديد نمط وخصائص المصفوفة أعلاه. المصفوفة العامة السباعية الشبه صفرية تمثل نمط مصفوفة المحاكاة للخزانات المائية الكامنة. المصفوفة السباعية يجب التعامل معها بشكل فعال وكفاء وخاصة بالنسبة لعمليات الضرب بين المصفوفة والمتجه الشبه صفري (SpMV) والتي تأخذ معظم الوقت الحسابي في المحاكاة. لتحسين واستغلال التخزين، اقترحنا في هذا العمل صيغة جديدة لتخزين المصفوفة الشبه صفرية يسمى المصفوفة السباعية القطرية القاطعات (BDIA-GH) لتسريع عمليات الضرب بين المصفوفة والمتجه الشبه صفري (SpMV). التقييم اوضح أن سرعة استخدام المصفوفة السباعية القطرية القاطعات BDIA-GH لحل عمليات الضرب بين المصفوفة والمتجه الشبه صفري على وحدة المعالجة الرسومية الواحدة يصل إلى ثلاثة اضعاف مقارنة مع غيرها من طرق التخزين مثل ضغط التخزين المعتمد على الصف CSR ، 2.8 اضعاف مقارنة بضغط التخزين المعتمد على قطاعات الصف BSR ، و 1.4 مقارنة ب صيغة الخليط HYB . تحسين عمليات الضرب بين المصفوفة والمتجه الشبه صفري (SpMV) ، يؤدي إلى تحسين طريقة المرافق الثنائي المتدرج الثابت BiCGStab على وحدة معالجة رسومية واحدة ب 2.64 ضعف على ضغط التخزين المعتمد على الصف CSR، 2.4 اضعاف مقارنة بضغط التخزين المعتمد على قطاعات الصف BSR ، و 1.35 مقارنة ب صيغة الخليط HYB . مع تعدد وحدات المعالجة الرسومية، عمليات الضرب بين المصفوفة والمتجه الشبه صفري (SpMV) وتتناسب بشكل مناسب جدا. التسريع يصل الى 2 باستخدام وحدتي معالجة رسومية مقارنة بوحدة معالجة رسومية واحدة، و 3.97 ل 4 وحدات معالجة الرسومات، و 7.97 ل 8 وحدات معالجة الرسومات. طريقة المرافق الثنائي المتدرج الثابت BiCGStab على وحدات معالجة رسومية متعددة اقترحت ايضا كنهج جديد لتحسين أداء الحوسبة. ليصل الى 1.88 ضعف باستخدام وحدتي معالجة رسومات مقارنة بوحدة معالجة رسومية، و 3.28 باستخدام 4 وحدات معالجة الرسومات، و 4.38 باستخدام 8 وحدات معالجة رسومات. ويسهم هذا العمل في تطوير وتحسين عمليات الضرب بين المصفوفة والمتجه الشبه صفري (SpMV) و طريقة المرافق الثنائي المتدرج الثابت BiCGStab لمحاكاة الخزانات المائية الكامنة على المعالجات متعددة النواة.

# CHAPTER 1

## INTRODUCTION

Modern scientific and engineering applications are increasing in size and complexity due to the need for higher resolution in large scale modeling and simulations. These applications dominant a wide range of daily engineering and scientific applications. Large scale scientific simulations generally need high-performance computing to accelerate the simulation time and/or increasing the simulation accuracy. Solving large system of linear equations requires a lot of computation time using both direct solvers and iterative approaches. Most science simulations spend a significant fraction of time in the solver algorithm [1].

Usually, two solver approaches are most commonly used, which are the direct and the iterative linear algebra methods. The direct methods are more reliable and more accurate but require much more storage than the other methods, which make them difficult to scale. Excessive transfer and storage of data in a massive parallel computing system may easily become the bottleneck in addition to the high computational complexity. The iterative methods are more scalable, so large

sparse systems have abundant data parallelism, which favor the iterative methods over the direct methods especially when a high accuracy is not needed. The Krylov subspace solvers [1] represents the iterative methods to solve large sparse linear systems. Specifically, the Bi-conjugate Gradient Stabilized (BiCGstab) algorithm is one example of techniques that belongs to the above sub-space of solvers.

The most computational effort in science simulation is the solving of large sparse linear systems. These approaches require linear algebra operations such as matrix-vector multiplication, norm, dot product, vector scaling, and summation of vectors [2].

Modeling of various modern scientific and engineering applications requires repetitive solving of a large sparse system of linear equations. Hence the use of efficient sparse matrix data structure is the pre-requisite for the efficient implementation of these application on wide SIMD (Single Instruction Multiple Data) such as GPUs and other many core. Several bottlenecks which may limit the performance of the GPUs which are the low efficiency due to the sparse structure of above matrices and the increased overhead due to the irregular accesses to the memory which may cause a limitation on the memory bandwidth [3].

Offloading all computation of matrix and vectors to the GPU is considered a straightforward method to utilize the GPU accelerator to implement a Krylov subspace solver using the functions available in the numerical libraries. Although, this approach provides good improvement in the performance comparing to CPU-based implementation, sometimes the high capability of these accelerators

is not efficiently exploited due to the limitations caused by the operations of the linear algebra functions available in the numerical libraries [4], [5].

The GPU acceleration provides tremendous computational power provided that it is properly programmed to take advantage of all its abundant parallelism [6].

The Compute Unified Device Architecture (CUDA) programming language is developed by NVIDIA. CUDA is designed as a general purpose programming model [2] which has enough control of the hardware to control the explicit memory system, distribute the work over the available parallel compute units, and provide some tools for run-time profiling.

The sparse matrix-vector multiply (SpMV) is considered one of the most important computational and time-consuming kernel for a wide range of engineering and scientific applications ranging from structural mechanics to quantum physics including fluid dynamics. Thus, to achieve scalable performance for these applications, it is important to optimize the SpMV by developing customized sparse matrix storage format and efficiently utilizing the underlying GPU architecture to optimize its operations.

To decrease the overhead and avoid the bottlenecks mentioned above, many studies have proposed some storage formats for sparse matrices to take advantage of their data layouts in optimizing the memory access pattern and to reduce the matrix storage space in GPU memory. Each of these formats has different storage requirements and computational properties.

## 1.1 Iterative Methods

Solving systems of linear equations in scientific computing is one of the most common encountered problems [7]. Given a known matrix  $A$  and a known vector  $b$ , the problem is to find a solution  $x$  satisfying:

$$Ax = b \tag{1.1}$$

Gaussian Elimination (GE) is the most straightforward approach to solve the above system of linear equations. Although GE provides more accurate solution compared to other methods, it has one important drawback which is the excessive storage which is a major problem to scalability. This problem can be alleviated by iterative methods.

Iterative methods are not very accurate compared to direct methods but have abundant parallelism and can be engineered to produce a refined solution. The solution of this method could approach the exact solution within the margin of error determined by the user. By using initial guess as the value of  $x$ , the  $x$  may converge to an exact solution using more iterations. On every iteration, the solution will be updated and the margin of error will be reduced if some conditions are satisfied such the use of proper preconditioning.

## 1.2 Problem Statement

The high-performance computing technology nowadays is escalating rapidly. Devices and architectures are improving along with so many research conducted to maximize computer architecture performance. But programming with optimal code on high performance computing is not easy. Certain problems usually need to be solved by considering the detailed knowledge of the hardware. There is no general solution for implementing all of the possible programming optimizations to account for the hardware complexity. That is also the case for iterative linear algebra solver with sparse matrix vector multiplication. Several storage formats have been proposed to accommodate many kinds of sparse matrices, but still none of them provide the solution in best practices. Implementation on BiCGStab as the chosen iterative linear algebra solver also needs to be optimized. That is why several questions arise to address these problems. Here are some critical questions that our work has attempted to answer:

- What is the best storage scheme for sparse matrix on reservoir simulation?
- How to implement matrix vector multiplication that suits well with the storage scheme proposed for reservoir simulation?
- How to optimize the implementation of the BiCGStab to utilize the available hardware using CUDA?



## 1.3 Contributions to this Topic

In this thesis we developed a customized storage format that is suitable to solve reservoir simulation problems. The storage is called BDIA-GH (Block DIA General-Hepta) Format. Sparse matrix-vector multiplication SpMV uses the above storage format as one of its problem optimizations. We first present a methodology for optimizing SpMV and BiCGStab algorithm using CUDA programming for one GPU. Next, we extend our work to a cluster of GPUs and present a methodology for Multi-GPU implementation of SpMV and BiCGStab using a shared virtual memory at the server level. We evaluate performance using execution time, speedup, and Flops performance and assess the scalability of the proposed implementations. Also to the best of our knowledge, our work is the first implementation BiCGStab on multi-GPU with several optimizations.

## CHAPTER 2

# BACKGROUND

## 2.1 GPU and CUDA

### 2.1.1 GPU Architecture

The new architecture of GPU with Tesla leads to very highly efficient general-purpose parallel computing application [8]. Because this GPU architecture could be programmed directly in C with CUDA, the number of research on this general-purpose GPU (GPGPU) computing escalates very quickly (figure 2.1).

This programmability of GPU in C with CUDA is available in many kinds of computing devices like servers, workstations, desktops, and laptop. For example, as shown in Table 2.1, Tesla K80 that has CUDA computing capability of 3.7 has 13 Streaming Multiprocessors (MP) that called as SMX with 192 single-precision CUDA Cores on each SMX (figure 2.2).

Each Tesla K80 GPU has a total 2496 single-precision CUDA Cores. Each

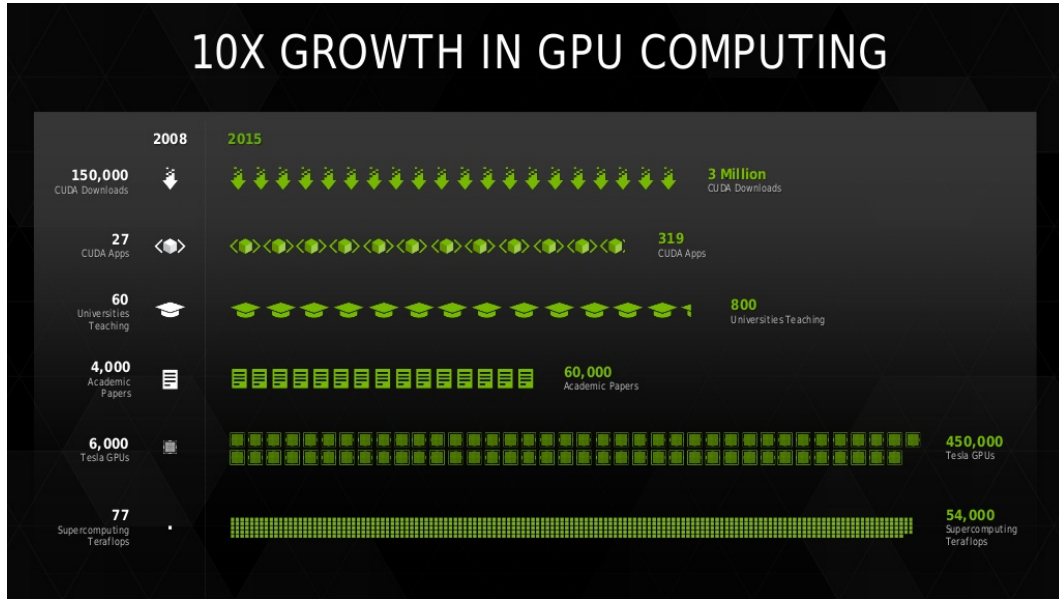


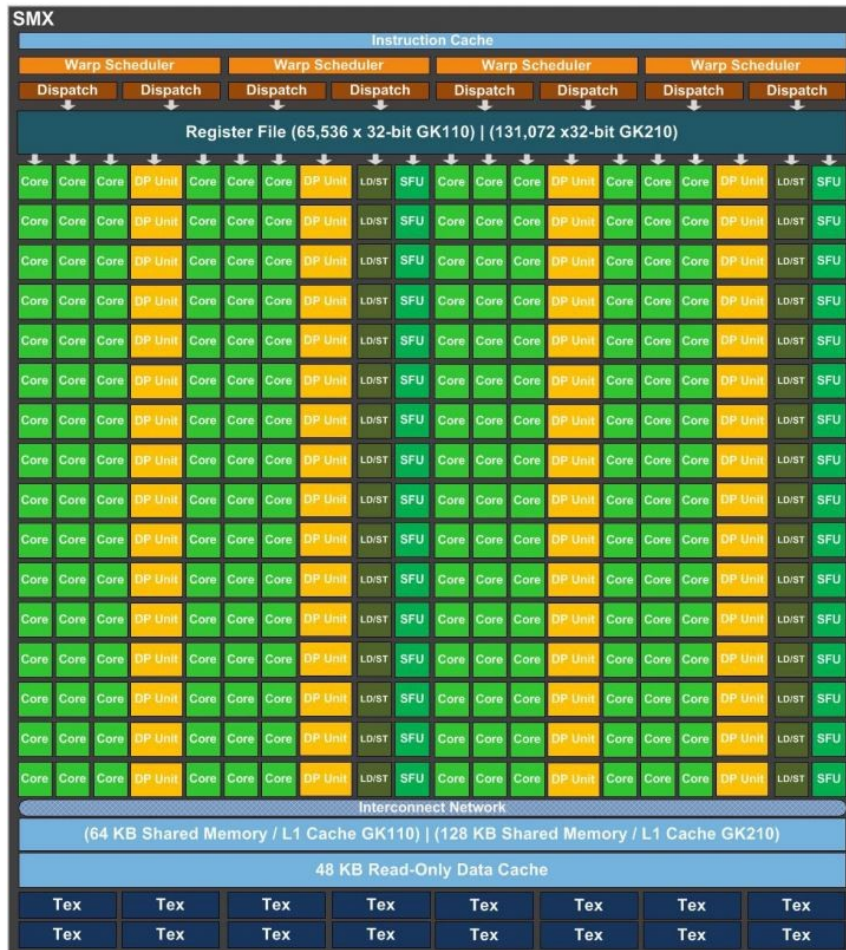
Figure 2.1: Growth of GPU computing [9]

SMX has 2048 maximum number of threads. It means maximum number of active thread at a time is:

$$N_{max\_active\_threads} = N_{SMX} \times N_{active\_thread/SM} = 13 \times 2048 = 26624 \quad (2.1)$$

Every 32 threads are grouped as one warp. In every SMX, there are 4 warp schedulers with 2 instruction dispatch unit for each scheduler to schedule the work on warps (figure 2.3).

For each GPU, the global memory is 11441 MBytes and shared memory per block is 49152 bytes. There are also 64 double-precision units, 32 special function units (SFU), and 32 load/store unit (LD/ST). On Kepler architecture, the CUDA cores could run tasks simultaneously with Hyper-Q feature (figure 2.4). Instead of using only one single hardware work queue for all the streams, the Kepler architecture supports each stream to work in parallel using separate work queues.

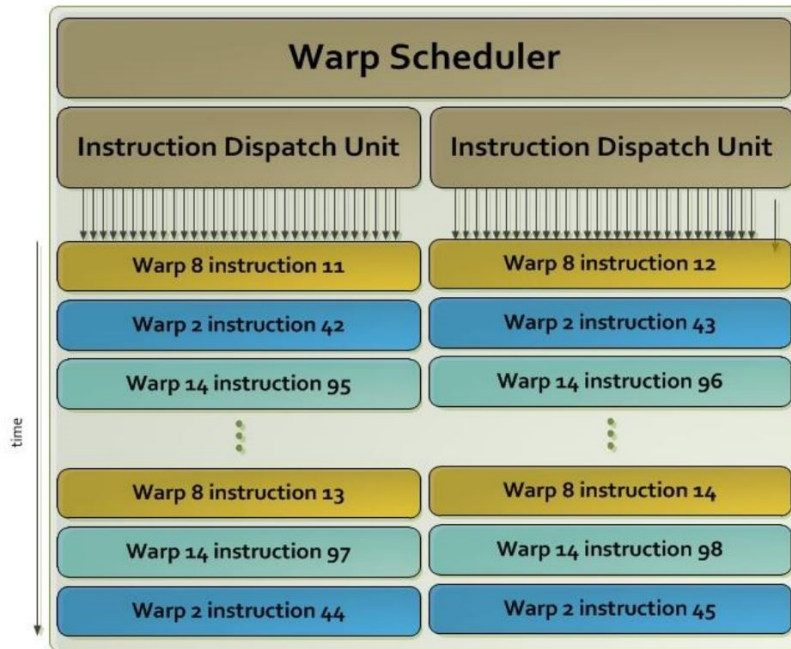


SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

Figure 2.2: SMX on Kepler Architecture [10]

Table 2.1: Device Specification for Tesla K80

Device Specification for Tesla K80	
CUDA Driver Version / Runtime Version	8.0 / 8.0
CUDA Capability Major/Minor version number:	3.7
Total amount of global memory:	11441 MBytes
(13) Multiprocessors, (192) CUDA Cores/MP:	2496 CUDA Cores
GPU Max Clock rate:	824 MHz (0.82 GHz)
Memory Clock rate:	2505 Mhz
Memory Bus Width:	384-bit
L2 Cache Size:	1572864 bytes
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Support host page-locked memory mapping:	Yes
Device has ECC support:	Enabled
Device supports Unified Addressing (UVA):	Yes



Each Kepler SMX contains 4 Warp Schedulers, each with dual Instruction Dispatch Units. A single Warp Scheduler Unit is shown above.

Figure 2.3: Warp Scheduler [10]

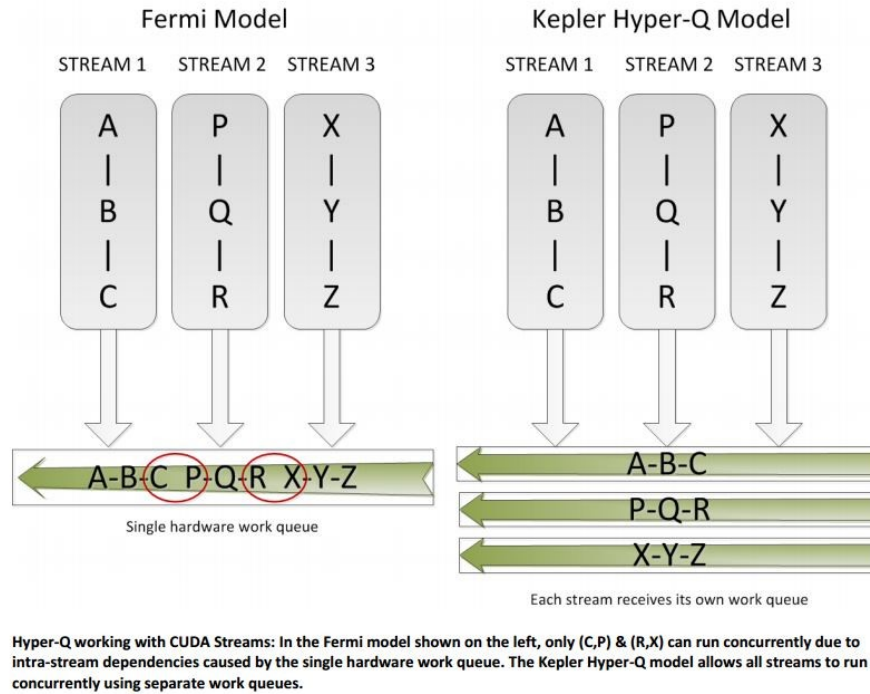


Figure 2.4: Hyper-Q on Tesla Kepler K80 [10]

Other features that are supported by this architecture of Kepler is GPUDirect (figure 2.5) that enable multi-GPU system to communicate each other without the need of host to be involved. The communication goes through the NIC (Network Interface Card) using RDMA (Remote Direct Memory Access) feature.

### 2.1.2 CUDA Programming

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model for many programming languages including C/C++ that enable the programmer to do programming in software and hardware side by using NVIDIA GPU.

A CUDA application could call a parallel program to be executed in NVIDIA

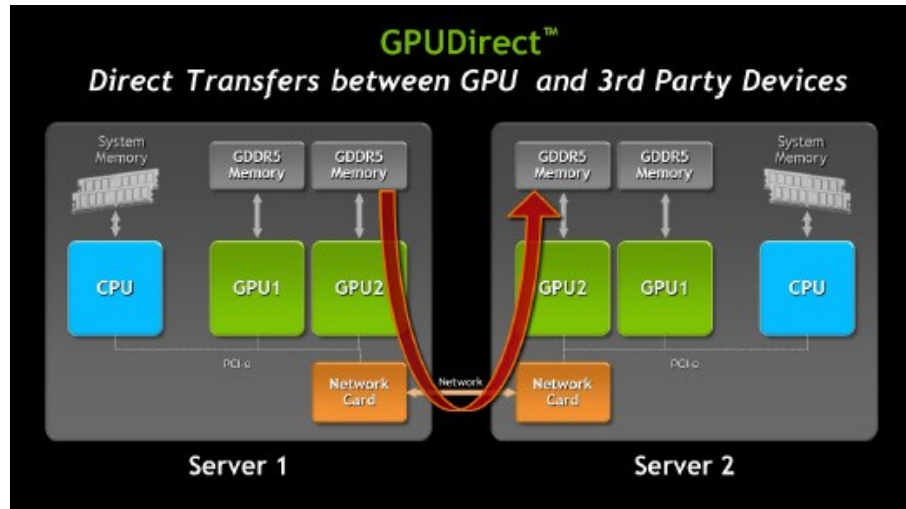


Figure 2.5: RDMA GPUDirect [10]

GPU CUDA enabled device that called as kernel. This kernel invoking many threads in parallel to work on the CUDA core inside GPU. A kernel is a set of thread block organized as a 1D, 2D, or 3D grid as shown in figure 2.6.

In every thread block, each thread executes the same program from the kernel and run all created threads in parallel using the SIMD (Single Instruction Multiple Data) model. Figure 2.7 shows that every thread has own private local memory, and the thread block has Shared Memory that could be accessed by every thread in the thread block. The global memory is accessed by all thread blocks in the grid. The mapping of CUDA application into the hardware is as follows: a GPU could execute one or more kernel grids. The SMX and CUDA cores in the GPU works on their assigned thread blocks.

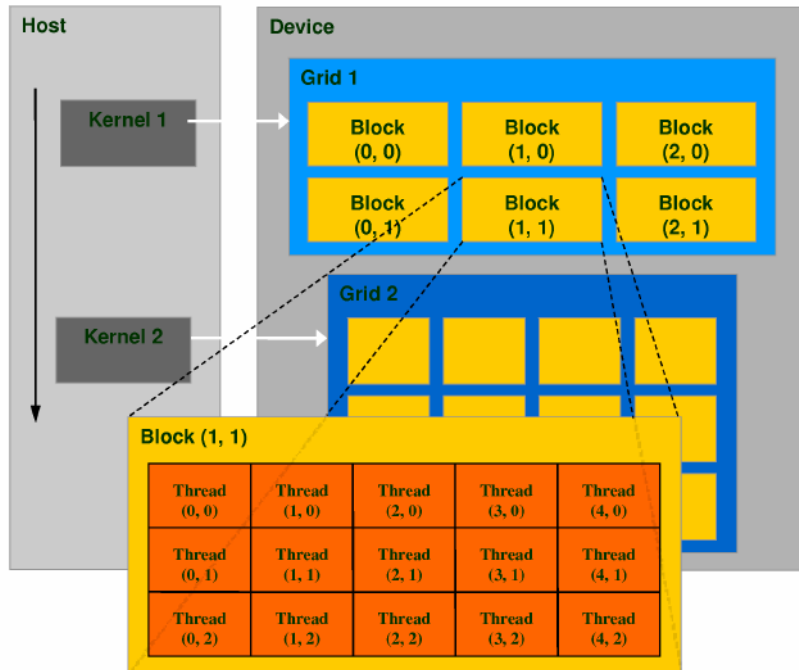


Figure 2.6: Description of CUDA Architecture [11]

### 2.1.3 Numerical Library

As parallel programming in CUDA is complex, researcher use numerical libraries for the basic algebra operators and many other math function to alleviate the problem of developing optimized code for all involved operations. This greatly improves the use of the NVIDIA CUDA for programming scientific computing applications. Therefore, there are many numerical libraries available to be used for linear algebra like BLAS and Sparse, FFT, seismic imaging, etc. Our focus in this thesis is on NVIDIA CUBLAS and CUSPARSE numerical libraries.



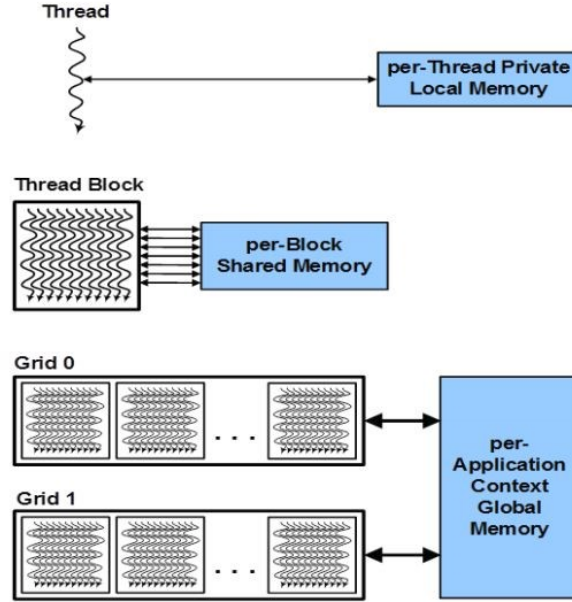


Figure 2.7: CUDA Threads and Memory Hierarchy [10]

### 2.1.3.1 CUBLAS

CUBLAS (CUDA Basic Linear Algebra Subroutines) is a library of BLAS (Basic Linear Algebra Subroutines) that contains many operators for solving linear algebra problems using CUDA. The operation is divided into 3 level functions. Level-1 is scalar-vector operation, level-2 is matrix-vector operation, Level-3 is matrix-matrix operation. Dense matrix is assumed in this library. The examples operation of Level-1 operations are copying vector to other vector (e.g. `cublasScopy`), calculating dot product of two vectors (e.g. `cublasSdot`), axpy operation that multiply vector  $x$  by the scalar  $a$  and adds it to the vector  $y$  (e.g. `cublasSaxpy`). The examples for level-2 CUBLAS operations are symmetric banded matrix-vector multiplication (`cublasSsbmv`), symmetric packed matrix-vector multiplication (`cublasSspmv`), and many more. As for level-3, the example of operations are matrix-matrix multiplication (`cublasSgemm`),

symmetric matrix-matrix multiplication (cublasSsymm), hermitian matrix-matrix multiplication (CublasChemmm), and others.

### **2.1.3.2 CUSPARSE**

CUDA Sparse Matrix library (CUSPARSE) is a library of BLAS (Basic Linear Algebra Solver) that contains many operators for solving sparse linear algebra problem using CUDA but specialized for working on sparse matrices. This library supports some general purpose sparse storage formats like ELL/HYB, CSR, Blocked CSR (BSR), and COO. These assume general sparse distribution of non-zeros (NZs) without specific regularity. Just like CUBLAS, CUSPARSE also has three level operation. Level-1 is sparse vector and dense vector operations, level-2 is sparse matrix and dense vector operations, and level-3 is sparse matrix and dense vector operation (tall matrix). There are also routines for sparse matrix by sparse matrix addition and multiplication. Also in this library, the storage formats conversion is available between COO to other formats and vice versa.

## **2.2 Multi-GPU Programming**

Multi-GPU programming is becoming a new trend in recent years that attracts many researchers to carry out research on parallel application scalability over a set of cooperating GPUs [12] that communicate using share host memory or using some dedicated links. There are many advantages on implementing applications on Multi-GPU systems. Firstly, Multi-GPU could accelerate the computation of

scientific applications. Secondly, Multi-GPU implementation could handle larger problem size because more memory is available. Furthermore, it could save power by using Multi-GPU on the same node so it could amortize the server cost among more GPUs.

Performance of Multi-GPU systems depends on how communication could be done within GPUs and CPU(s). Figure 2.8 shows the memory transfer path in Multi-GPU communication [13]. This octo-GPU system illustrating 6 kinds of memory transfer paths. The type of data transfers are:

1. from the host memory to other host memory,
2. from host memory to own host memory,
3. from device memory to other device memories in the same host,
4. from device memory to other device memory in different host,
5. from device to host and host to device, and
6. from device to other hosts, and from other hosts to own device.

Besides the transfer paths, the communication model also determined by how the data will be allocated in the host memory. Figure 2.9 shows the difference between paged and pinned memory scheme in host. By default, the data in the host will be stored in paged memory. But in Tesla "Kepler" K80, as shown in table 2.1, it support host page-locked memory mapping. It means the data in the host could be stored in pinned memory so the transfer path from host to device or vice versa

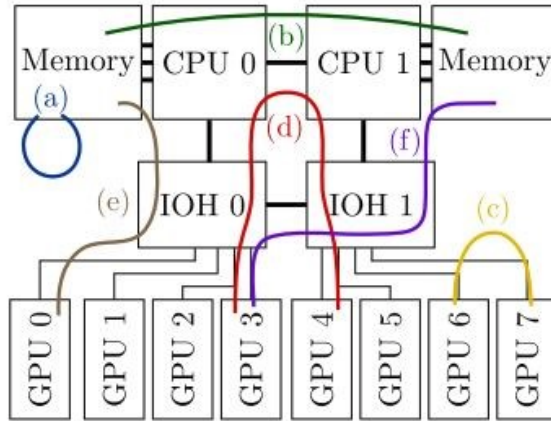


Figure 2.8: CPU-GPU communication schemes (a) inside CPU memory (b) between CPUs memory (c) between GPUs memory in same host (d) between GPUs memory in different host (e) CPU memory to GPU memory and vice versa in same system (f) CPU memory to GPU memory and vice versa in the different system [13]

could go directly from pinned memory to DRAM instead of from paged memory to pinned memory and then stored into DRAM. But there is limitation of using pinned memory, because by default this memory is reserved as the temporary memory and if it reserved for certain data, then available physical memory to be used by other applications will be reduced. The other component that effected the performance of communication is PCI-e bridge. PEX 8747 (figure2.10) is PCI Express that has 48 lanes with 5 ports. Maximum latency of this PCI-e is 100ns. The configuration of this PCI-e is by using 4 port from the devices, and 1 port to the host. Each port from the device has 8 lanes, while the port to the host is 16 lanes. This number of ports could make some limitation to the number of connection from devices to host.

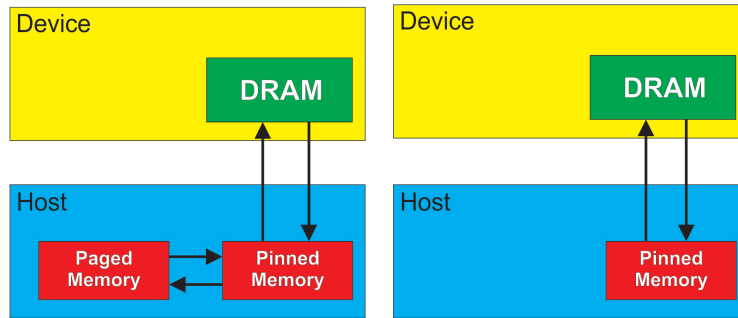


Figure 2.9: Paged host memory scheme (left), Pinned host memory scheme (right)

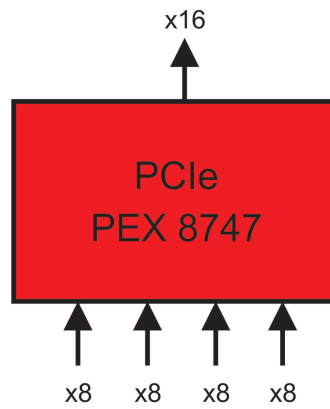


Figure 2.10: PEX 8747 48-Lane, 5-Port PCI Express

## 2.3 Biconjugate Gradient Stabilized Method (BiCGStab)

On solving a system of linear equations, there are two kinds of iterative methods. First is stationary iterative methods and the second is Krylov subspace methods that has more general implementation of matrices. Stationary iterative methods examples are Successive over-relaxation method, Gauss-Seidel method, and Jacobi method. While Krylov methods examples are generalized minimal residual method (GMRES), conjugate gradient (CG), and biconjugate gradient method (BiCG) that has variant called biconjugate gradient stabilized method (BiCGStab) that could be used for non-symmetric linear system and could be used when there is no transposed matrix available. This method also gives faster and smoother convergence than original BiCG [14].

From algorithm 1, we could see that there are several reductions and global writes that must be done cooperatively by all blocks and all kernels especially for Multi-GPU BiCGStab implementation. At the step 7, 11, and 14 the reductions are occurring. It means all the GPUs should communicate to gather the dot products. Before SpMV (at step 10 and 13), a global synchronization also has to be called to make sure the operation on SpMV using previous arrays is correct because of the communication between GPUs and CPU to transfer parts of the array from previous operation on devices to host and then host to devices. For all of the SpMV operations are using our own code BDIA-GH and also CUSPARSE

---

**Algorithm 1** BiCGStab algorithm

---

- 1:  $\mathbf{x}_0$  is an initial guess
- 2:  $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
- 3: Choose an arbitrary vector  $\hat{\mathbf{r}}_0$  such that  $(\hat{\mathbf{r}}_0, \mathbf{r}_0) \neq 0$ , e.g.,  $\hat{\mathbf{r}}_0 = \mathbf{r}_0$
- 4:  $\rho_0 = \alpha_0 = \omega_0 = 1$
- 5:  $\mathbf{v}_0 = \mathbf{p}_0 = \mathbf{0}$
- 6: **for**  $i = 0$  **to**  $i_{max}$  **do**
- 7:    $\rho_i \leftarrow (\hat{\mathbf{r}}_0, \mathbf{r}_{i-1})$
- 8:    $\beta \leftarrow (\rho_i / \rho_{i-1})(\alpha / \omega_{i-1})$
- 9:    $\mathbf{p}_i \leftarrow \mathbf{r}_{i-1} + \beta(\mathbf{p}_{i-1} - \omega_{i-1}\mathbf{v}_{i-1})$
- 10:    $\mathbf{v}_i \leftarrow \mathbf{A}\mathbf{p}_i$
- 11:    $\alpha \leftarrow \rho_i / (\mathbf{r}_0, \mathbf{v}_i)$
- 12:    $\mathbf{s} \leftarrow \mathbf{r}_{i-1} - \alpha\mathbf{v}_i$
- 13:    $\mathbf{t} \leftarrow \mathbf{A}\mathbf{s}$
- 14:    $\omega_i \leftarrow (\mathbf{t}, \mathbf{s}) / (\mathbf{t}, \mathbf{t})$
- 15:    $\mathbf{x}_i \leftarrow \mathbf{x}_{i-1} + \alpha\mathbf{p}_i + \omega_i\mathbf{s}$
- 16:   If  $\mathbf{x}_i$  accurate enough, then quit
- 17:    $\mathbf{r}_i \leftarrow \mathbf{s} - \omega_i\mathbf{t}$
- 18: **end for**

---

library, while other operations like vector dot products (step 7, 11, 14), axpy operation (step 9, 12, 15, 17), and scal (step 9) are using CUBLAS library.

## CHAPTER 3

# SPARSE MATRIX STORAGE FORMATS

### 3.1 General Sparse Storage Format and Its SpMV

Matrix operation often used in scientific computations are mainly needed for implementing iterative linear algebra solver. From the real applications, it is found that matrix to be used in scientific computing is in form of sparse matrices (figure 3.1) [15]. By definition, sparse matrix defined as "The matrix may be sparse, either with the nonzero elements concentrated on a narrow band centered on the diagonal or alternatively they may be distributed in a less systematic manner. We shall refer to a matrix as dense if the percentage of zero elements or its distribution is such as to make it uneconomic to take advantage of their presence." [16].



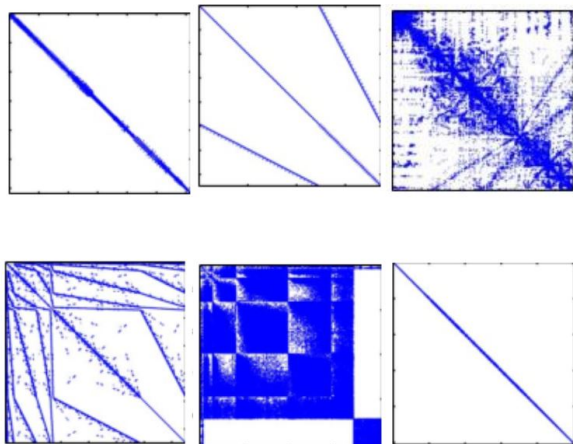


Figure 3.1: Examples of Sparse Matrices [2]

The distribution of zeros and non-zeros in matrices may considerably vary for each case. But generally, [15] these matrices can be grouped into two categories. The first category includes the problems which have no 2D/3D geometry. The second category is for the problems with 2D/3D geometry. In each category there are sub-categories that grouped by the type of computation. For example, circuit simulation problem, directed graph, linear programming problem that lies into the problems with no 2D/3D geometry, where materials problem, thermal problems, computational fluid dynamics problem are lies into the problems with 2D/3D geometry.

Because of the variation in the type of sparse matrix, Bell et. al. [17] proposed a general storage format that could help computing and manipulating the sparse matrix to achieve better performance.

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

$$\begin{aligned} \text{row} &= [0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3] \\ \text{col} &= [0 \ 1 \ 1 \ 2 \ 0 \ 2 \ 3 \ 1 \ 3] \\ \text{data} &= [1 \ 7 \ 2 \ 8 \ 5 \ 3 \ 9 \ 6 \ 4] \end{aligned}$$

Figure 3.2: COO representation of A with arrays row, col, and data [17]

### 3.1.1 Coordinate Format (COO)

The simplest storage format is the Coordinate (COO) of the non-zero elements. This storage format basically will have stable performance on every kind of sparse matrices, because the required storage capacity is linear with the number of non-zero in the matrix regardless its structure. This scheme stores row and column indices explicitly. Therefore, this storage formats have three arrays: an array to store the data, an array to store the row indices and an array to store the column indices (figure 3.2). Because the storage format is straight forward, the SpMV application in this format is also straight forward. The parallelization is done by dividing the data among the threads. Because the indices are all stored explicitly, any access to the data requires accessing the row and column indexing. The drawback of this scheme is the access to the memory is not coalesced manner which makes it inefficient in using the parallel memory bandwidth (figure 3.3).

row	[0 0 1 1 2 2 2 3 3]
col	[0 1 1 2 0 2 3 1 3]
data	[1 7 2 8 5 3 9 6 4]
Iteration 0	[0 1 2 3                   ]
Iteration 1	[                   0 1 2 3 ]
Iteration 2	[                                   0]

Figure 3.3: COO SpMV kernel memory access pattern with the arrays row col, and data [17]

### 3.1.2 Compressed Storage Row Format (CSR)

Basically, Compressed Storage Row format (CSR) is like the extension of COO format. All the values in matrix data are stored in row-major order and the column indices for the corresponding value of data is stored explicitly. But the difference with the COO format is instead of storing all the row numbering explicitly, CSR stores the row implicitly using a pointer ptr to determine the address of the first and last row elements (figure 3.4). The SpMV operation requires loading the values of the data after accessing the row and the column indices on each row (figure 3.5). Therefore, parallelizing the SpMV is based on assigning a separate thread to each row. The problem of this implementation are the data and the column indices which are stored contiguously but their access is not in coalesced manner (figure 3.6). There are some proposed approach to overcome this problem, but not discussed in detail.

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

```
ptr = [0 2 4 7 9]
indices = [0 1 1 2 0 2 3 1 3]
data = [1 7 2 8 5 3 9 6 4]
```

Figure 3.4: CSR representation of A with arrays ptr, indices, and data [17]

```
__host__ void
spmv_csr_serial(const int num_rows,
               const int * ptr,
               const int * indices,
               const float * data,
               const float * x,
               float * y)
{
    for(int row = 0; row < num_rows; row++){
        float dot = 0;

        int row_start = ptr[row];
        int row_end = ptr[row+1];

        for (int jj = row_start; jj < row_end; jj++)
            dot += data[jj] * x[indices[jj]];

        y[row] += dot;
    }
}
```

Figure 3.5: CSR sparse matrix format using serial CPU SpMV kernel [17]

```
indices [0 1 1 2 0 2 3 1 3]
data [1 7 2 8 5 3 9 6 4]

Iteration 0 [0 1 2 3 ]
Iteration 1 [ 0 1 2 3 ]
Iteration 2 [ 2 ]
```

Figure 3.6: CSR SpMV kernel memory access pattern using arrays indices and data [17]

### 3.1.3 Block Compressed Storage Row Format (BSR)

Block Compressed Sparse Row (BSR) is another version of compressed storage row (CSR). This scheme takes advantages of sparse matrices that have non-zeros in a group of neighbours, which can be considered as a data block. In scientific computation, the blocked structure represents interaction between cells that have more than one state variable. The general idea of this storage scheme is the same with CSR but in blocked version. Instead of storing each value of the data in row major order, they combine the row by the size of the block and put the block values into the data array with row major order of the block in block rows and also row major order within the block. Inside the block, there at least one number of non-zero. Therefore, it is not suitable if the non-zero elements in the sparse matrix is highly scattered because the block data will store too many zero elements. Other variations of this scheme stores the column indices of the blocks, pointers to each block in the matrix to determine the first end of the last block in a row of the matrix.

### 3.1.4 ELLPACK Format (ELL)

ELLPACK storage format is more specific compared to previous mentioned storage format because this scheme is suitable when the maximum number of non-zeros in the row of the matrix ( $K$ ) is close to the average number of non-zero in other rows. Otherwise, like in figure 3.9 the data and indexing arrays will store too many non-zeros. Figure 3.8 shows some suitable matrices for this scheme.

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

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

Figure 3.7: ELL representation of A with arrays data and indices [17]

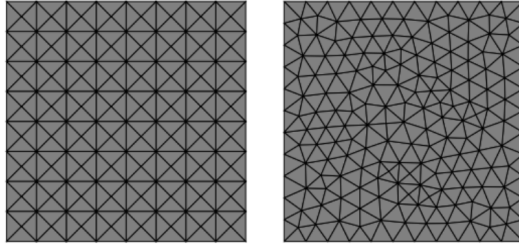


Figure 3.8: Example of the matrix that is suitable with ELL format [17]

This format is constructed by grouping the non-zero elements in every row of the matrix (assuming  $M \times N$  matrix) to the left side of the matrix and the zeros will be on the right. The row ( $K^{th}$ ) that has the maximum number of non-zero will be the limit and the zeros on the right side of  $K^{th}$  row that will be trimmed so the new data matrix will be  $M \times K$ . Besides the data, other array called indices are used in this format for indexing the column. ELL scheme stores the row implicitly and stores the column indices explicitly in an array like COO (figure 3.7).

Figure 3.10 shows how SpMV with this ELL format has been done. Generally, kernel of ELL is similar to DIA format, ELL can be helpful to parallelize the computation by assigning one thread per row. But because the column indices is stored explicitly, while DIA store the column implicitly, generally DIA will

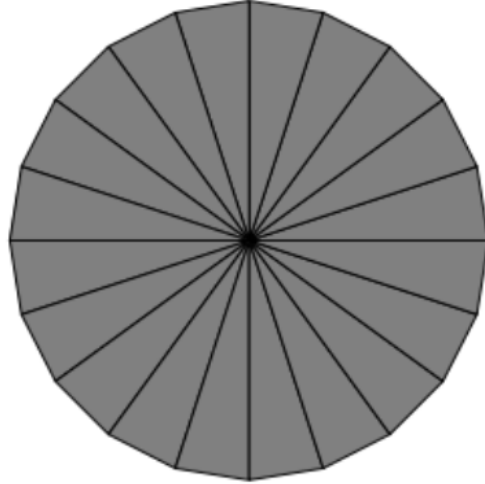


Figure 3.9: Example of the matrix that is not suitable for ELL format [17]

perform better on matrices that have some diagonal trend. Also the memory access of the  $x$  vector is not always referring to contiguous addresses (3.11).

### 3.1.5 Hybrid Format (HYB)

Hybrid format is combining the format of ELL and format of COO. As mentioned in section 3.1.4, ELL is not suitable if the maximum number non-zero in a row is very different compared to the average non-zero number in a row. To overcome this problem, hybrid scheme is storing the non-zero to the left and the zeros to the right like ELL storage scheme and then truncating the number of non-zero in the row that has maximum number of non-zero. The new maximum number of non-zero in the row decreases and approaches the average number of non-zero in a row for other rows. This truncated non-zero elements in the matrix will be stored as COO storage format.





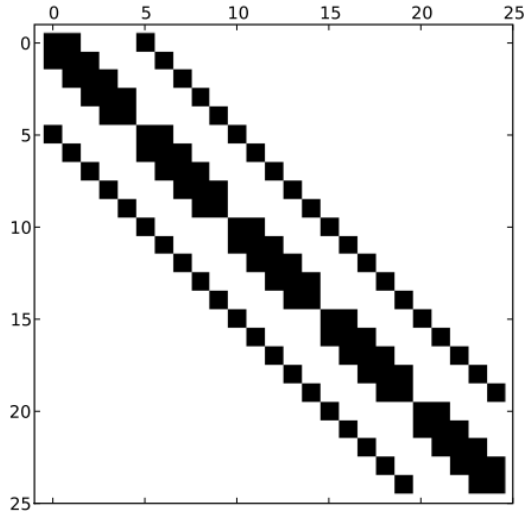


Figure 3.12: Five diagonals sparse matrix [17]

### 3.1.6 Diagonal Format (DIA)

Diagonal format is not suitable for general sparse matrix type and it is even more specific format than ELL. But when the non-zero elements are highly concentrated and restricted in the diagonal of the sparse matrix, this storage scheme could give the best performance because this format is very efficient in term of data storage and computation.

Figure 3.12 shows a band diagonal sparse matrix that is suitable with this format and figure 3.14 is not suitable. There are two arrays that represents this storage scheme. We need to store the data values of the non-zero elements and the offsets to determine the row and column position of the non-zero element. The main diagonal in the matrix represented by 0 in the offset, where the upper diagonal will be represented by positive value ( $i > 0$ ) and lower diagonal will be represented with negative value ( $i < 0$ ) compared to diagonal. The value is determined by how far the distance from the main diagonal. To understand more about the

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

$$\mathbf{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathbf{offsets} = [-2 \quad 0 \quad 1]$$

Figure 3.13: DIA representation of A in arrays data and offsets [17]

storage scheme of this format, Figure 3.13 illustrate the implementation of this scheme. From the Figure 3.13, we see that the matrix A has non-zeros in three diagonal places. In the main diagonal, upper diagonal with one distance, and lower diagonal with two distances. So the data will have three columns to represent each diagonal, and there will be an offset array that represent the diagonal position of each data column. In term of storage scheme, if the matrix has a diagonal type the storage requirement for this format will be the least compared to other formats.

This storage format has some advantages. This format stores the data, x, and y, in contiguous way so that the memory access will be efficient. Furthermore, unlike the COO formats, the row and the column of this format stored implicitly so that it will reduce the memory occupation and also reduce memory access when SpMV is being computed.

Figure 3.16 shows how SpMV in DIA storage format is being computed. Parallelizing the SpMV in DIA storage format can be done by assigning a thread to every result. The thread will compute the dot product along the non zero on

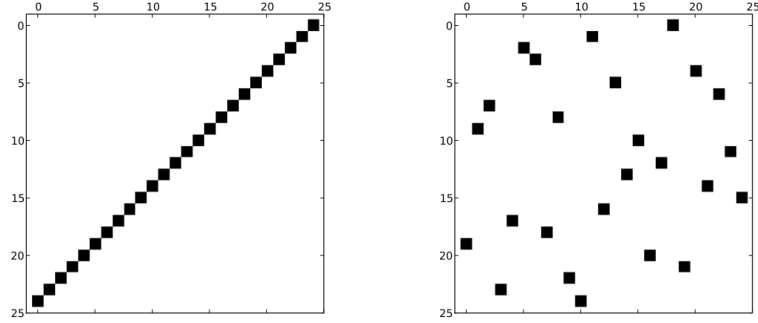


Figure 3.14: Sparse matrix type that not suited well with DIA format [17]

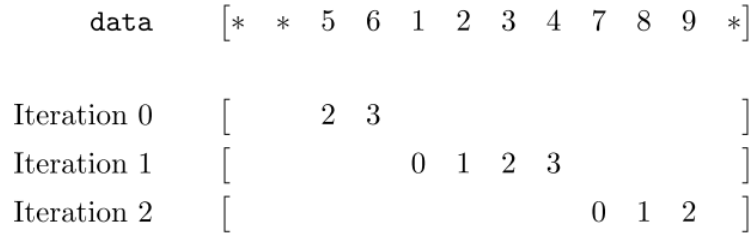


Figure 3.15: DIA SpMV memory access pattern and data array linearization [17]

the same row and compute the column index the non-zero value is placed using the offset. The result  $y$  of each row is the accumulation of the value of non-zero in the data times the value of  $x$  vector in the certain column. The best practice on storing the data is by using column major so the access of thread to the data will be coalesced. And also, each thread access to every diagonal will leads to the contiguous access to the vector  $x$  because in the same diagonal the column is contiguous (figure 3.15).

## 3.2 Previous Work Studies

Bell and Garland paper [18] first presented a comprehensive analysis of performance for the SpMV on General Purpose GPUs (GPGPUs) using different

```

__global__ void
spmv_dia_kernel(const int num_rows,
                const int num_cols,
                const int num_diags,
                const int * offsets,
                const float * data,
                const float * x,
                float * y)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    if(row < num_rows){
        float dot = 0;

        for(int n = 0; n < num_diags; n++){
            int col = row + offsets[n];
            float val = data[num_rows * n + row];

            if(col >= 0 && col < num_cols)
                dot += val * x[col];
        }

        y[row] += dot;
    }
}

```

Figure 3.16: DIA sparse matrix format SpMV kernel [17]

storage formats. It has been shown that the main bottleneck of the COO and CSR is memory bandwidth due to the explicit storage of the row and column indices. So the computation to communication ratio is low due to frequent access to the GPU global memory. Unlike COO and CSR, DIA and ELL formats store both row and columns indices implicitly. This paper also concluded that DIA is most appropriate for the structured matrices especially when non-zero values are located into a small number of matrix diagonals. They have showed that for unstructured matrices, the CSR or HYB schemes produce the best performance. However, the DIA storage format produces the best performance for the structured matrices.

Using the libraries or basic routines is an excellent approach to support programming on the GPU to compute many scientific simulation operations and practical applications accelerated via GPU. In this direction, a large number

of routines related to matrix computation are provided by NVIDIA such as CUSPARSE [19] and Cusp [20]. CUSPARSE firstly introduced in 2010 [21] by NVIDIA. This library supports Level-1, Level-2, Level-3 operations and also Sparse Matrix Format conversion. Sparse matrix storage format that supported by this library is COO, CSR, CSC, ELL, HYB, BSR, and BSRX. In CUSPARSE, level-1 operation means vector-vector operation, level-2 is sparse matrix-vector operation, and level-3 is sparse matrix-matrix operation.

Paper [22] proposed a static method to predict the optimal storage format depending on the input matrix and time needed to communicate and transfer data between CPU and GPU. They claim that the overhead time needed for the prediction is very small compared to overall time of SpMV execution time. All analyzed matrices are square for simplicity. The capability of bus between CPU and GPU and data size of an input matrix are used to measure the communication cost. The communication time is defined as the ratio of total data size for the chosen format to the effective bandwidth of bus used between CPU and GPU. In that paper, the proposed algorithm tested on number of matrices presented in Sparse Matrix Collection of Florida University.

In addition to communication time used as a factor in a prediction model, pre-processing associated with the format conversion is also used to choose the optimal sparse format and this factor is used when the format is not decided based on the previous factor. Pre-processing overhead comes from three parameters which are number of used data Structures, number of iterations, and the non-zero

elements or number of rows in the representation. This paper concluded that, CSR format is recommended for non-square matrices and dense matrices. If the number of columns is more than number of rows, then it is better to use CSR, otherwise ELL is recommended. The matrix is categorized into highly sparse or densely sparse types for square sparse matrices, based on number of the rows and the average number of non-zero elements in a row. In each category, the matrix is further analyzed with respect to the non-zero element distribution based on maximum number non-zero elements per row and average number of non-zero elements in the rows.

Further, paper [22] proposed a Bit Level Single Index (BLSI) representation to reduce the memory foot print and pre-processing overhead. This procedure is used if the prediction model mentioned above selects the HYB or ELL as an optimal storage format and that because these two formats have high pre-processing overhead. In this case the input matrix is converted into BLSI representation. A BLSI format is not used to represent the matrix but used for indices to reduce the pre-processing overhead. As a summery, this paper don't consider the structure of non-zero elements in the matrix. It carry out a test on a set of specific matrices and suggest the optimal storage format to be used.

The Authors in paper [23] proposed a method to increase the SpMV kernel performance by studying some quantities values of the input matrix such as number of non-zero elements deviation in each row, number of non-zero elements average in each row, etc. They improve the performance by balancing the load for

every thread, reducing irregular memory access of the vector, coalescing memory access, reducing the overhead of loading matrix element, and concerning about the data structure. From the observation, they conclude that CSR and ELL format have low computational load. The memory access of both formats is also coalesced. From that observation, they decided to combine both CSR and ELL formats and proposed some changes in HYB by storing some rows in CSR and the remaining rows in ELL.

To address the issue of load balancing when using this combination, multiple warps are assigned to a row that depends on the number of non-zero in that row. In their proposed method, the threshold value is found empirically because the CSR format is used for rows that have more nonzero elements than the threshold and for the remaining rows the ELL format is used. Also, the thresholds on the minimum and maximum number of nonzero elements are evaluated for warps assigned to the CSR and ELL formats, respectively. Multiple warps are assigned to a row stored in both formats. Their storage format improves the performance by a factor of 25% on average compared with the best results of HYB that was previously proposed by Bell and Garland.

The experiments performed on a set of highly unstructured sparse matrices and also on standard dataset contains 14 sparse matrices from the work of Williams et. al. [24] and these matrices have varying degree of sparsity. Some matrices have a structure of a dense matrix, some highly unstructured sub-matrices, and others with few non-zero elements per row. They compared their results with the

best performance of CSR and HYP formats presented in the work of [25].

Paper [26] proposed a new sparse matrix storage format named Compress Sparse Row with Non-zero Size (CSRNS) and According to the new format, the authors designed a Hybrid Processing Method for SpMV on GPU and at the end the performance of the new method is compared with the traditional method. The storage format proposed in this paper is updated version of traditional CSR format. The basic idea behind the proposed method is to sort the sizes of rows in a roughly descendant order. An another two-dimensional array is used in addition to those used in CSR format and this array stores the size of non-zero elements in each row and its corresponding row index.

The Authors proposed Hybrid Processing Method (HPM) corresponding to CSRNS format. HPM starts by sorting the sizes of rows with a descendant order for the whole sparse matrix. It is claimed that this classification and the way threads are assigned to the rows contribute in solving the load balancing problem which CRS suffers from. The performance of the proposed format and its corresponding hybrid SpMV kernel is compared with that of used scalar CSR and vector CSR formats proposed in the art of the literature. It is clear from the results that the proposed method outperforms both of the traditional methods.

Paper [2] proposed the Vector CSR storage format for SpMV kernel based on Bell & Garland kernel [17] using Alinea library. They used several types of matrices from the University of Florida repository [15] to compare the performance of their kernel with kernels from CUSPARSE and Cusp libraries. For the solution of sparse



linear systems with non-symmetric matrices, they consider transpose-free Quasi Minimal Residual (P-tfQMR), Bi-Conjugate Residual (P-BiCGCR), Generalized Conjugate Residual (P-GCR), Stabilized BiConjugate Gradient (BiCGStab), and Stabilized BiConjugate Gradient (L) (BiCGStabl) as the solution for sparse linear system with non-symmetric matrices and for symmetric positive definite matrices using Conjugate Gradient. For vector addition and multiplication, CUDA library CUBLAS (CUDA Basic Linear Algebra Solver) is being used. Furthermore, for dot product and norm, to make the computation effective, optimizing dot product is important by dividing the operation into two parts. First part is by multiplying each vector element, and second part is computing the summation of each vector to get the final result. For Sparse matrix-vector multiplication (SpMV) several storage format is being used, such as CSR, COO, HYB, and ELL. They found that their implementation outperforms CUSPARSE and Cusp libraries for double precision computations.

## CHAPTER 4

# LITERATURE REVIEW

The graphics processing units GPUs offer a modern parallel computing alternative to solve computational problems that have abundant parallelism. The performance and capabilities of GPUs is increasing significantly in the past decade due to the effectiveness and powerfulness of this device. The GPU is not only powerful but also has massive arithmetic hardware and large memory bandwidth which overwhelm the host CPU in some cases. Because of this superiority, the research community has been focused on the use of GPUs or accelerators as to boost performance of traditional microprocessors or multi-cores in high-performance computing system such as clusters and grids [6].

Initially GPUs were intended to do processing for display in computer video card but now they appear to be among the most robust compute units for high performance computing. In comparison to multicore CPU architectures that has only has scant number of cores, GPU architectures has many core with hundreds of core capable of running thousands of threads in parallel [27]. But the problem

on early stages of GPGPU computing model (general purpose graphics processing unit) is the difficulty of parallel programming and associated optimizations [8].

Latest generation of GPU architecture provides easier programmability and increased generality while keeping the large memory bandwidth and computational power compared to earlier GPU architectures. The computational capabilities come together with the programming complexity, diversity of optimizations, and the difficulty of making efficient use of the resource. The GPU uses a massive multithreading that utilizes a huge amount of cores. The global memory latency hiding hold the key point on increasing the performance. Ryoo et al. reported that to efficiently use a huge amount of cores and hide the global memory latency, balancing resources on solving computational problem is a necessary. By managing a large number of registers, amount of on-chip memory used per thread, number of threads per multiprocessor, and global memory bandwidth, speedup of their kernel codes could reach between  $10.5\times$  to  $457\times$  and  $1.16\times$  to  $431\times$  for their total application [28].

Since NVIDIA released CUDA in 2007, developers have rapidly developed scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models. These applications scale transparently to hundreds of processor cores and thousands of concurrent thread [8]. But unfortunately, CUDA is still not supported for mobile GPU application on smartphone [29].

## 4.1 CUDA Application

### 4.1.1 Ocean Modelling

The Regional Ocean Modeling System (ROMS) is a primitive equations ocean, terrain-following, free-surface model for many kinds of application that widely used by the scientific community ([www.myroms.org](http://www.myroms.org)). This ocean modelling simulation take a very long time to complete the computation from hours to days because of the nature from the software is compute-intensive. Therefore, the performance limitations of modern computing hardware constrain the size and resolution of simulation. The existing ROMS code could be run in parallel with either MPI or OpenMP to address these issues. This paper using CUDA Fortran to implement a new parallelization of ROMS on a graphics processing unit (GPU). This work could gain a better performance for a less power and lower cost by exploiting the massive parallelism offered by modern. To test the implementation, real data collected from coastal waters near central California for benchmarking with idealistic marine conditions has been done. This implementation gains speedup 2.5x over an OpenMP and 8x over a serial implementations and demonstrating comparable performance to a MPI implementation with much less device cost [30].

### 4.1.2 Artificial Intelligence

With the new paradigm and evolution of the architecture of Graphics Processing Units (GPUs), it has computational-power that surpassed the performance of

CPUs (Central Processing Units). Machine Learning (ML) algorithm has a very intensive computational problem that suit-well with GPUs that inherent high parallelism as a device. However, some of the currently available ML algorithms that have been implemented on the GPU are not shared openly. This create difficulties for engineers and researchers to develop a better ML algorithm on GPU. To tackle this issue, creation of an open source GPU Machine Learning Library (GPUMLib) is proposed to provide the building blocks for the development of efficient GPU ML software. Experimental results show that the algorithms on benchmark datasets could implemented yield significant time savings over the CPU counterparts [31].

### **4.1.3 Oil and Gas**

#### **4.1.3.1 Reservoir Data Visualization**

In oil and gas exploration, processing large set of data from the field and visualizing it, is very important task to do. By doing those processes and visualizing the simulation results, the data become useful. Paper [32] is describing how the visualization tool on a multi-phase 3D oil-water reservoir is being developed and parallelized with CUDA enabled GPU on IBM Cell computer. Ertekin et al. [33] described an independent 2-Phase oil reservoir simulator, in which the grid state is characterized by the oil/water saturation and pressure values over a period of time. This oil reservoir simulator displayed the data grids in 3D and the user could interact with it. This interaction needs a HPC to accelerate the simulation. The

author proposed to parallelize the computation including the user interactions with the data grids such as zooming, transformation [34], compute intensive lighting [35] and also camera movement [36]. IBM Cell SDK 3.0 and NVIDIA CUDA along with OpenGL libraries and QT are being used as the environment of this implementation. The result shows that this proposed implementation could speedup 67x over serial implementation.

#### **4.1.3.2 Reservoir Simulation**

Paper [37] proposed a novel poly-algorithmic solver for solving large sparse linear systems with multicore CPU and GPU. This algorithm is implemented for realistic compositional and black oil flow scenarios. This approach mainly uses multicore CPU computing to exploit the functional parallelism in reduction operations, sparse algebra, algebraic multigrid preconditioning, reordering, and system partitioning in order to decrease the number of GCR (General Conjugate Residual) iterations while accelerating it. On the other hand, the author also exploited the data parallelism using GPU in multi-coloring SSOR algorithm with preconditioning option such as BILUT and BILU(k). In addition, the basic linear algebra solver (BLAS) kernel is used by reducing the memory overhead per floating point operations while simultaneously deploying thousands of threads. The result shows that this method could achieve speedup 2x of overall simulation time over conventional multicore CPU implementation. This shows that many core solver has the potential to accelerate the reservoir simulation operation and computation.

### 4.1.3.3 Iterative Linear Algebra Solver

Paper [38] is purposed to show the difficulties encountered and advantages on developing sparse linear algebra computation in GPU. This paper is an overview of early experience in developing a high-performance iterative linear solver accelerated by GPU processors. Many techniques are discussed in order to find suitable methods for preconditioning and to speedup kernel for sparse matrix vector product (SpMV). The experiments is performed by using NVIDIA TESLA M2070. The result show that GPU could achieve speedup up to 8 factor over Intel MKL on the host Intel Xeon X5675 Processor. The performance overall for Incomplete Cholesky (IC) factorization preconditioned CG method using GPU accelerated can outperform its CPU by a smaller factor, up to 3, and on incomplete LU (ILU) factorization preconditioned GMRES method with GPU-accelerated can reach a speedup nearing 4. However, with better suited preconditioning techniques for GPUs, this performance can be further improved.

### 4.1.4 Math Compiler

Bergstra et. al., [39] describing how to use Theano (<https://github.com/Theano/Theano>), explains its overall design, provides benchmarks on both CPU and GPU processors, and outlines the scope of the compiler. Theano is a compiler that combines the speed of optimized native machine language with the convenience of NumPy's syntax in Python for mathematical expressions. While being statically typed and functional, the

user composes mathematical expressions in a high-level description that mimics NumPy's syntax and semantics. Theano optimizes the choice of expressions before performing computation. It compiles them into dynamically loaded Python modules after automatically being translated into C++ (or CUDA for GPU). The speedup gains by implementing Theano for common machine learning algorithms scaled from  $6.5\times$  and  $44\times$  faster compared to other alternatives (including machine learning that implemented with MATLAB, NumPy/SciPy, and C/C++) when compiled for the GPU and between  $1.6\times$  to  $7.5\times$  faster when compiled for the CPU.

#### 4.1.5 RSA Decryption

Paper [40] proposed an efficient parallel algorithm for RSA decryption with CUDA that use many-core GPUs. In the telecommunication, when there is an unrecognized communication is trying to connect in the network, it is necessary to use cryptography. Public-key cryptography algorithm like RSA, use  $D$  as the private key and a pair  $(N, E)$  as the public key. The  $N$  is a large number that produced by two large prime numbers  $p$  and  $q$  that are kept secret. To extract  $p$  and  $q$  from  $N$  cannot use any polynomial time algorithm and it is very hard. There are many proposed methods to factoring large numbers. Furthermore, for modern GPUs, the advantages of memory bandwidth and computing power have made porting applications on it become a very important issue. The experimental results showed that to find out the result of factoring large numbers compared to



the CPU-based algorithm, the proposed GPU-based algorithm can achieve 1197.5x average speedup within a reasonable time.

## 4.2 CUDA Multi-GPU

In CUDA, it's possible to do implementation using Multi-GPU to increase the performance of the application. Multi-GPU implementation of an application means that the application uses several GPUs instead of a single GPU to further speedup the computation. By having multiple GPUs per node, it means improving performance/Watt that amortize the CPU server cost among more GPUs. Similar effects are observed for the price [41]. Micikevicius et al. [42] implemented a 3D Finite Difference Computation on single GPU and Multi-GPU using CUDA, and showed for single Tesla 10-series GPU a better performance than 4-core Harpertown CPU using similar code from seismic industry. On Multi-GPU implementation, it shows that the performance for the above application achieves linear speedup by overlapping inter-GPU communication with computation either for 2 GPUs or 4 GPUs. Asynchronous communication and computation is used in this application along with the page-locked memory in the CPU part. The memory exchange has been done in this application by carrying the data from the GPU to CPU, and then from CPU to CPU, and return it back from CPU to GPU to continue the process.

Xu et al. [43] is implementing Multi-GPU of bottom-up attention selection using CUDA that used for autonomous switching on mobile robot vision and achieving

a decent scaling for two, three, and four GPUs over one GPU. The CUDA implementation is not described much on this work, but they said Multi-GPU performance strongly depends on the efficient usage of thread-block organization and different memories. Because the application is using mobile robot and vehicle, the main drawbacks of using Multi-GPU is the power demand for Multi-GPU.

On novel implementation in cellular genetic algorithm with Multi-GPU implementation by Vidal et al. [44], the performance of single GPU is better than Multi-GPU. It's happened because the communication only enabled when the current GPU kernel that executes on both GPUs finished and then the data sent to the host, then it could continue with the other GPUs after getting the new data serially. Schaetz et al. [13] presenting Multi-GPU programming library called MGPU for real-time applications by using CUDA as the backend. For the testing application, they use CUFFT library to calculate the Fourier transforms and CUBLAS library to calculate scalar products. The result shows that the Multi-GPU achieving 2.1 speedup for 4 GPUs over single GPU and 1.7 speedup for 2 GPUs over single GPU. Referring to the power drain and energy consumption, 2 GPUs show the most efficient energy consumption while 4 GPUs do not consume significantly more energy compared to single GPU for energy consumed per frame. Thibault et al. [45] implementing an incompressible flow Navier-Stokes solver for Multi-GPU workstation platforms. In this application, the communication has been done synchronously by using single `cudaMemcpy()`. Many small memory

copy operations are not advisable according to CUDA best practices. However, this application is not communication extensive, so on 4 GPUs it stills achieve speedup of 3 over single GPU and 2 GPUS achieve speedup 1.6 times over single GPU. Organizing data for shared memory and global memory gives better results than storing the data in all-global memory or all-shared memory.

Simulating Multilevel Fast Multipole Algorithm (MLFMA) Guan et al. [46] has been proposed using hybrid OpenMP-CUDA programming model with Multi-GPU. This algorithm is calculating radiation patterns of the basis functions ( $V_s$ ) and receiving patterns of testing functions ( $V_f$ ), translator ( $T$ ), and the assembly of the near-field system matrix ( $Z_{near}$ ) before invocation of an iterative solver using BiCGStab to calculate far-field interaction. COO (coordinates list) storage scheme is used for the block matrices to reduce the storage and to achieve coalesced memory access. The idea of this implementation is by using 4 core CPU using OpenMP that mapped each core to each GPU using CUDA. To calculate the sparse matrix-vector multiplication for BiCGStab, this implementation uses CUDA library CUSPARSE. For memory management, the authors use two different strategies. First the use of the pinned memory strategy, and the second is based on using the global memory strategy. For calculating  $V_s$  and  $V_f$ ,  $T$ ,  $Z_{near}$ , the same parallelization scheme is applied which leads to the same speed-ups for pinned memory and global memory strategies. The assembly of the near-field system matrix ( $Z_{near}$ ), the speed-up of OpenMP-CUDA-MLFMA (Multilevel Fast Multipole Algorithm) is 60 times faster than CPU-MLFMA . For BiCGStab as

the last part of the algorithm, the speedup of OpenMP-CUDA-MLFMA using pinned memory is 4.2 faster than CPU-MLFMA, and 16 times speedup is achieved by using global memory strategy. The global memory strategy shows better result over pinned memory strategy because the host-device communication on pinned memory strategy degrade the performance of overall operation including BiCGStab, while global memory strategy does not need to do any host-device communication.

Boyer et al. [47] presenting a Dense Dynamic Programming on Multi-GPU or knapsack problems. The parallelization has been done by dividing the loop that process the value of  $f(c)$  on solving Knapsack Problems (KP) as this operation consumes the major part of processing time. Furthermore, the computation is partitioned so that each GPU computes a subset of values of  $f(c)$ . The result shows that single GPU could achieve 14.7 times speedup over CPU and 2 GPUs achieving about 28 times speedup over CPU.

Sourouri et al. [48] compared the state of the art implementation of 3D stencil using MPI with combination of OpenMP and CUDA. Overlapping communication with computation is the main focused on this work. Three OpenMP threads are spawned for every GPU to handle processing, incoming, and outgoing buffers. Two CUDA streams on every GPU are created to overlap communication with computation. The implementation has been done by using Fermi architecture Tesla C2050 and Kepler architecture Tesla K20. The result shows that OpenMP-CUDA outperform MPI with 1.85x faster in 4 GPUs implementation

on Fermi architecture, but for 2 GPUs, the difference is not really noticed. Also in Kepler architecture, the result of OpenMP-CUDA approach is only slightly increasing.

Ren et al. [49] investigated the power consumption for multi core and GPUs processing in large scale SIMD computation with CUDA. It is concluded that to get the best efficiency of power consumption, it is better to balance the workload between CPU cores and GPUs. Other optimizations are proposed to reduce the power consumption using the CPU instead of GPU for small workloads because CUDA GPUs initialization has significant overhead that consumes more power.

An implementation of an iterative tomographic reconstruction algorithm on Multi-GPU has been done by Jang et al. [50]. The background of this work is the need of computational demands for this algorithm. First task is to differentiate which work should be done in CPU and which work should be done in GPU. Thread mapping takes an important role on this implementation to make sure the threads have balanced workload. On Multi-GPU implementation, the same number of CPU threads is created as much as the number of GPUs to be utilized during GPU operations like kernel invocation and data copying to minimize the overhead. The result shows that speedup of 4 GPUs over single GPU is about 2.1 for the Forward Projection.

Implementation of the simplex algorithm using Multi-GPU has been done by Lalami et al. [51]. Simplex algorithm is one of popular method to solve linear programming (LP) problems. This work is using Quad-Core Intel Xeon E5640

and NVIDIA Tesla C2050 for the GPUs. There are two kinds of data in this algorithm, first is shared data, and second is local data. The shared data is stored in page-locked host memory and the local data is stored in global memory of GPUs. This work achieves maximum speedup of 24.5 with two Tesla C2050 over CPU, and 1.93 speedup over single GPU.

Paper [52] proposed a Multi-GPU and Multi-CPU parallelization for interactive physics simulation. Author stated that CPUs is mostly working better on small workload than GPUs while GPUs is better on the large workloads. Because of this, the author dividing the task based on the weight of the workload by looking at the execution time and use the appropriate processing unit (PU) to do the job. The experiments show that more than 400 thousands finite element methods (FEM) and 64 colliding objects in the complex simulation could be done in a time of 0.082s on every iteration for 8 GPUs instead of 3.82s on single CPU. On heterogeneous computation that contains complex and simple objects, the tasks could be separated to exploit all of the resources between 8 CPU cores and 4 GPUs to make the job done efficiently. The result shows that with 4 CPU cores manage the GPUs and other 4 CPU cores done individual tasks, the performance is increasing by 30% because the CPUs unload GPUs task on small workload and makes the job done more efficiently. The author developed a mechanism that works like the Distributed Shared Memory (DSM) to ease the programmer work on transferring data between CPU and GPU also between GPUs. To minimize communication between CPUs and GPUs, the author use METIS or SCOTCH by

re-computing the mapping of each partition every time the task graph changing, which indicates that the partitioning is adaptive. For the objects that are not colliding, the speedup could reach 7 with 8 GPUs while the objects that collide the efficiency decreased but still get about 50% efficiency. This loss of efficiency is due to the need to adapt ever time the collide is happening so there will be an overhead for GPU-CPU-GPU communication.

### 4.3 CUDA BiCGStab

Ament et al. [53] presented a parallel preconditioned conjugate gradient solver [54] [55] for the simulating the Poisson's Equation on a Multi-GPU platform. The author stated that for Multi-GPU platform, the bandwidth limitations and the implied communications in computing sparse linear systems could cause poor performance or even negative speedups at least for a small problem size. Therefore, this approach is questionable to become a solution. Conjugate gradient solver is well known iterative algorithm that rapidly converges when the solver matrix is symmetric and positive definite. With a proper implementation of preconditioning for conjugate gradient (PCG), the convergence is greatly enhanced. A strategy to improve the problem of efficient preconditioning has been proposed. This strategy leads to a reasonable speedup. The preconditioners like Jacobi are easy to parallelize but only have minor impact on the speed of convergence. The incomplete Cholesky factorization (IC) or symmetric successive over-relaxation (SSOR) have a major impact on the speed of convergence but are hard to

parallelize due to their sequential nature. The problem of this implementation is when adding more GPUs, sooner or later the host capabilities will be exhausted. For the small problem the cost of communication is relatively high which leads to the poor performance. The result shows that the speedups of 2.02 for 4 GPUs over single GPU.

Paper [56] proposed to accelerate the well-known Strassen Algorithm and Conjugate Gradient by using CUDA and MPI. By examining the application, the authors noted that the implementation with normal MPI give better performance over CUDA+MPI because of the lack of second level parallelism in the Conjugate Gradient method. The latency between the GPU memory and the host memory represent a huge disadvantage of CUDA+MPI implementation.

Cevahir et al. [57] presented a fast conjugate gradients with multiple GPUs. In this implementation, the authors used BCSR (Blocked Compressed Sparse Row) storage scheme. Multiple GPUs is useful not only to accelerate the application simulation time and enhance its performance, but also to overcome the memory bottleneck. To provide a load balanced execution between GPUs, the proposed approach lead to distributing the same amount of non-zeros to each GPU on matrix vector multiplication. CPU and GPUs are communicating when exchanging input vector for matrix-vector multiplication in conjugate gradient algorithm. The global synchronization is occurring before SpMV by exchanging the input vector for matrix vector multiplication, and twice when computing the scalars. The result shows that the speedup of 2 GPUs over single GPU is 1.73



and 4 GPUs is 2.83.

## CHAPTER 5

# METHODOLOGY

### 5.1 General Hepta BDIA (BDIA-GH)

The basis of this work is the need of design for a customized storage scheme that will be suitable for the solver matrices encountered in reservoir simulation [58]. This General Hepta matrix is meant to represent the interaction among the grid cells for a Reservoir Simulation (RS) using a Black Oil system (BO). There are several parameters for characterizing the solver matrix in RS:

1.  $N_c$ : Number of components used in reservoir simulation in Black Oil system.
2. Grid defined using  $(J, H, \text{ and } I)$  which consist of 3D grid with dimension  $J \times H \times I$  and so that each grid cell has  $N_c$  independent variables.
  - (a) The definition of the order for the X,Y, and Z directions  $(J, H, \text{ and } I)$ .
  - (b) Unfolding the above 3D grid space into 1D array consists of converting

the each cell address  $(j, h, i)$  into its index in 1D vector as

$$m = (j, h, i) = j + hJ + iJH$$

- (c) The stencil in the 3D space define the immediate neighbours to each grid cell located at  $(j, h, i)$  are  $(m - 1)$ ,  $(m + 1)$ ,  $(m - J)$ ,  $(m + J)$ ,  $(m - JH)$  and  $(m + JH)$  representing the cells located at the left, right, down, up, backward, and forward to the central cell  $m = (j, h, i)$ .
- (d) Each grid cell is associated with a block of  $N_c \times N_c$  in the GH matrix as we have  $N_c$  components for each grid cells.
- (e) Assume a black oil reservoir with k-phase, each block in the *block* $\times$ *block* section of the Jacobian or GH matrix is denoted as 'square' which a square matrix of size  $N_c \times N_c$ , i.e  $N_c$  residuals.

Suppose a grid  $(J = 2, H = 4, I = 2)$ . A grid point

$$m = (j, h, i) = j + hJ + iJH \tag{5.1}$$

In the *block* $\times$ *block* section of Jacobian, the 6 points stencil lead to the distribution rows as in figure 5.1

From this General Hepta matrix (figure 5.2), we are supposed to develop a storage

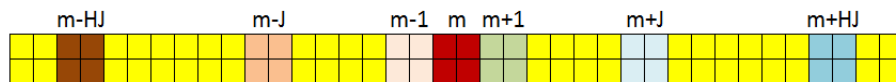


Figure 5.1: Example of General Hepta (GH) row

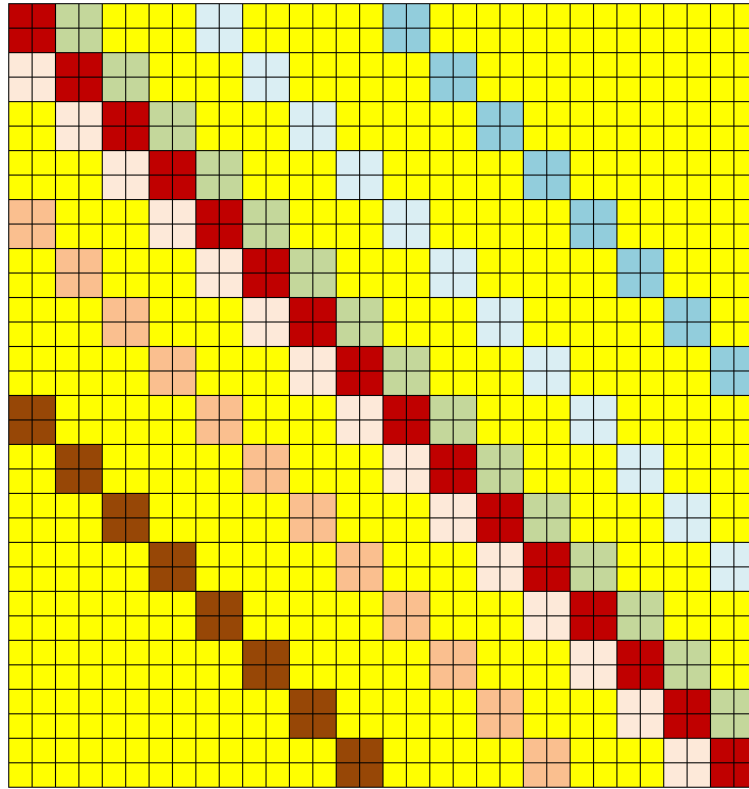


Figure 5.2: Example of GH matrix with parameter  $J=2$ ,  $H=4$ ,  $I=2$ ,  $N_c=2$

format that is suitable with this sparse matrix. From the chapter 3, we could see that this sparse matrix type is diagonal matrix and it is very structured. For this kind of problem, the well-suited solution is by using DIA storage format or ELL storage format.

Figure 5.3 shows that ELL has data values smaller than DIA. But ELL need twice of the storage to store the column indices while DIA is only need additional offset as much as the number diagonal as shown in figure 5.4.

From this observation, we cannot decide which one of these storage formats represents the best solution, because each of them has its own advantage and drawbacks. For ELL, the main drawback is the need of column indices that will lead to more storage needed and more memory access, while the DIA format

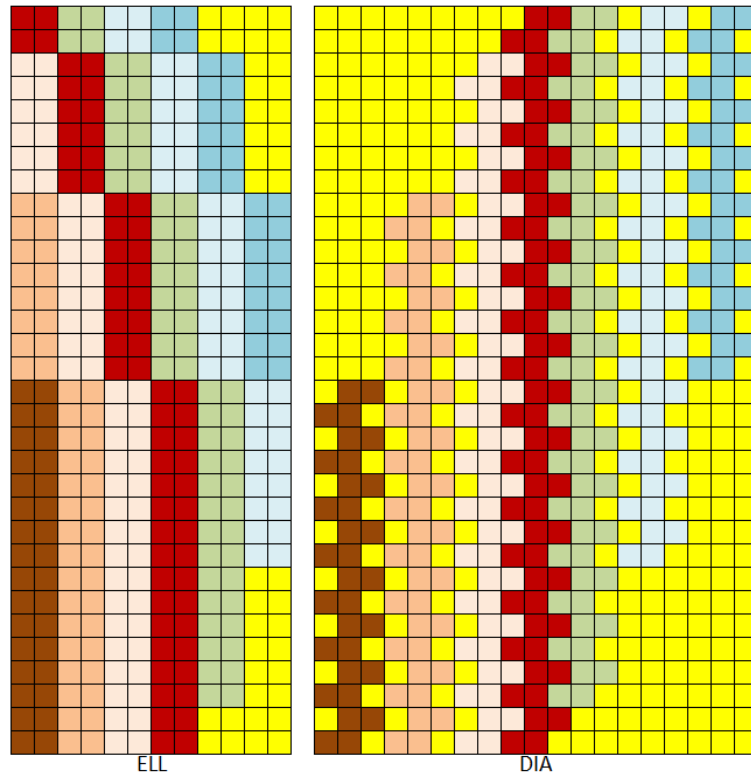


Figure 5.3: Comparison between ELL and DIA storage format for GH

only needs a small offset size to calculate the column indexing. For DIA, the main disadvantage over ELL is the data value that will storing many zeros and it could become worse when the block size ( $N_c$ ) is larger (figure 5.6). Because of that, we proposed a solution on presenting the General Hepta matrix called Block Diagonal-General Hepta format (BDIA-GH). This format is similar to DIA format but it is customized to the above pattern with a General Hepta matrix because it could adapt with the changing size of block size, so there will be no zeros in the middle of data value.

If we see this proposed storage format, this format is as compact as ELL but without the need of column index vector, it only needs an offset vector just like DIA.

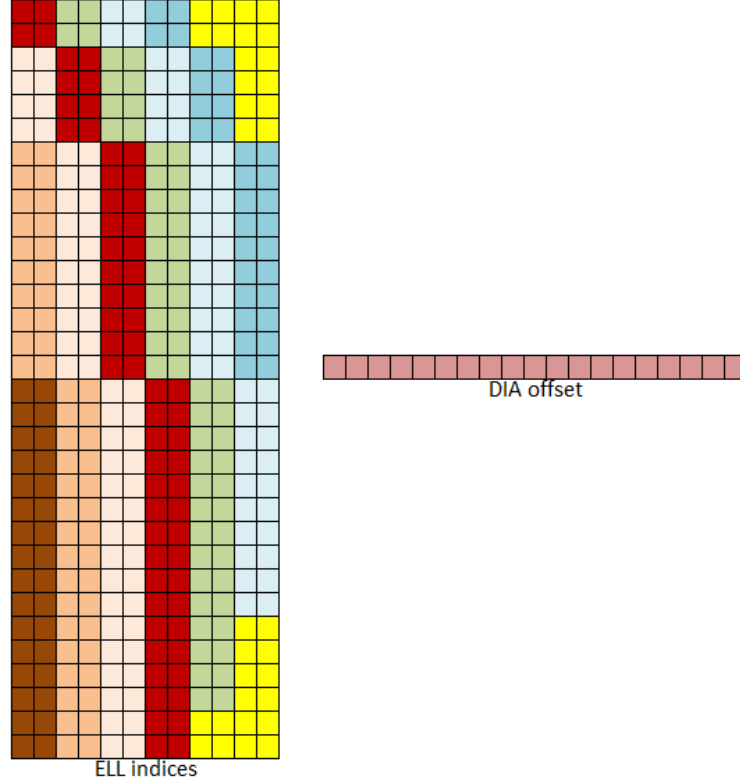


Figure 5.4: Comparison between ELL indices and DIA offset

To work on this format, we have to build a program to generate BDIA-GH matrix and store it as a matrix market file that is stored in COO format so it could be used by CUDA libraries CUBLAS and CUSPARSE.

Matrix Market I/O library is used to export the generated BDIA-GH matrix. As the preparation on generating BDIA-GH matrix, memory should be allocated as much as BDIA-GH matrix needed. The memory needed for the BDIA-GH is

$$mem_{size} = n_{dia} \times N \times float_{size} \quad (5.2)$$

$$n_{dia} = 7 \times N_c \quad (5.3)$$

$$N = I \times J \times H \times N_c \quad (5.4)$$

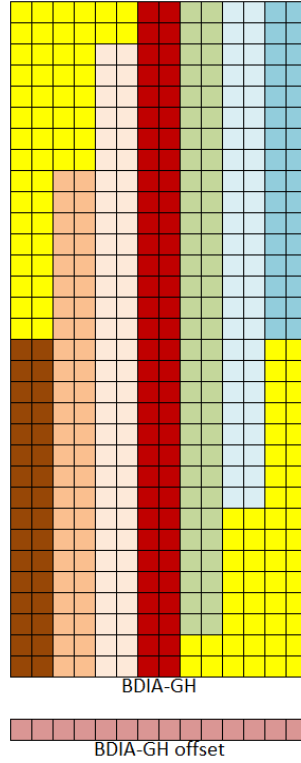


Figure 5.5: Proposed BDIA-GH storage format that contains two array: data and offset

The offsets memory also has to be allocated by

$$offset_{size} = n_{dia} \times N_c \times float_{size} \quad (5.5)$$

Unlike DIA format that has a straight forward implementation for offset by putting the distance between the main diagonal and desired diagonal with positive value on upper diagonal and negative value on lower diagonal, determining the offset value of BDIA-GH is a little bit tricky. Because of the block characteristic of BDIA-GH, in the same column of BDIA-GH doesn't always have the same diagonal distance with the main diagonal (figure 5.6).

So we decided to use the top row of the block (figure 5.6) as the main offset and

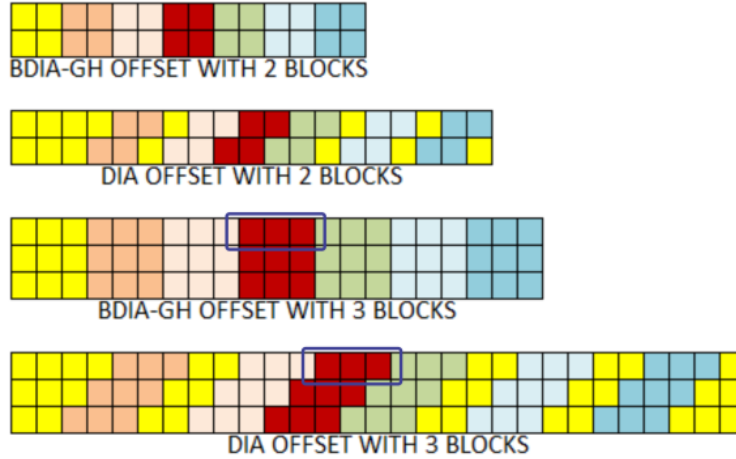


Figure 5.6: Comparison between BDIA-GH and DIA in one row of block

later in kernel, the calculation of the column indices will be different with DIA format (algorithm 2).

There are three routines that are used for generating BDIA-GH matrix. The first is `generalHeptaBdia()` (algorithm 3) that moves across the row block and calls `fillRowBdia()` (algorithm 4) routine to fill every row with blocks by calling `fillBlock()` (algorithm 5) routine to fill the block in every point. This BDIA-GH format is in column major order.

To generate matrix market file, there is additional function that translates the BDIA-GH formats into COO format (algorithm 6). As mentioned in chapter 3, COO format has three arrays to store value, column indices, and row indices. These array then become the input for matrix market I/O library to generate matrix market file (algorithm 7).



---

**Algorithm 2** Generate Offset algorithm

---

```
1: for  $r = 0$  to 6 do
2:   for  $s = 0$  to  $N_c$  do
3:     if  $r = 0$  then
4:        $\text{offsests}[r \times N_c + s] \leftarrow s - (N_c \times j \times h)$ 
5:     else if  $r = 1$  then
6:        $\text{offsests}[r \times N_c + s] \leftarrow s - (N_c \times j \times h)$ 
7:     else if  $r = 2$  then
8:        $\text{offsests}[r \times N_c + s] \leftarrow s - N_c$ 
9:     else if  $r = 3$  then
10:       $\text{offsests}[r \times N_c + s] \leftarrow s$ 
11:    else if  $r = 4$  then
12:       $\text{offsests}[r \times N_c + s] \leftarrow s + N_c$ 
13:    else if  $r = 5$  then
14:       $\text{offsests}[r \times N_c + s] \leftarrow s + (N_c \times j)$ 
15:    else if  $r = 6$  then
16:       $\text{offsests}[r \times N_c + s] \leftarrow s + (N_c \times j \times h)$ 
17:    end if
18:  end for
19: end for
```

---

---

**Algorithm 3** General Hepta BDIA Algorithm

---

```
1: procedure GENERALHEPTABDIA
2:   for  $m_i = 0$  to  $i$  do
3:     for  $m_h = 0$  to  $h$  do
4:       for  $m_j = 0$  to  $j$  do
5:          $m \leftarrow m_i + m_h \times j + m_i \times h \times j$ 
6:          $\text{fillRowBdia}(\text{data}, m, j, h, i, N_c)$ 
7:       end for
8:     end for
9:   end for
10: end procedure
```

---

---

**Algorithm 4** Fill Row BDIA-GH Algorithm

---

```
1: procedure FILLROWBDIA
2:    $mxSize \leftarrow j \times h \times i \times N_c$ 
3:    $colSize \leftarrow 7 \times N_c$ 
4:    $blockSize \leftarrow N_c$ 
5:    $mainStart \leftarrow m \times mxSize \times k + m \times k$ 
6:    $mainStartBdia \leftarrow m \times k + mxSize \times 3 \times k$ 
7:    $//0 < value < 1$ 
8:   fillBlock(data,  $mainStartBdia$ ,  $blockSize$ ,  $mxSize$ ,  $value$ )
9:    $min \leftarrow m \times k \times mxSize$ 
10:   $max \leftarrow min + mxSize$ 
11:   $start \leftarrow mainStart - (h \times j \times k)$ 
12:   $startBdia \leftarrow mainStartBdia - (3 \times k \times mxSize)$ 
13:  if  $min < start$  then
14:    fillBlock(data,  $startBdia$ ,  $blockSize$ ,  $mxSize$ ,  $value$ )
15:  end if
16:   $start \leftarrow mainStart - j \times k$ 
17:   $startBdia \leftarrow mainStartBdia - (2 \times k \times mxSize)$ 
18:  if  $min \leq start$  then
19:    fillBlock(data,  $startBdia$ ,  $blockSize$ ,  $mxSize$ ,  $value$ )
20:  end if
21:   $start \leftarrow mainStart - 1 \times k$ 
22:   $startBdia \leftarrow mainStartBdia - (1 \times k \times mxSize)$ 
23:  if  $min \leq start$  then
24:    fillBlock(data,  $startBdia$ ,  $blockSize$ ,  $mxSize$ ,  $value$ )
25:  end if
26:   $start \leftarrow mainStart + 1 \times k$ 
27:   $startBdia \leftarrow mainStartBdia + (1 \times k \times mxSize)$ 
28:  if  $start < max$  then
29:    fillBlock(data,  $startBdia$ ,  $blockSize$ ,  $mxSize$ ,  $value$ )
30:  end if
31:   $start = mainStart + j \times k$ 
32:   $startBdia \leftarrow mainStartBdia + (2 \times k \times mxSize)$ 
33:  if  $start < max$  then
34:    fillBlock(data,  $startBdia$ ,  $blockSize$ ,  $mxSize$ ,  $value$ )
35:  end if
36:   $start \leftarrow mainStart + (h \times j \times k)$ 
37:   $startBdia \leftarrow mainStartBdia + (3 \times k \times mxSize)$ 
38:  if  $start \leftarrow max$  then
39:    fillBlock(data,  $startBdia$ ,  $blockSize$ ,  $mxSize$ ,  $value$ )
40:  end if
41: end procedure
```

---

---

**Algorithm 5** Fill Block BDIA-GH Algorithm

---

```
1: procedure FILLBLOCK
2:   for  $j = 0$  to  $blockSize$  do
3:     for  $i = 0$  to  $blockSize$  do
4:        $m \leftarrow m_i + m_h \times j + m_i \times h \times j$ 
5:        $data[start + i + j \times mxSize] = value$ 
6:     end for
7:   end for
8: end procedure
```

---

---

**Algorithm 6** BDIA-GH to COO Conversion Algorithm

---

```
1: procedure BDIATOCO
2:   for  $m_i = 0$  to  $mtxSize$  do
3:     for  $m_j = 0$  to  $num_{dia}$  do
4:        $diff \leftarrow m_i \bmod k$ 
5:        $idx \leftarrow m_i + offset[m_j] - diff$ 
6:        $val \leftarrow data[m_i \times num_{dia} + m_j]$ 
7:       if  $idx \geq 0 \wedge idx < mtxSize$  then
8:          $y[count] \leftarrow m_i$ 
9:          $x[count] \leftarrow idx$ 
10:         $data[count++] \leftarrow val$ 
11:       end if
12:     end for
13:   end for
14: end procedure
```

---

The configuration that used to generate the matrix market file is as table 5.1.

## 5.2 Sparse Matrix-Vector Multiplication for BDIA-GH

There are two ways for implementing the sparse matrix-vector multiplication kernel. First is using CUDA library CUSPARSE and the second is by calling

---

**Algorithm 7** Generate Matrix Market File Algorithm

---

```
1: procedure MMIO
2:   FILE f2 = fopen(str, "w")
3:   mmInitializeTypecode(&matcode)
4:   mmSetMatrix(&matcode)
5:   mmSetCoordinate(&matcode)
6:   mmSetReal(&matcode)
7:   mmWriteBanner(f2, matcode)
8:   mmWriteMtxCrdSize(f2, N, N, nz)
9:   // NOTE: matrix market files use 1-based indices,
10:  // i.e. first element of a vector has index 1, not 0.
11:  for i = 0 to nz do
12:    fprintf(f2, I[i] + 1, J[i] + 1, values[i])
13:  end for
14: end procedure
```

---

Table 5.1: Matrix Market File configuration

N	J	H	I	Nc	nnz
65536	16	16	32	8	3635072
131072	16	32	32	8	7272320
131072	16	16	32	16	14540288
524288	32	64	64	4	14613472
262144	16	32	32	16	29089280
524288	32	32	64	8	29224832
1048576	32	128	64	4	29228000
1048576	32	64	64	8	58453888
1048576	32	32	64	16	116899328
2097152	32	128	64	8	116912000

spMV\_bdia\_gh\_kernel() (algorithm 8). For the CUSPARSE, the application accepts input of the matrix market file that must be executed and also the block size to determine the size of the block that will be used by the BSR matrix format. After that, the file will be read and memory will be allocated to store the data from the file into the host memory in COO format. A vector X with the size of

$$V = I \times J \times H \times N_c \times float_{size} \quad (5.6)$$

will be allocated and filled randomly to be multiplied with the desired matrix.

After all of the required data are available in the host, the data are copied from host to device in desired format (i.e. CSR, BSR, and HYB). Furthermore, when all the data are ready in the device, the SpMV are executed according to the certain format (i.e. CSR, BSR, and HYB).

For the BDIA-GH format, there is no need to use the matrix market file. The application could directly input the configuration of the desired BDIA-GH matrix and after that, the memory will be allocated in the host and the BDIA-GH will be ready at the host by calling routine generalHeptaBdia() (algorithm 3). The offset vector for BDIA-GH is allocated and filled directly to the host according to the input from the user. The solution of the vector  $x$  that will be multiplied by BDIA-GH matrix is also allocated and filled in the host randomly. After all the data ready in the host, the data are copied to the device and sparse matrix vector multiplication (SpMV) is executed by calling spMV\_bdia\_gh\_kernel() (algorithm 8). Since each thread is assigned to compute each row for SpMV result, the grid

size to be executed in the kernel must equals to number of blocks, where

$$N_{blocks} = N/block_{size} \quad (5.7)$$

and

$$block_{size} = N_{threads} \quad (5.8)$$

For the optimization in the kernel algorithm 8, the offset from the global memory is copied to shared memory to make sure that the communication will be as fast as possible (algorithm 9). So in every block, the thread that has ID `threadIdx.x` less than the number of offset elements, must copy the offset element of `threadIdx.x` from the global memory to the shared memory. Furthermore, the thread will go along the row and fetch its value (step 4) and the index of the column could be fetch from offset that has already stored in shared memory (step 3) and it will be subtracted by `diff` that depends on its row position in the block `k` (step 2). Because the data is put in column major order in global memory, the access of the thread for data also for vector `x` is coalesced. To get the best execution time, we tried several block size of thread, and the result is the optimized thread number in a block for this application is 256.

The data correctness is addressed by comparing the computation from device and computation from host using `checkPracticalErrors()` function (algorithm 10).

---

**Algorithm 8** SpMV BDIA-GH Algorithm

---

```
1: procedure SPMV_BDIA_GH_KERNEL
2:    $sum \leftarrow 0$ 
3:   for  $j =$  to  $num_{dia}$  do
4:      $diff \leftarrow i \bmod k$ 
5:      $idx \leftarrow i + \mathbf{offset}[j] - diff$ 
6:      $val \leftarrow \mathbf{data}[i + j \times N]$ 
7:     if  $idx \geq 0 \wedge idx < N$  then
8:        $sum \leftarrow sum + val \times \mathbf{x}[idx]$ 
9:     end if
10:  end for
11:   $\mathbf{v}[i] \leftarrow sum$ 
12: end procedure
```

---

---

**Algorithm 9** Copy from Global Memory to Shared Memory Algorithm

---

```
1:  $\_shared\_ \mathbf{sharedOffset}$ 
2:  $i \leftarrow threadIdx.x$ 
3: if  $i < offset_{size}$  then
4:    $\mathbf{sharedOffset}_i \leftarrow \mathbf{offset}_i$ 
5: end if
6:  $\_syncthreads$ 
```

---

---

**Algorithm 10** Check Practical Error Algorithm

---

```
1: procedure CHECKPRACTICALERRORS
2:    $sum \leftarrow 0$ 
3:    $low \leftarrow \mathbf{c}_0 - \mathbf{gold}_0$ 
4:    $up \leftarrow low$ 
5:   for  $i \leftarrow 0$  to  $n$  do
6:      $temp \leftarrow \mathbf{c}_i - \mathbf{gold}_i$ 
7:      $sum \leftarrow sum + (temp < 0? -temp : temp)$ 
8:     if  $temp > up$  then
9:        $up \leftarrow temp$ 
10:    else if  $temp < low$  then
11:       $low \leftarrow temp$ 
12:    end if
13:  end for
14: end procedure
```

---

### 5.3 BiCGStab with General Hepta

Implementing BiCGStab with BDIA-GH required another library to do vector-vector operation called CUBLAS. Refer to BiCGStab algorithm in table 5.2, at the initial state all the vectors required are allocated in the host and device. Like in the SpMV, the matrix for the CUSPARSE matrix-vector operation is taken from matrix market file, while for BDIA-GH the matrix is built directly into BDIA-GH format along with the offset.

At the step 1,  $\mathbf{b}$  will be copied into  $\mathbf{r}_0$  using `copy_kernel` from CUBLAS, then SpMV will be called either using BDIA-GH or CUSPARSE (step 2). The  $\mathbf{r}_0$  will be subtracted by the result from SpMV using `axpy_kernel_val` (step 3). Furthermore,  $\mathbf{r}_0$  will be copied to the  $\hat{\mathbf{r}}_0$  (step 4). Then, we initiate  $\mathbf{v}_0$  and  $\mathbf{p}_0$  as 0 (step 5). At the step 6, there will be a *for* loop that will be executed 5 times. At the step 7, `dot_kernel` will be called to get  $\rho_i$  value.  $\mathbf{p}_i$  will be calculated in the step 8, 9, and 10 using `axpy`, `scal`, and another `axpy` kernel. The first SpMV inside loop will be at step 11. At step 12, dot product is occurred and the scalar operation will be done in the host. Step 13 and 14 will calculate  $\mathbf{s}$  by copying  $\mathbf{r}_{i-1}$  to  $\mathbf{s}$  and then it will be subtracted by  $\alpha\mathbf{v}_i$  using `axpy_kernel_val`. Second SpMV is occurred at step 15. To calculate  $\omega_i$ , `dot_kernel` called twice in step 16 and 17. Furthermore,  $\mathbf{x}_i$  will be calculated by calling `axpy_kernel_val` twice (step 18, 19) and the result will become input value for `dot_kernel` to determine whether the  $\mathbf{x}_i$  is converged or not. The last operation will be calculating  $\mathbf{r}_i$  by calling `copy_kernel` and `axpy_kernel_val`.

There are four methods (storage schemes) for computing the solution using the



BiCGStab algorithm. First is using CSR, second is using BSR, third is using HYB, and the last one is using BDIA-GH.

All of the vector operations have been done by using CUBLAS library routines. CUBLAS library routines has vector-vector product (dot), axpy, and vector-scale functions to implement the vector operations in BiCGStab Algorithm.

For the SpMV operation in BiCGStab, the output vector from CUBLAS operation is the input operand for the corresponding SpMV format (i.e. `cusparseScsrmmv()`, `cusparseSbsrmv()`, `cusparseShybmvmv()`, `spMV_bdia_gh_kernel()`).

After all the operations have been finished, all the memory is freed, and library handle are destroyed.

## 5.4 Multi-GPU BiCGStab BDIA-GH

For Multi-GPU BiCGStab implementations, we are not using any CUSPARSE library since our BDIA-GH SpMV implementation is already the best over any CUSPARSE library SpMV (see Chapter 6 Result and Discussion).

The implementation of Multi-GPU BiCGStab BDIA-GH is by dividing the workload of BiCGStab across the GPUs. We are using 2, 4, and 8 GPUs for the implementation. In the CUDA code, to use multi-GPU for the implementation is by calling every CUDA device before using the function that desired to be called (algorithm 11).

From the table 5.2, we could see that there are several reductions and global writes that must be done cooperatively by all blocks and all kernels especially

Table 5.2: BiCGStab Algorithm with Its Function Call kernels

No	BiCGStab Algorithm	Kernel Calls	Library/Function
1	$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$	copy_kernel	CUBLAS
2		spMV_bdia_gh_kernel	BDIA-GH/CUSPARSE
3		axpy_kernel_val	CUBLAS
4	$\hat{\mathbf{r}}_0 = \mathbf{r}_0$	copy_kernel	CUBLAS
5	$\mathbf{v}_0 = \mathbf{p}_0 = \mathbf{0}$	memset	CUDA
6	inside loop		
7	$\rho_i \leftarrow (\hat{\mathbf{r}}_0, \mathbf{r}_{i-1})$	dot_kernel	CUBLAS
8	$\mathbf{p}_i \leftarrow \mathbf{r}_{i-1} + \beta(\mathbf{p}_{i-1} - \omega_{i-1}\mathbf{v}_{i-1})$	axpy_kernel_val	CUBLAS
9		scal_kernel_val	CUBLAS
10		axpy_kernel_val	CUBLAS
11	$\mathbf{v}_i \leftarrow \mathbf{A}\mathbf{p}_i$	spMV_bdia_gh_kernel	BDIA-GH/CUSPARSE
12	$\alpha_i \leftarrow \rho_i / (\mathbf{r}_0, \hat{\mathbf{v}}_i)$	dot_kernel	CUBLAS
13	$\mathbf{s} \leftarrow \mathbf{r}_{i-1} - \alpha\mathbf{v}_i$	copy_kernel	CUBLAS
14		axpy_kernel_val	CUBLAS
15	$\mathbf{t} \leftarrow \mathbf{A}\mathbf{s}$	spMV_kernel	BDIA-GH/CUSPARSE
16	$\omega_i \leftarrow (\mathbf{t}, \mathbf{s}) / (\mathbf{t}, \mathbf{t})$	dot_kernel	CUBLAS
17		dot_kernel	CUBLAS
18	$\mathbf{x}_i \leftarrow \mathbf{x}_{i-1} + \alpha\mathbf{p}_i + \omega_i\mathbf{s}$	axpy_kernel_val	CUBLAS
19		axpy_kernel_val	CUBLAS
20	$(\mathbf{x}_i, \mathbf{x}_i)$	dot_kernel	CUBLAS
21	$\mathbf{r}_i \leftarrow \mathbf{s} - \omega_i\mathbf{t}$	copy_kernel	CUBLAS
22		axpy_kernel_val	CUBLAS

for Multi-GPU BiCGStab implementation. At the step 7, 12, 16, 17 and 20 the reductions are occurring. It means all the GPUs should communicate to gather the dot product. Before SpMV at step 2, 11 and 15, a global synchronization also has to be called to make sure the operation on SpMV using previous arrays is correct because there need communication between GPUs and CPU to transfer part of the array from previous operation on device to host and host to device for SpMV. For all of the SpMV operations are using our own code BDIA-GH and also CUSPARSE library (step 2, 11 and 15), while other operations like vector dot products (step 7, 12, 16, 17 and 21), axpy operation (step 3, 8, 10, 14, 18, 19, and 22), and scal (step 9) are using CUBLAS library.

There are several model communications that we tried in this implementation to find out which communication model is the best. There are two types of storing memory in host which are pinned memory or paged memory. For the data copying between host and devices, there are synchronous and asynchronous memory copy. Thus, there will be 4 combination of communication model: synchronous pinned memory copy, asynchronous pinned memory copy, synchronous paged memory copy, and asynchronous paged memory copy.

---

**Algorithm 11** Multi-GPU implementation Algorithm

---

```

1:  $n \leftarrow N_{gpu}$ 
2: for  $i \leftarrow 0$  to  $n$  do
3:   cudaSetDevice( $i$ )
4:   cudaFunction()
5: end for

```

---

## 5.5 Analysis of Deadlock in Inter-Block Synchronization

In order to know how deadlock on GPU (Graphics Processing Unit) could occur, we have to understand about how the GPU distributing works between the SMX (Streaming Multiprocessor) inside the GPU. First of all, we have to know the specification of GPU that we used. NVIDIA Kepler Architecture Tesla K20Xm 3.2 compute capability. It means the maximum number of resident blocks per streaming multiprocessor is 16 and this GPU has 14 SMX.

Secondly, as the flowchart in figure 5.7, we have to carry out the experiment by ourself to understand how the block dispatching order works because NVIDIA didn't give any of detail about it. To do that, we initialize two arrays A and B on host with the size of number of blocks to record start and end time for every block execution. After that, these two arrays copied into device so every block could access on it. Furthermore, we invoke dummy kernel and array A[blockID] will be filled with clock cycle by using inline PTX (parallel thread execution) assembly in CUDA to record when the block start to be executed. Dummy work is executed inside the block before the clock cycle being recorded in array B[blockID] to know when the clock ends and the block is finished and the kernel will take another block until the kernel exit. After exiting the kernel, array A and B are copied from device to host, and print it to a file.

From the figure 5.8, if we saw the clock start for each block, we could see that the clock start lost it patterns when it reaches blockID 224 and its multiplier. And if

we order the table based on clock start, we could see that at the first 224 block, the block is starting randomly, but the next 224 block, is starting from the lowest blockID number to the highest. From that observation, we could conclude that 224 block is working concurrently, and whenever a block is finished, the lowest number of block in the next group blocks will be started.

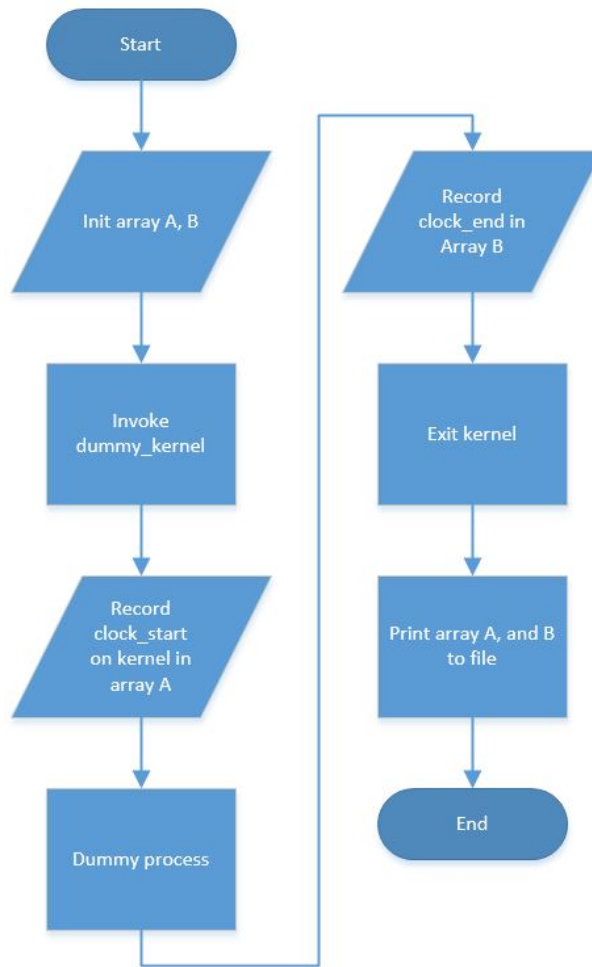


Figure 5.7: Flowchart of block dispatching order

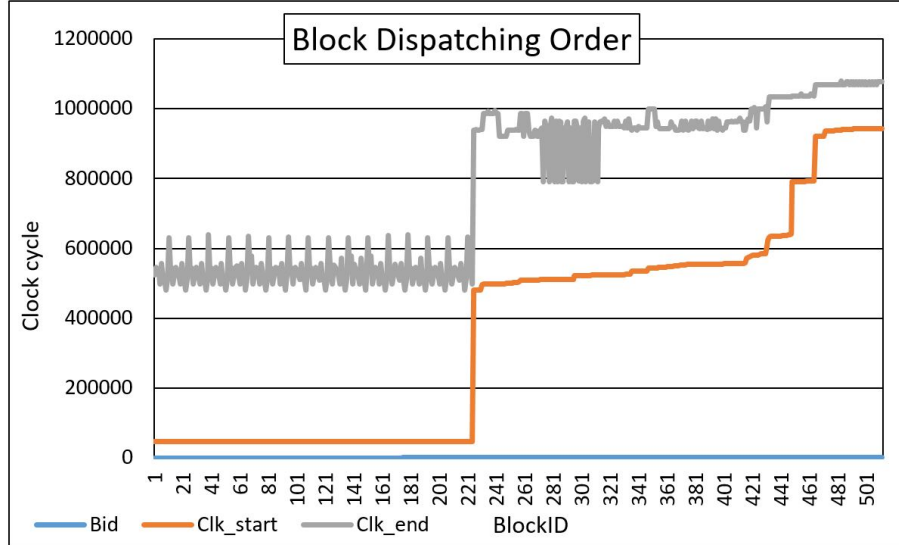


Figure 5.8: Block dispatching order result

### 5.5.1 Evaluating Number of Active Blocks on Kernel in Kepler K20Xm and Deadlock Condition with Experimental Validation

In order to understand the rule of number of active blocks that limited by number of register, shared memory, and warp, several experiments has been done. The kernel that used in this experiment is Jacobi Algorithm. The machine used for this experiment is NVIDIA Kepler Tesla K20Xm with compute capability of 3.2.

Note:

$B_a$  = block limit due to architecture

$B_l$  = Block size in launch configuration

$B_s$  = block limit per multiprocessor due to shared memory usage

$B_r$  = Limit number of active blocks due to register

$B_w$  = Block limit per multiprocessor due to warp

$R_m$  = total registers per multiprocessor

$R$  = registers used

$R_a$  = register unit allocation size

$ShM_b$  = shared memory allocation per block

$ShM_u$  = shared memory used per block

$ShM_a$  = shared memory unit allocation size

$ShM_x$  = configured shared memory per multiprocessor

$T_w$  = threads per warp

$W_a$  = warp allocation granularity

$W_b$  = Number of warps per block

$W_l$  = warps per block in launch configuration

$W_r$  = warps per multiprocessor imposed by register use

$W_s$  = warp size for architecture

$W_x$  = maximum warp per multiprocessor

1. Block Size: 16 1 1, Shared Memory: 1.0625KB, Registers: 38

With this configuration of kernel, from previous work by Anas Almousa's Thesis (KFUPM, 2017), we could analyse deadlock-occupancy trade-off in inter-block GPU synchronization.

The first parameter we need to calculate is shared memory allocation per multiprocessor. With compute capability 3.2, the allocation unit size for the

shared memory ( $ShM_a$ ) is 256, so

$$ShM_b = \left\lceil \frac{ShM_u}{ShM_a} \right\rceil \times ShM_a = \left\lceil \frac{1062.5}{256} \right\rceil \times 256 = 1280 \text{ bytes} \quad (5.9)$$

After that, to calculate the block limit per multiprocessor due to shared memory usage, we could divide the configured shared memory size in the processor (for Kepler Tesla K20Xm it is 49152 bytes) by the used shared memory.

$$B_s = \left\lfloor \frac{ShM_x}{ShM_b} \right\rfloor = \left\lfloor \frac{49152}{1280} \right\rfloor = 38 \text{ blocks} \quad (5.10)$$

Another factor for number of active blocks per multiprocessor is the limit due to number of warps to be launched on a multiprocessor.

$$W_b = \left\lceil \frac{B_l}{W_s} \right\rceil = \left\lceil \frac{16}{32} \right\rceil = 1 \text{ warp} \quad (5.11)$$

$$B_w = \left\lfloor \frac{W_x}{W_b} \right\rfloor = \left\lfloor \frac{64}{1} \right\rfloor = 64 \text{ blocks} \quad (5.12)$$

Other factor that limiting the number of active block is register. To calculate the limit due to register usage:

$$W_r = \left\lfloor \frac{\frac{R_m}{\left\lceil \frac{R \times T_w}{R_a} \right\rceil \times R_a}}{W_a} \right\rfloor \times W_a = \left\lfloor \frac{\frac{65536}{\left\lceil \frac{38 \times 32}{256} \right\rceil \times 256}}{4} \right\rfloor \times 4 = 48 \text{ warps}$$



(5.13)

Hence

$$B_r = \left\lfloor \frac{W_r}{W_l} \right\rfloor = \left\lfloor \frac{48}{1} \right\rfloor = 48 \text{ blocks} \quad (5.14)$$

After finishing this calculation, we could conclude that number of active blocks per multiprocessor would be

$$\text{Resident blocks} = \min\{B_s, B_w, B_r, B_a\} = \min\{38, 64, 48, 16\} = 16 \text{ blocks} \quad (5.15)$$

After experiment by using various number of block (table 5.3), this kernel configuration got deadlock on number of blocks 225, which is exceeding available  $\text{number of resident block} \times \text{number of processor} = 16 \times 14 = 224 \text{ blocks}$ .

2. Block Size: 32 1 1, Shared Memory: 4.125KB, Registers: 38

With this configuration of kernel, like previous calculation, we could analyse Deadlock-occupancy trade-off in inter-block GPU synchronization.

The first number we need to calculate is shared memory allocation per multiprocessor. With compute capability 3.2, the allocation unit size for

the shared memory is 256, hence

$$ShM_b = \left\lceil \frac{ShM_u}{ShM_a} \right\rceil \times ShM_a = \left\lceil \frac{4125}{256} \right\rceil \times 256 = 4352 \text{ bytes} \quad (5.16)$$

After that, to calculate the block limit per multiprocessor due to shared memory usage, we could divide the configured shared memory size in the processor (for Kepler Tesla K20Xm it is 49152 bytes) by the used shared memory.

$$B_s = \left\lfloor \frac{ShM_x}{ShM_b} \right\rfloor = \left\lfloor \frac{49152}{4352} \right\rfloor = 11 \text{ blocks} \quad (5.17)$$

Another factor for number of active blocks per multiprocessor is the limit due to number of warps to be launched on a multiprocessor.

$$W_b = \left\lceil \frac{B_l}{W_s} \right\rceil = \left\lceil \frac{32}{32} \right\rceil = 1 \text{ warp} \quad (5.18)$$

$$B_w = \left\lfloor \frac{W_x}{W_b} \right\rfloor = \left\lfloor \frac{64}{1} \right\rfloor = 64 \text{ blocks} \quad (5.19)$$

Other factor that limiting the number of active block is register. To calculate the limit due to register usage:

$$W_r = \left\lfloor \frac{\frac{R_m}{\left\lceil \frac{R \times T_w}{R_a} \right\rceil \times R_a}}{W_a} \right\rfloor \times W_a = \left\lfloor \frac{\frac{65536}{\left\lceil \frac{38 \times 32}{256} \right\rceil \times 256}}{4} \right\rfloor \times 4 = 48 \text{ warps}$$

(5.20)

Hence

$$B_r = \left\lfloor \frac{W_r}{W_l} \right\rfloor = \left\lfloor \frac{48}{1} \right\rfloor = 48 \text{ blocks} \quad (5.21)$$

After finishing this calculation, we could conclude that number of active blocks per multiprocessor would be

$$\text{Resident blocks} = \min\{B_s, B_w, B_r, B_a\} = \min\{11, 64, 48, 16\} = 11 \text{ blocks} \quad (5.22)$$

After experiment by using various number of block (table 5.3), this kernel configuration got deadlock on number of blocks 155, which is exceeding available  $\text{number of resident block} \times \text{number of processor} = 11 \times 14 = 154 \text{ blocks}$ .

3. Block Size: 48 1 1, Shared Memory: 9.1875KB, Registers: 43

With this configuration of kernel, like previous calculation, we could analyse deadlock-occupancy trade-off in inter-block GPU synchronization.

The first number we need to calculate is shared memory allocation per multiprocessor. With compute capability 3.2, the allocation unit size for

the shared memory is 256, hence

$$ShM_b = \left\lceil \frac{ShM_u}{ShM_a} \right\rceil \times ShM_a = \left\lceil \frac{9187.5}{256} \right\rceil \times 256 = 9216 \text{ bytes} \quad (5.23)$$

After that, to calculate the block limit per multiprocessor due to shared memory usage, we could divide the configured shared memory size in the processor (for Kepler Tesla K20Xm it is 49152 bytes) by the used shared memory.

$$B_s = \left\lfloor \frac{ShM_x}{ShM_b} \right\rfloor = \left\lfloor \frac{49152}{9216} \right\rfloor = 5 \text{ blocks} \quad (5.24)$$

Another factor for number of active blocks per multiprocessor is the limit due to number of warps to be launched on a multiprocessor.

$$W_b = \left\lceil \frac{B_l}{W_s} \right\rceil = \left\lceil \frac{64}{32} \right\rceil = 2 \text{ warps} \quad (5.25)$$

$$B_w = \left\lfloor \frac{W_x}{W_b} \right\rfloor = \left\lfloor \frac{64}{2} \right\rfloor = 32 \text{ blocks} \quad (5.26)$$

Other factor that limiting the number of active block is register. To calculate the limit due to register usage:

$$W_r = \left\lfloor \frac{\frac{R_m}{\left\lceil \frac{R \times T_w}{R_a} \right\rceil \times R_a}}{W_a} \right\rfloor \times W_a = \left\lfloor \frac{\frac{65536}{\left\lceil \frac{43 \times 32}{256} \right\rceil \times 256}}{4} \right\rfloor \times 4 = 40 \text{ warps}$$

(5.27)

Hence

$$B_r = \left\lfloor \frac{W_r}{W_l} \right\rfloor = \left\lfloor \frac{40}{1} \right\rfloor = 40 \text{ blocks} \quad (5.28)$$

After finishing this calculation, we could conclude that number of active blocks per multiprocessor would be

$$\text{Resident blocks} = \min\{B_s, B_w, B_r, B_a\} = \min\{5, 32, 36, 16\} = 5 \text{ blocks} \quad (5.29)$$

After experiment by using various number of block (table 5.3), this kernel configuration got deadlock on number of blocks 71, which is exceeding available  $\text{number of resident block} \times \text{number of processor} = 5 \times 14 = 70 \text{ blocks}$ .

### 5.5.2 General Conclusion

Linear algebra algorithms  $Ax = b$  in computation are being solved using iterative method where for every iteration, the solution of  $x$  is getting narrow to the real solution. In GPGPU computation for iterative linear algebra solver like Jacobi algorithm where the elements are distributed on every processing unit, it is required for every element to be synchronized on every iteration to make sure the

data correctness because the data layout of this algorithm is all cooperatively computed results in one iteration are needed for every thread of the next iteration. NVIDIA with his CUDA is facilitating programmers to do thread synchronization within block using `__syncthreads()` but that is not the case for inter-block synchronization. In iterative linear algebra solver algorithm, there will be a deadlock for any inter-block synchronization if number of blocks is exceeding number of resident blocks, because the active blocks is waiting for inactive blocks to be completed (as shown in table 5.3).

Table 5.3: Validation Experiment for Analyzing Deadlock on inter-block GPU synchronization using Jacobi Algorithm

Jacobi Algorithm						
No	Threads/ Block	ShM/ Block (KB)	Regs/ Block	Resident Blocks	Blocks in Kernel	Time (ms) or Deadlock
1	16	1.0625	38	224	16	137.19
2	16	1.0625	38	224	64	34.926
3	16	1.0625	38	224	128	28.326
4	16	1.0625	38	224	223	30.677
5	16	1.0625	38	224	224	30.76
6	16	1.0625	38	224	225	Deadlock
7	16	1.0625	38	224	226	Deadlock
8	16	1.0625	38	224	256	Deadlock
9	16	1.0625	38	224	512	Deadlock
10	32	4.125	38	154	16	63.907
11	32	4.125	38	154	64	26.264
12	32	4.125	38	154	128	26.943
13	32	4.125	38	154	153	27.64
14	32	4.125	38	154	154	27.618
15	32	4.125	38	154	155	Deadlock
16	32	4.125	38	154	156	Deadlock
17	32	4.125	38	154	256	Deadlock
18	48	9.1875	43	70	16	72.145
19	48	9.1875	43	70	64	29.785
20	48	9.1875	43	70	69	29.934
21	48	9.1875	43	70	70	30.024
22	48	9.1875	43	70	71	Deadlock
23	48	9.1875	43	70	72	Deadlock
24	48	9.1875	43	70	128	Deadlock

## CHAPTER 6

# RESULTS AND DISCUSSION

In this thesis, there are several experiments that have been carried out to evaluate the performance of the proposed storage scheme, SpMV, and BiCGStab solver algorithm. Our target is to develop an optimized CUDA implementation for a class of iterative linear algebra solver, in this case is BiCGStab. Firstly, we are comparing our method with available libraries on sparse matrix vector multiplication. Secondly, we are comparing BiCGStab implementation that use our own SpMV with BiCGStab that use the available libraries. After that, we are benchmarking the available memory transfer methods to determine which method we should use to generate the best results for BiCGStab on Multi GPU. All of these results are using the same GPU system. The specification of the GPU is available in Table 2.1.



## 6.1 Sparse Matrix-Vector Multiplication

In order to know the effectiveness of our implementation on sparse matrix vector multiplications, we compare our BDIA-GH SpMV with the available SpMV from CUSPARSE. The result from the table 6.1 figure 6.1 and figure 6.2 shows that our BDIA-GH SpMV has the best performance compared to other storage format along with the SpMV of CUSPARSE library. Every row has different configuration of matrix (table 5.1) that is characterized by specific number of non-zeros, i.e. to quantify the number of arithmetic operations involved in each problem configuration. Where the formula of number of non-zero is as follows:

$$nnz = (7 \times N_c^2 \times J \times H \times I) - (N_c^2 \times (2 + I + I \times H)) \quad (6.1)$$

It can be seen that with different size of blocks but similar number of non-zeros, CSR and BSR show the worst performance on block size of 8. The best performance is found for the block size of 4 for BSR and block size of 16 for CSR while block size of 16 slightly take longer execution time compared to block size of 4 on BSR and block size of 4 is slightly faster compared to block size of 8 on CSR. On the other hand, BDIA-GH and HYB give similar performance within its own format regardless the block size as long as the number of non-zero as far as the nnz are comparable. The execution time scale well for all formats versus the number of non-zero when the size is large enough (starting from nnz=14540288). From the table 6.2 and figure 6.3, the speedup of our implementation with BDIA-GH

gain over CSR format is up to 3.067, and BSR is up to 2.86, while compared to HYB the speedup is up to 1.404. The speedup over BSR and CSR are vary along with the variations of the block size.

This best performance of BDIA-GH SpMV is due to the effective access of the data like offset that stored in shared memory and the data is being accessed in coalesced manner and it taken from memory contiguously while the other formats are accessing more data, pointer, or indices that could be not available in shared memory and the access not in coalesced manner, also memory access is not contiguous that leads to longer communication time.

From the checkPracticalErrors function, this SpMV implementation has error means of  $1.2503 \times 10^{-7}$ , the highest error value is  $1.90735 \times 10^{-6}$ , and the lowest error value is  $-1.90735 \times 10^{-6}$  compared to serial implementation. Note that SpMV is invoked 100 times for each measurement of the average execution time that will be reported on the performance plots. From the table 6.3 and figure 6.4 and 6.5 we could see that the best performance by floating-point operations per second (FLOPS) reach by BDIA-GH format with about 45 GFLOPS. This result means the implementation of BDIA-GH SpMV only reach about 2.4% of peak performance of GK210 with 1.87 TFLOPS. This low performance is acceptable for sparse matrix vector multiplication due to its characteristic that it is a memory bound application [59]. Unlike matrix multiplications that perform  $O(n^3)$  operations on  $O(n^2)$  data, the sparse matrix vector multiplication is performing  $O(n^2)$  operations on  $O(n^2)$  data. It means that the number of memory access

Table 6.1: SpMV Execution Time of Various Storage Formats (Single GPU)

SpMV Execution Time of Various Storage Formats (Single GPU)						
N	Block Size	nnz	Execution Time (ms)			
			BDIA-GH	CSR	BSR	HYB
65536	8	3635072	0.18851	0.50463	0.47276	0.24265
131072	8	7272320	0.1878	0.50389	0.47292	0.24259
131072	16	14540288	0.66271	1.2885	1.1027	0.91189
524288	4	14613472	0.68753	1.8479	0.93752	0.93817
262144	16	29089280	1.3078	2.5671	2.2011	1.8007
524288	8	29224832	1.3351	4.0743	3.7693	1.8147
1048576	4	29228000	1.3646	3.8202	1.952	1.888
1048576	8	58453888	2.6393	8.097	7.5488	3.6241
1048576	16	116899328	5.1904	10.283	8.8488	7.2885
2097152	8	116912000	5.3238	15.991	14.53	7.2856

ratio to the number of floating point operation on SpMV is higher than  $M \times M$ . The memory access of SpMV is also higher compared to  $M \times M$  operation. As we know the SpMV has additional data to be accessed called offsets (in BDIA-GH) to get the benefit of the sparsity of the matrix. This memory access operation is greatly degrade the performance of the SpMV since there will be additional cache interference, memory bandwidth pressure, and memory access operations.

## 6.2 BiCGStab on Single GPU

Once the performance of SpMV from the various format has been assessed, the implementations of various SpMV on BiCGStab are collected. In our experiments, the BiCGStab inner loop is computed 5 times in each BiCGstab invocation. Note that BiCGStab is invoked 100 times for each measurement of the execution time that will be reported on the performance plots. In each BiCGStab invocation we have 11 calls to SpMV (i.e. 1 SpMV call before loop, and 10 SpMV calls

Table 6.2: SpMV Speedup of BDIA-GH over Other Storage Formats (Single GPU)

SpMV Speedup of BDIA-GH over Other Storage Formats (Single GPU)					
N	Block Size	nnz	Speedup BDIA-GH over Other Formats		
			CSR	BSR	HYB
65536	8	3635072	2.67694	2.50788	1.2871996
131072	8	7272320	2.68312	2.51821	1.2917465
131072	16	14540288	1.94429	1.66393	1.3760016
524288	4	14613472	2.68774	1.36455	1.3645514
262144	16	29089280	1.96291	1.68306	1.3768925
524288	8	29224832	3.05168	2.82323	1.359224
1048576	4	29228000	2.7995	1.43046	1.3835556
1048576	8	58453888	3.06786	2.86015	1.3731292
1048576	16	116899328	1.98116	1.70484	1.404227
2097152	8	116912000	3.00368	2.72925	1.3684962

Table 6.3: SpMV FLOPS of Various Storage Formats (Single GPU)

SpMV FLOPS of Various Storage Formats (Single GPU)						
N	Block Size	nnz	GFLOPS			
			BDIA-GH	CSR	BSR	HYB
65536	8	3635072	38.5664	14.4069	15.378086	29.9614424
131072	8	7272320	77.4475	28.8647	30.754969	59.9556453
131072	16	14540288	43.8813	22.5693	26.372156	31.8904429
524288	4	14613472	42.5101	15.8163	31.174742	31.1531428
262144	16	29089280	44.4858	22.6631	26.431584	32.3088577
524288	8	29224832	43.7792	14.3459	15.506769	32.2089954
1048576	4	29228000	42.8375	15.3018	29.946721	30.9618644
1048576	8	58453888	44.295	14.4384	15.486935	32.25843
1048576	16	116899328	45.0444	22.7364	26.42151	32.0777466
2097152	8	116912000	43.9205	14.6222	16.092498	32.0939936

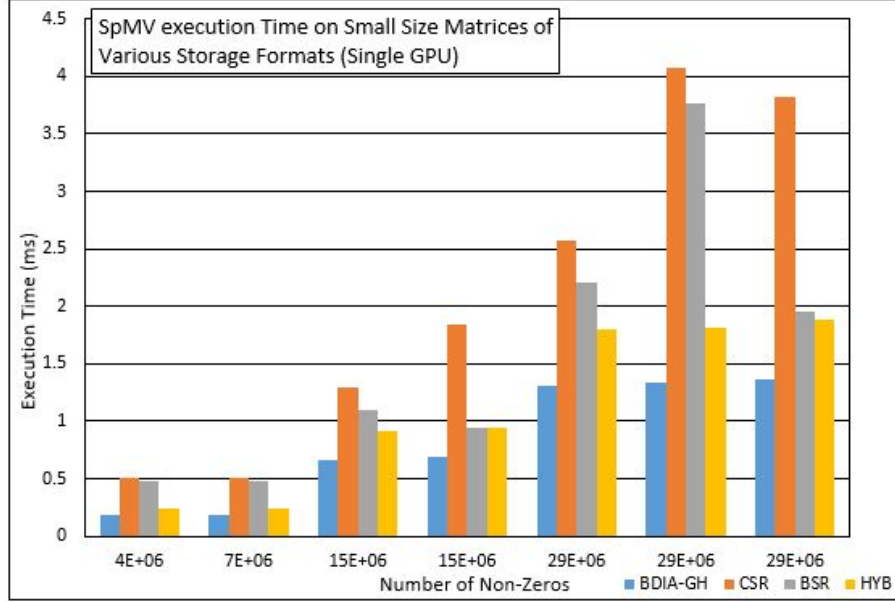


Figure 6.1: SpMV execution Time on Small Size Matrices of Various Storage Formats (Single GPU)

inside the loop). In one iteration, percentage of exit-re-entry kernel of BiCGStab implementation is 0.192%, with average of exit-re-entry kernel time is  $10 \pm 1.567 \times 10^{(-5)} us$ . From the table 6.4 which represented in chart by figure 6.6 and 6.7, it shows that the BDIA-GH in BiCGStab is performing the best compared to the other formats. BDIA-GH format could reach up to 2.64 speedup over CSR, 2.41 over BSR, and 1.352 over HYB. We could see that the speedup of BDIA-GH BiCGStab over other formats are quite comparable to the SpMV. This is due the fact the SpMV is the main function of BiCGStab and the other functions are scalable vector operations. This especially true for larger problem sizes where problem scalability is proved to be more effective. On small sizes of the solver matrix (i.e. nnz=3635072), SpMV takes about 75% of overall operations, while on large size of matrix (i.e. nnz=116912000), SpMV takes about 90% of overall

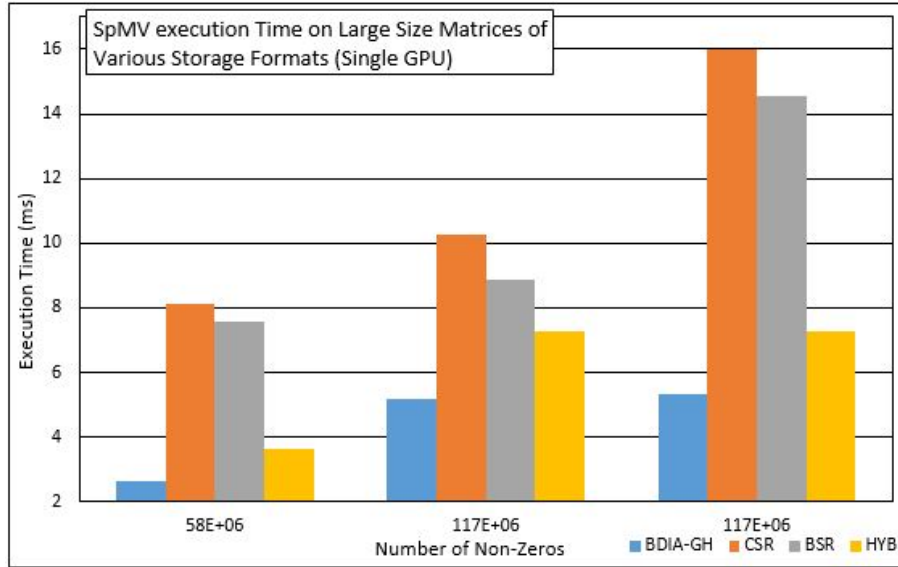


Figure 6.2: SpMV execution Time on Large Size Matrices of Various Storage Formats (Single GPU)

operations. The error on this implementation of result  $x$  compared to serial is  $1.54069 \times 10^{-5}$  for error means, the highest error value is  $6.38962 \times 10^{-5}$ , and the lowest error value is  $-6.00815 \times 10^{-5}$ .

### 6.3 CUDA Memory Transfer

This experiment is to figure out which kind of memory transfer will perform the best on BiCGStab. This communication models mimics the communication model of BiCGStab. The implementation is using left-right approach that proposed by [60] and it is fully compatible with our BiCGStab implementation.

Figure 6.11 illustrate how the communication is being done. Our goal is to distribute current GPU data to the neighbour GPU(s). We are implementing this methods using device-host communication and peer-to-peer (Figure 6.12) GPUDirect communication along with synchronous and asynchronous

Table 6.4: BiCGStab Execution Time on GH Matrices of Various Storage Formats (Single GPU)

BiCGStab Execution Time on GH Matrices of Various Storage Formats (Single GPU)						
N	Block Size	nnz	Execution Time (ms)			
			BDIA-GH	CSR	BSR	HYB
65536	8	3635072	3.479	6.745	5.198	4.008
131072	8	7272320	4.786	10.974	8.617	5.991
131072	16	14540288	10.116	16.217	14.512	12.727
524288	4	14613472	12.126	24.768	14.609	14.702
262144	16	29089280	14.136	30.668	17.949	18.479
524288	8	29224832	14.3	48.89	30.234	19.294
1048576	4	29228000	22.075	47.917	27.688	27.74
1048576	8	58453888	35.98	95.126	86.792	46.493
1048576	16	116899328	64.251	117.408	101.92	86.921
2097152	8	116912000	71.115	188.163	171.983	92.391

Table 6.5: Speedup BiCGStab BDIA-GH over Other Formats (Single GPU)

Speedup BiCGStab BDIA-GH over Other Formats (Single GPU)					
N	Block Size	nnz	Speedup		
			CSR	BSR	HYB
65536	8	3635072	1.938776	1.494108	1.1520552
131072	8	7272320	2.292938	1.80046	1.251776
131072	16	14540288	1.603104	1.434559	1.258106
524288	4	14613472	2.042553	1.204767	1.2124361
262144	16	29089280	2.169496	1.269737	1.3072298
524288	8	29224832	3.418881	2.114266	1.3492308
1048576	4	29228000	2.170646	1.25427	1.2566251
1048576	8	58453888	2.643858	2.412229	1.2921901
1048576	16	116899328	1.827333	1.586279	1.352835
2097152	8	116912000	2.645897	2.418379	1.2991774

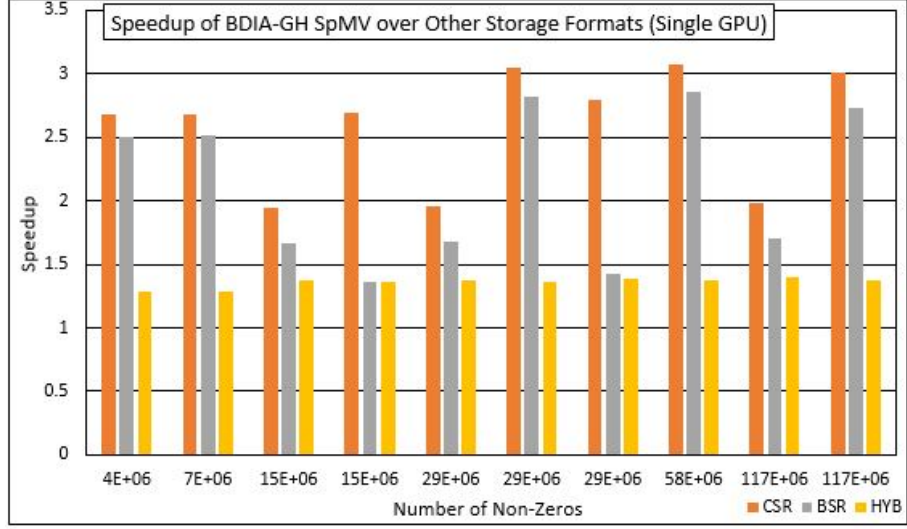


Figure 6.3: Speedup of BDIA-GH SpMV over Other Storage Formats (Single GPU)

communication, and also pinned and paged host memory models. The result (Table 6.7, 6.8,6.9 and Figure 6.13, 6.14, 6.15) shows that asynchronous device-host with pinned memory communication model give the best result. In the 2 GPUs, the ideal throughput of device-host-device communication model is 6GB/s [60] while our implementation reach up to 5.99 GB/s (Figure 6.7). It means that for 2 GPUs communication, the implementation is almost fully parallel. For the 4 GPUs, the ideal throughput of device-host-device communication is 12 GB/s, while our implementation reach about 9.72 GB/s (Figure 6.8) because of the overhead from device synchronization. Unfortunately, for the 8 GPUs implementation, our implementation is very degraded from the ideal scaling of 24 GB/s, we only got 3.67 GB/s due to the limitation of the PCIe channel of 4.



Table 6.6: BiCGStab GFLOPS on GH Matrices of Various Storage Formats (Single GPU)

BiCGStab GFLOPS on GH Matrices of Various Storage Formats (Single GPU)						
N	Block Size	nnz	GFLOPS			
			BDIA-GH	CSR	BSR	HYB
65536	8	3635072	24.88954	12.83776	16.658465	21.60447
131072	8	7272320	36.195	15.78543	20.103202	28.91492
131072	16	14540288	32.93047	20.54169	22.955112	26.17464
524288	4	14613472	30.87988	15.11828	25.631423	25.46929
262144	16	29089280	47.14493	21.73082	37.129683	36.06476
524288	8	29224832	48.66429	14.23398	23.017112	36.06818
1048576	4	29228000	33.92626	15.62957	27.048619	26.99791
1048576	8	58453888	38.68515	14.63208	16.037097	29.93766
1048576	16	116899328	41.67548	22.80672	26.272482	30.80604
2097152	8	116912000	39.14612	14.79503	16.186927	30.13147

Table 6.7: Transfer rate between 2 GPUs with various memory access

Transfer rate between 2 GPUs with various memory access						
Chunk Size	Data Size	Throughput (GB/s)				
		Async Pinned	Sync Pinned	Async Paged	Sync Paged	P2P
128	256	1.41436	1.04634	0.06468	0.06567	0.00479
256	512	3.48299	0.40716	0.83388	0.98344	0.01005
512	1024	3.64413	1.9256	1.02196	1.20804	0.01976
2048	4096	5.58799	2.2258	1.5176	1.59725	0.08093
4096	8192	5.83060	3.49717	1.76021	1.83908	0.16232
8192	16384	5.99927	3.60504	1.86904	1.98709	0.32107

Table 6.8: Transfer rate between 4 GPUs with various memory access

Transfer rate between 4 GPUs with various memory access						
Chunk Size	Data Size	Throughput (GB/s)				
		Async Pinned	Sync Pinned	Async Paged	Sync Paged	P2P
128	768	3.03557	1.07459	0.64864	0.88621	0.01555
256	1536	6.144	1.94645	1.59336	1.6432	0.02714
512	3072	7.52941	2.91401	2.39066	2.39755	0.04680
2048	12288	9.02202	3.39670	2.03040	1.93451	0.1463
4096	24576	9.53297	3.53423	1.91715	2.03555	0.42152
8192	49152	9.72344	2.4316	0.77033	0.75570	0.8294

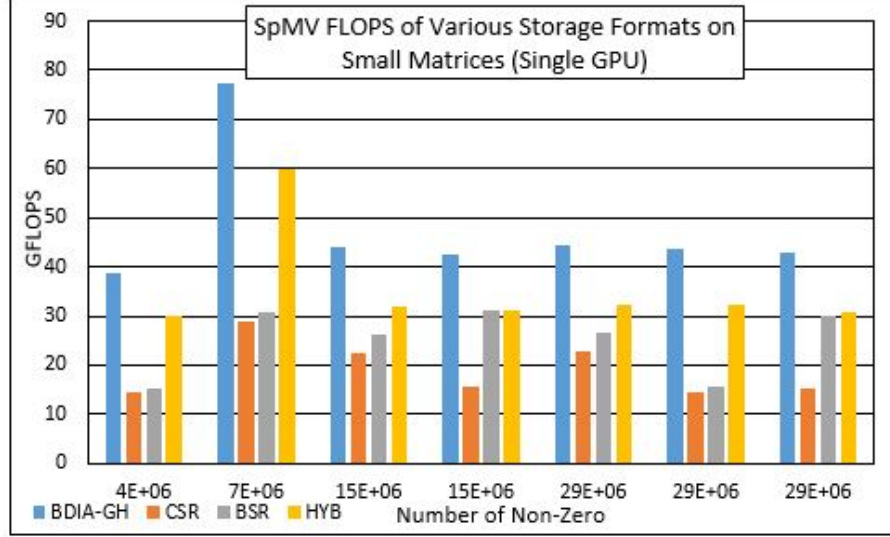


Figure 6.4: SpMV FLOPS of Various Storage Formats on Small Matrices (Single GPU)

Table 6.9: Transfer rate between 8 GPUs with various memory access

Transfer rate between 8 GPUs with various memory access						
Chunk Size	Data Size	Throughput (GB/s)				
		Async Pinned	Sync Pinned	Async Paged	Sync Paged	P2P
128	1792	5.25513	1.3053	1.06921	1.1899	0.01203
256	3584	5.95348	1.84911	1.5548	1.47347	0.02419
512	7168	5.05857	2.5150	1.64177	1.64394	0.0475
2048	28672	3.43459	2.4020	1.76725	1.78979	0.14462
4096	57344	3.63581	2.40781	1.99909	2.1498	0.28802
8192	114688	3.67613	2.43086	2.3283	2.31371	1.06592

## 6.4 BiCGStab on Multi GPU

Multi-GPU implementation on BiCGStab has been done by using 2, 4, and 8 GPUs of GK210. We could see from the table 6.10, figure 6.16, 6.17, the SpMV performance is almost fully scaled on every number of GPUs for various size of matrices. It means that the operations (especially SpMV) is relatively load balanced for all the studied number of GPUs and configurations.

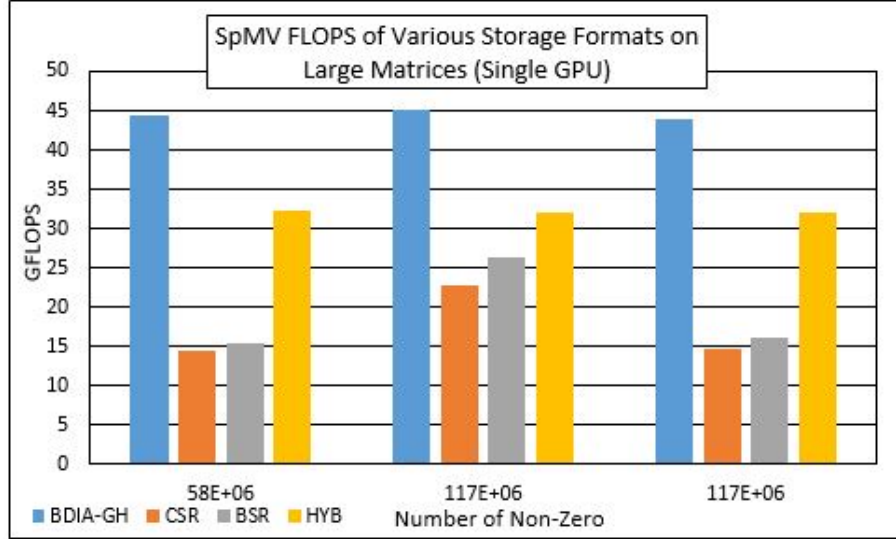


Figure 6.5: SpMV FLOPS of Various Storage Formats on Large Matrices (Single GPU)

From the table 6.11 and figure 6.18, we observe that the execution time on multi-GPU does not scale perfectly versus problem size. This is happening due to the computation over communication that increases gradually along with the problem size. At the larger size (figure 6.19), the larger computation could amortize the communication overhead so that the speedup could go almost to 2 (figure 6.20, 6.21) when the problem size is doubled on each multi-GPU configuration. It means on the 2 GPUs, the implementation is scaled well and the best communication model is shown by asynchronous pinned memory.

From the table 6.13 that represented by the figure 6.22 and 6.23, we could see that at the small size of matrices the execution time is not scaled well while on the larger size, the execution time is scaled well. The asynchronous pinned memory communication scheme shows the best performance over other

Table 6.10: Execution time of SpMV with Various Number of GPU(s)

Execution time of SpMV with Various Number of GPU(s)					
N size	nnz	Execution time (ms)			
		1 GPU	2 GPU <sub>s</sub>	4 GPU <sub>s</sub>	8 GPU <sub>s</sub>
4096	110304	0.028407	0.02671	0.03398	0.077103
8192	224992	0.027291	0.0273	0.02974	0.060425
16384	450272	0.037309	0.03002	0.04191	0.038135
32768	900576	0.066582	0.0483	0.02778	0.058688
65536	1818080	0.102553	0.06767	0.04647	0.056994
131072	3636704	0.184461	0.10352	0.06816	0.038174
262144	7273440	0.352834	0.18491	0.10322	0.068892
524288	14613472	0.69292	0.36019	0.18644	0.103858
1048576	29228000	1.351868	0.68468	0.35393	0.184603
2097152	58456032	2.358288	1.35347	0.68607	0.351683
4194304	117176288	4.362842	2.42619	1.38289	0.661889
8388608	234354656	8.101533	4.38098	2.24341	1.184934
1.7E+07	468709344	16.237883	8.16651	4.06914	2.713917
3.4E+07	938471392	32.635021	16.5146	8.18372	4.076508

Table 6.11: Execution Time of 2 GPUs BiCGStab with Various Communication Model

Execution Time of 2 GPUs BiCGStab with Various Communication Model					
N size	nnz	Execution Time (ms)			
		Async Pinned	Sync Pinned	Sync Paged	Async Paged
4096	110304	3.174	5.419	5.562719	4.983
8192	224992	4.933	5.266	5.665826	5.237
16384	450272	4.923	5.846	5.62379	5.023
32768	900576	7.196	5.39	5.684459	5.392
65536	1818080	5.525	5.96	6.54797	5.768
131072	3636704	6.063	6.525	6.99361	6.398
262144	7273440	6.506	6.327	7.939675	7.46
524288	14613472	9.046	9.885	9.103636	9.689
1048576	29228000	12.106	11.798	14.451855	13.316
2097152	58456032	19.804	37.814	30.867605	21
4194304	117176288	36.676	48.966	39.448963	38.496
8388608	234354656	69.934	72.562	74.416008	73.528
16777216	468709344	135.757	139.845	141.790466	140.805
33554432	938471392	264.878	276.79	279.762573	282.147



Figure 6.6: BiCGStab Execution Time on Small Size GH Matrices of Various Storage Formats (Single GPU)

schemes. While synchronous with pinned memory shows better performance over synchronous with paged memory and asynchronous with paged memory and the worst performance is coming from asynchronous paged memory. The speedup that shown by table 6.14 and figure 6.24, 6.25, conclude that maximum speedup of 4 GPUs implementation could reach up to 3.28. The speedup is not achieving ideal speedup of 4 due to the communication that will be described in the section 6.5.

On 8 GPUs implementation results (table 6.15), as the previous results, we could see that on the small size of matrices the execution time is not scaled well over the size of the matrices (figure 6.26) while on the larger size (figure 6.27) the execution time is scaled well over the size of the matrices. Asynchronous pinned memory communication model shows the best performance over others. For the speedup (table 6.16, figure 6.28, 6.29), we could observe a speedup of almost 4.5

Table 6.12: Speedup of 2 GPUs BiCGStab with Various Communication Model over Single GPU

Speedup of 2 GPUs BiCGStab with Various Communication Model over Single GPU					
N size	nnz	Speedup			
		Async Pinned	Sync Pinned	Sync Paged	Async Paged
4096	110304	0.729364	0.427201	0.416163391	0.46458
8192	224992	0.469288	0.439613	0.408590027	0.442047
16384	450272	0.486695	0.409853	0.426047203	0.477006
32768	900576	0.378683	0.505566	0.479377193	0.505378
65536	1818080	0.581719	0.539262	0.490839146	0.557212
131072	3636704	0.655121	0.608736	0.567947026	0.620819
262144	7273440	0.960344	0.987514	0.786933974	0.837534
524288	14613472	1.05472	0.9652	1.048042782	0.984725
1048576	29228000	1.376921	1.412867	1.153415946	1.251802
2097152	58456032	1.595587	0.835643	1.023694582	1.504714
4194304	117176288	1.717417	1.286362	1.596695964	1.636222
8388608	234354656	1.78484	1.720198	1.677340714	1.697598
16777216	468709344	1.835117	1.781472	1.757029277	1.769326
33554432	938471392	1.879246	1.798371	1.779262303	1.764226

Table 6.13: Execution Time of 4 GPUs BiCGStab with Various Communication Model

Execution Time of 4 GPUs BiCGStab with Various Communication Model

N size	nnz	Execution Time (ms)			
		Async Pinned	Sync Pinned	Sync Paged	Async Paged
4096	110304	8.356	10.327	10.787149	10.307
8192	224992	8.91	11.032	10.513056	10.666
16384	450272	8.61	10.331	10.888618	10.467
32768	900576	8.603	7.311	11.054871	10.656
65536	1818080	8.852	10.917	11.388177	11.162
131072	3636704	9.615	11.857	11.976296	11.083
262144	7273440	10.299	13.122	13.009516	11.976
524288	14613472	11.443	10.787	14.375854	14.63
1048576	29228000	14.34	28.165	18.317841	18.826
2097152	58456032	15.528	23.039	20.171215	20.203
4194304	117176288	26.581	35.387	27.645063	30.629
8388608	234354656	47.264	53.105	48.967529	54.209
16777216	468709344	84.481	90.541	91.484093	90.168
33554432	938471392	151.572	166.07	174.382538	174.411

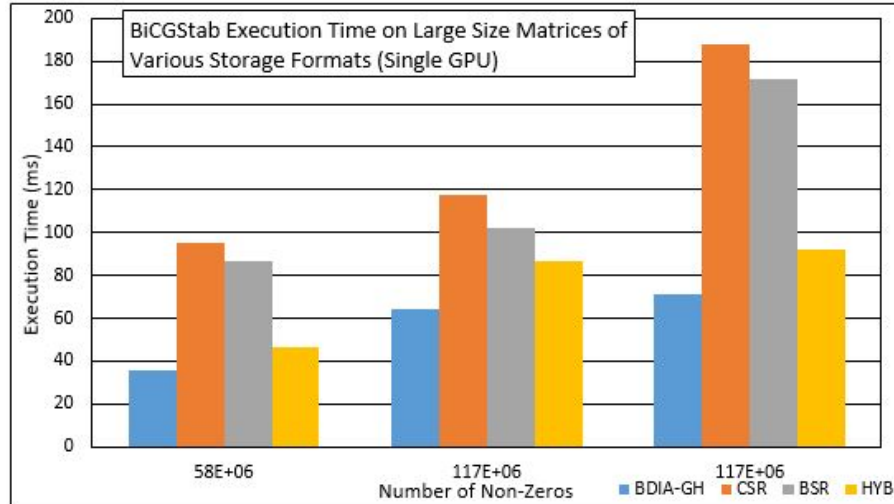


Figure 6.7: BiCGStab Execution Time on Large Size GH Matrices of Various Storage Formats (Single GPU)

is shown by the largest size of the matrix using asynchronous pinned memory communication model.

Table 6.17 and figure 6.30, 6.31, shows that at the small size of matrices, single GPU is performing better than other number of GPUs. This is happened due to the communication of other implementation is larger than the computation. But on the larger size, we could see that 8 GPUs is showing the best performance followed by the 4 GPUs and 2 GPUs. It means that the communication on the larger size of matrix could be amortized by the computation.

Table 6.14: Speedup of 4 GPUs BiCGStab with Various Communication Model over Single GPU

Speedup of 4 GPUs BiCGStab with Various Communication Model over Single GPU					
N size	nnz	Speedup			
		Async Pinned	Sync Pinned	Sync Paged	Async Paged
4096	110304	0.277046	0.22417	0.214607215	0.224605
8192	224992	0.25982	0.209844	0.220202385	0.217045
16384	450272	0.278281	0.231923	0.220046291	0.22891
32768	900576	0.31675	0.372726	0.246497675	0.255724
65536	1818080	0.363082	0.294403	0.282222519	0.287941
131072	3636704	0.413105	0.334992	0.331655129	0.358387
262144	7273440	0.606661	0.476147	0.480263831	0.52171
524288	14613472	0.833785	0.884491	0.663682311	0.652153
1048576	29228000	1.162413	0.591834	0.909987154	0.885424
2097152	58456032	2.034969	1.371544	1.566539249	1.564075
4194304	117176288	2.369663	1.779976	2.278453842	2.056482
8388608	234354656	2.640932	2.350457	2.549056539	2.302588
16777216	468709344	2.948947	2.751571	2.723205662	2.762954
33554432	938471392	3.284056	2.997357	2.854477322	2.854012

Table 6.15: Execution Time of 8 GPUs BiCGStab with Various Communication Model

Execution Time of 8 GPUs BiCGStab with Various Communication Model					
N size	nnz	Execution Time (ms)			
		Async Pinned	Sync Pinned	Sync Paged	Async Paged
4096	110304	14.794	19.16556	17.549328	18.305
8192	224992	14.882	19.00965	17.980619	16.708
16384	450272	14.822	18.83247	18.299026	17.137
32768	900576	14.719	19.26395	18.135403	17.008
65536	1818080	14.729	19.31243	18.465593	18.037
131072	3636704	14.999	20.50044	18.489302	18.393
262144	7273440	16.035	21.3932	19.462868	16.217
524288	14613472	16.939	22.09774	20.958366	24.722
1048576	29228000	18.019	21.16504	53.211971	25.524
2097152	58456032	21.855	29.25381	28.722815	28.076
4194304	117176288	29.699	38.7294	37.445099	38.4
8388608	234354656	44.815	61.90405	59.743706	58.037
16777216	468709344	75.224	86.95006	85.973877	84.644
33554432	938471392	113.458	131.4541	158.420776	132.333



Table 6.16: Speedup of 8 GPUs BiCGStab with Various Communication Model over Single GPU

Speedup of 8 GPUs BiCGStab with Various Communication Model over Single GPU					
N size	nnz	Speedup			
		Async Pinned	Sync Pinned	Sync Paged	Async Paged
4096	110304	0.156482	0.12079	0.131913883	0.126468
8192	224992	0.155557	0.12178	0.128749739	0.138556
16384	450272	0.161652	0.127227	0.130935931	0.139814
32768	900576	0.185135	0.141456	0.150258585	0.160219
65536	1818080	0.218209	0.166421	0.174053441	0.178189
131072	3636704	0.264818	0.193752	0.214826931	0.215952
262144	7273440	0.389648	0.292055	0.321021547	0.385275
524288	14613472	0.563256	0.431764	0.455235871	0.385932
1048576	29228000	0.925079	0.787572	0.313256579	0.653072
2097152	58456032	1.445848	1.080167	1.100135902	1.125481
4194304	117176288	2.120879	1.626361	1.682142702	1.640313
8388608	234354656	2.78525	2.016362	2.089274475	2.150714
16777216	468709344	3.311842	2.865208	2.897740671	2.943268
33554432	938471392	4.387271	3.786652	3.142081566	3.761503

Table 6.17: GFLOPS of BiCGStab on Various Number of GPU(s)

GFLOPS of BiCGStab on Various Number of GPU(s)					
N size	nnz	GFLOPS			
		1 GPU	2 GPU <sub>s</sub>	4 GPU <sub>s</sub>	8 GPU <sub>s</sub>
4096	110304	1.226939	0.894885	0.3399191	0.191994
8192	224992	2.495549	1.171132	0.648394613	0.3882
16384	450272	4.82502	2.348314	1.342711731	0.779972
32768	900576	8.485218	3.213205	2.687692665	1.57091
65536	1818080	14.50432	8.437444	5.266253502	3.164972
131072	3636704	23.47577	15.37947	9.697944878	6.216797
262144	7273440	29.8483	28.66465	18.10779726	11.63032
524288	14613472	39.24635	41.39393	32.72301424	22.10576
1048576	29228000	44.92904	61.86372	52.22609177	41.56291
2097152	58456032	47.40166	75.63346	96.46091164	68.53558
4194304	117176288	47.65198	81.83834	112.9191159	101.0641
8388608	234354656	48.09328	85.83882	127.0110829	133.9518
16777216	468709344	48.19213	88.4382	142.1160304	159.6047
33554432	938471392	48.286	90.74128	158.5739334	211.8438

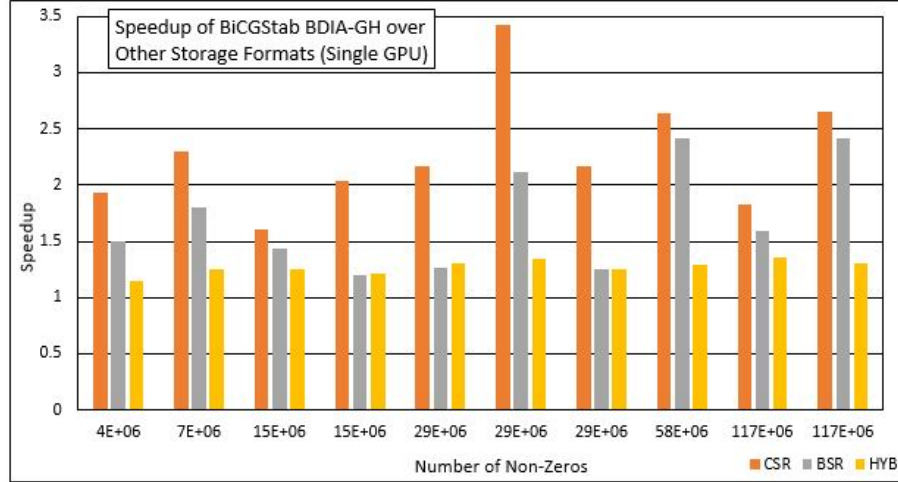


Figure 6.8: Speedup of BiCGStab BDIA-GH over Other Storage Formats (Single GPU)

## 6.5 Overhead in Computing BiCGStab

In order to understand the overhead in computing BiCGStab for multi-GPU implementation, we carried an experiment by running BiCGStab algorithm for one iteration. By using NVIDIA profiling tool, we could retrieve the start time and duration for each kernel calls. Table 6.18 shows the number of kernel that called in one iteration. As we see that SpMV is called 3 times in one iteration of BiCGStab while dot and reduce kernel is called 5 times. These operations are requiring device to host (D2H) communications. So the number of D2H communication is the addition of SpMV kernel and dot kernel, i.e. 8 times. In other hands, host to device (H2D) communication only required while distributing data to each GPUs to do SpMV operations, so the number of H2D communication is 3 times. Copy kernel, axpy, scale, and memset called for 4, 7, 1, and 2 times respectively. The configuration of this experiment that we are using is the largest problem size of our implementation that has about 33.5 million rows (N) and 938.5

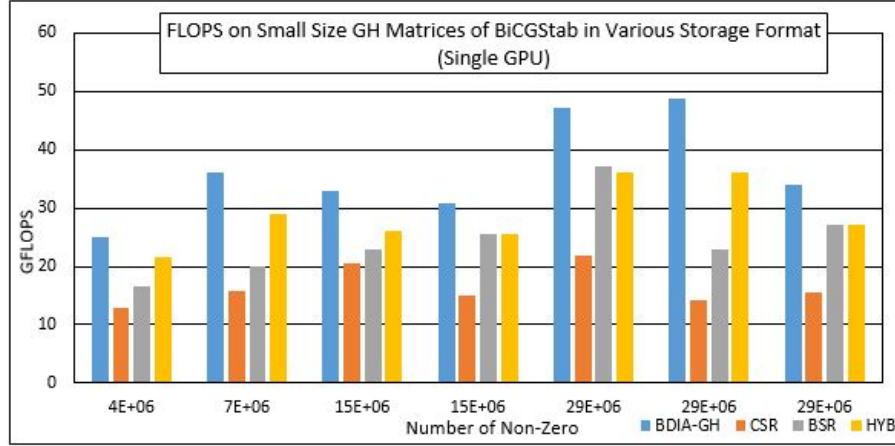


Figure 6.9: FLOPS on Small Size GH Matrices of BiCGStab in Various Storage Format (Single GPU)

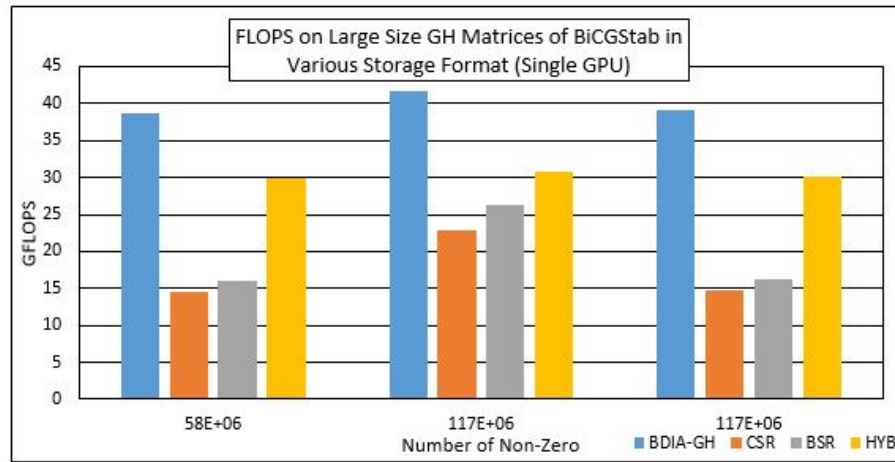


Figure 6.10: FLOPS on Large Size GH Matrices of BiCGStab in Various Storage Format (Single GPU)

million number of non-zero (nnz). From the table 6.19, we could see that the total number of arithmetic operation in one iteration of BiCGStab in this experiment is  $21N + 6nnz - 2$  or about 6.335 billion arithmetic operations.

By analysing the profiling result for one iteration of BiCGStab, we could see that from the table 6.20 and 6.21 the communication time is increasing along with the increasing number of GPUs. For 8 GPUs, the D2H communication takes almost 33 percent of total execution time while 4 GPUs takes about 16 percent

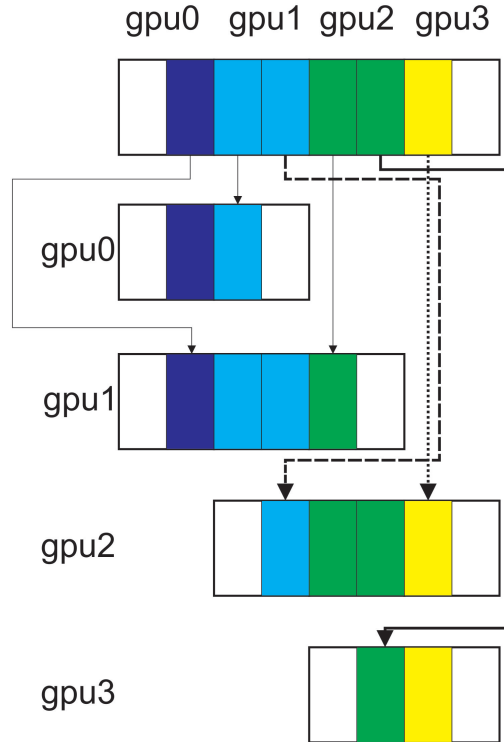


Figure 6.11: Memory transfer routes on 4 GPUs implementation

of total execution time. This very high percentage of communication makes the multi-GPU BiCGStab is not scaled perfectly while we are expecting with the increase number of GPUs should leads to the lower data transfer. But device synchronization on device to host communication is high that leads to the higher communication time. PCI-e channel also takes role in this poor performance of 8 GPUs implementations because the PCI-e that has 4 channel could only take 4 memory transfer from 4 GPUs at a time, so other transfers from other 4 GPUs is getting a low throughput.

Table 6.18: Number of Kernel Calls on BiCGStab

Number of Kernel Calls	
Kernel	Number of Calls
SpMV	3
dot_kernel	5
reduce_1Block_kernel	5
D2H	8
H2D	3
copy_kernel	4
axpy_kernel	7
scal_kernel	1
memset	2

Table 6.19: Number of Arithmetic Operation in BiCGStab

Operation	Number of Arithmetic Operation
$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$	$N+2nnz$
$\rho_i \leftarrow (\hat{\mathbf{r}}_0, \mathbf{r}_{i-1})$	$2N-1$
$\mathbf{p}_i \leftarrow \mathbf{r}_{i-1} + \beta(\mathbf{p}_{i-1} - \omega_{i-1}\mathbf{v}_{i-1})$	$4N$
$\mathbf{v}_i \leftarrow \mathbf{A}\mathbf{p}_i$	$2nnz$
$\alpha \leftarrow \rho_i / (\hat{\mathbf{r}}_0, \mathbf{v}_i)$	$2N$
$\mathbf{s} \leftarrow \mathbf{r}_{i-1} - \alpha\mathbf{v}_i$	$2N$
$\mathbf{t} \leftarrow \mathbf{A}\mathbf{s}$	$2nnz$
$\omega_i \leftarrow (\mathbf{t}, \mathbf{s}) / (\mathbf{t}, \mathbf{t})$	$4N-1$
$\mathbf{x}_i \leftarrow \mathbf{x}_{i-1} + \alpha\mathbf{p}_i + \omega_i\mathbf{s}$	$4N$
$\mathbf{r}_i \leftarrow \mathbf{s} - \omega_i\mathbf{t}$	$2N$

Table 6.20: Kernel Calls Execution Time of BiCGStab on various number of GPUs

Kernel Calls Execution Time (ms)			
Kernel	2 GPUs	4 GPUs	8 GPUs
D2H	4.379223	8.773	11.316
H2D	0.31	1.045	1.313
Memset	0.874	0.439	0.234
SpMV	66.96	33.565	16.674
axpy	10.239593	5.145449	2.588105
copy	4.632	2.352	1.178
dot	3.801	1.918	0.993
reduce	0.022751	0.022592	0.053
scal	1.094	0.551	0.278

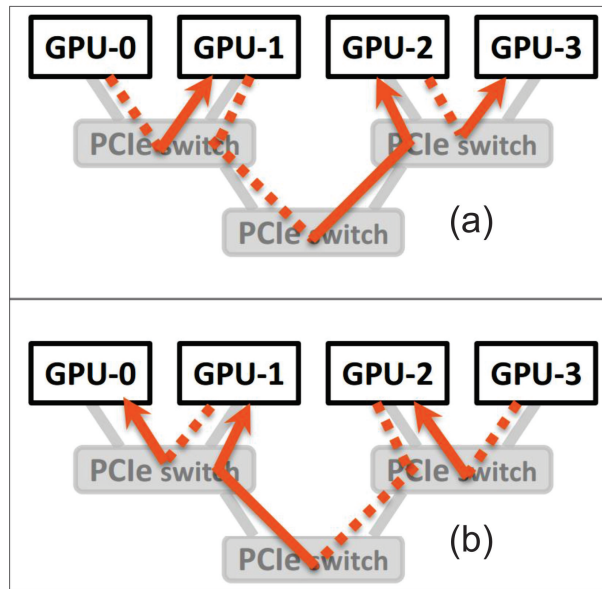


Figure 6.12: Peer-to-Peer GPUDirect 4 GPUs implementation. (a) right communication, (b) left communication

Table 6.21: Kernel Calls Execution Time Percentage of BiCGStab

Kernel Calls Execution Time Percentage (%)			
Kernel	2 GPUs	4 GPUs	8 GPUs
D2H	5.136895391	16.30334563	32.67960172
H2D	0.335815599	1.941980643	3.791827241
Memset	0.946783334	0.815817706	0.675771191
SpMV	72.53616943	62.37567491	48.15302925
axpy	11.09230664	9.562069242	7.474217091
copy	5.017735018	4.370850213	3.401959246
dot	4.117532557	3.564324281	2.867695697
reduce	0.024645615	0.041983949	0.153059287
scal	1.185104082	1.02395343	0.802839279

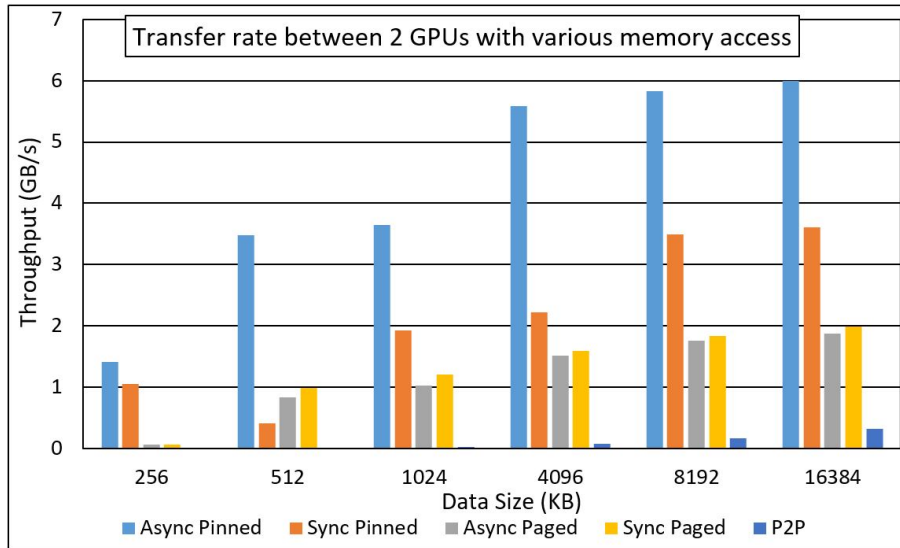


Figure 6.13: Transfer rate between 2 GPUs with various memory access

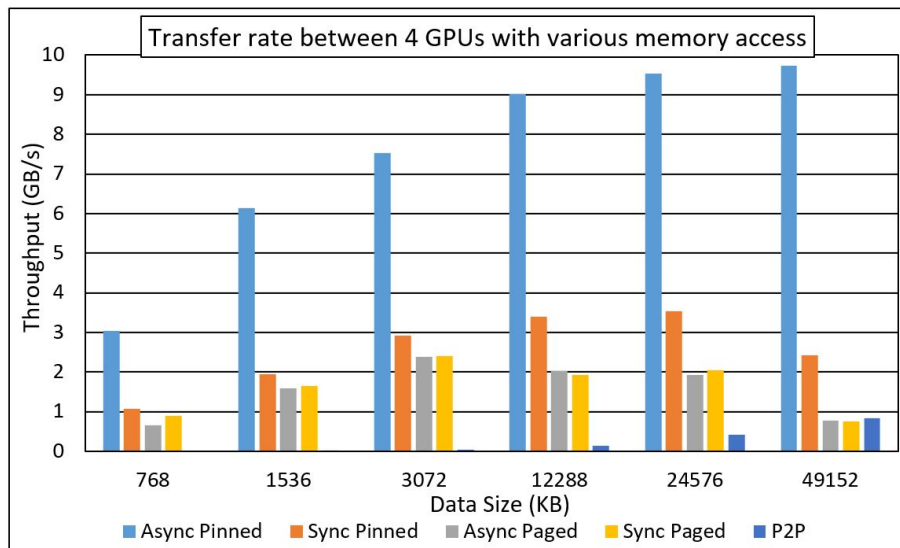


Figure 6.14: Transfer rate between 4 GPUs with various memory access

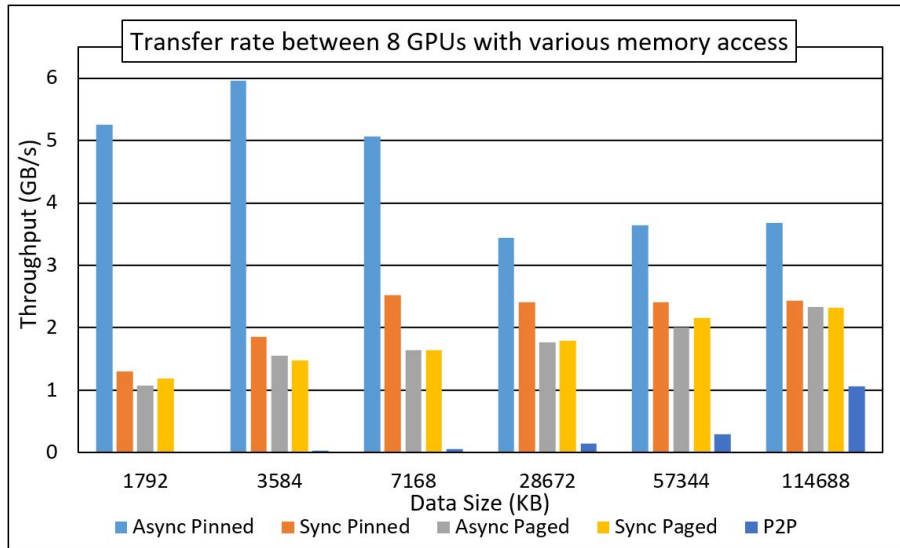


Figure 6.15: Transfer rate between 2 GPUs with various memory access

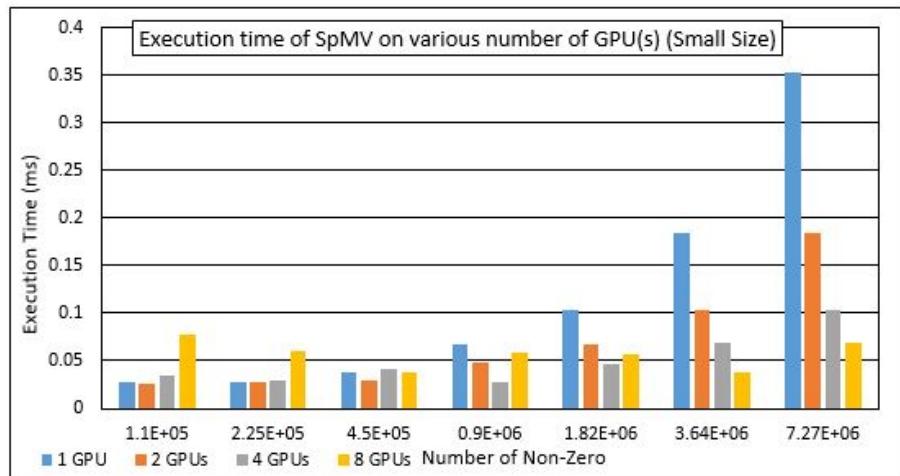


Figure 6.16: Execution time of SpMV on various number of GPU(s) (Small Size)



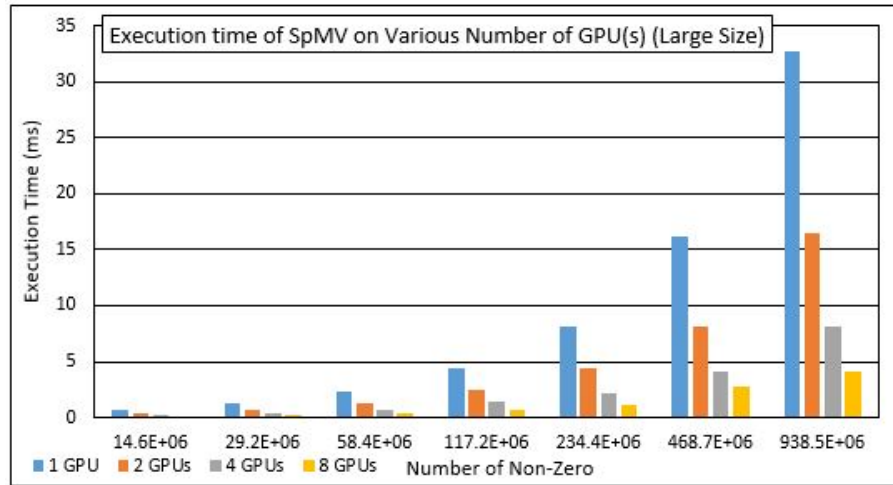


Figure 6.17: Execution time of SpMV on various number of GPU(s) (Large Size)

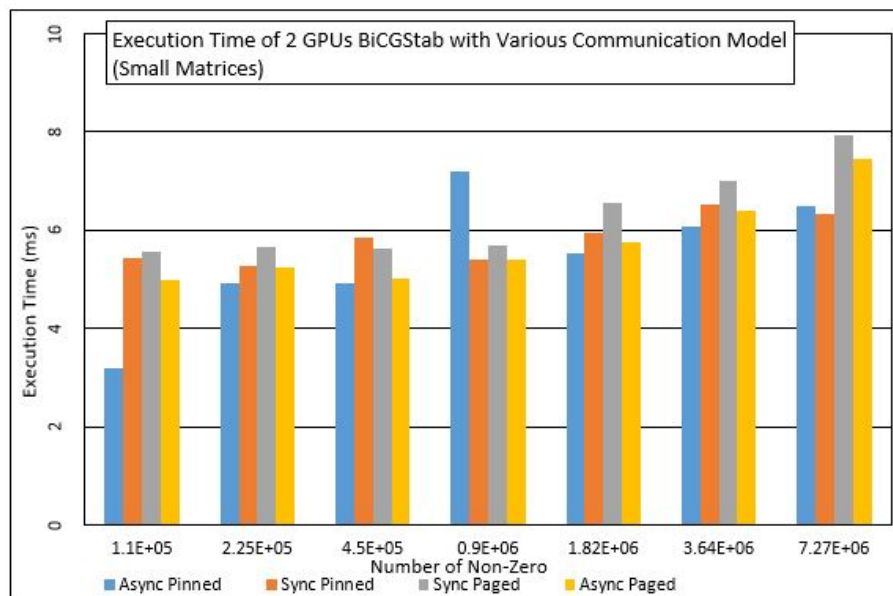


Figure 6.18: Execution Time of 2 GPUs BiCGStab with Various Communication Model (Small Size)

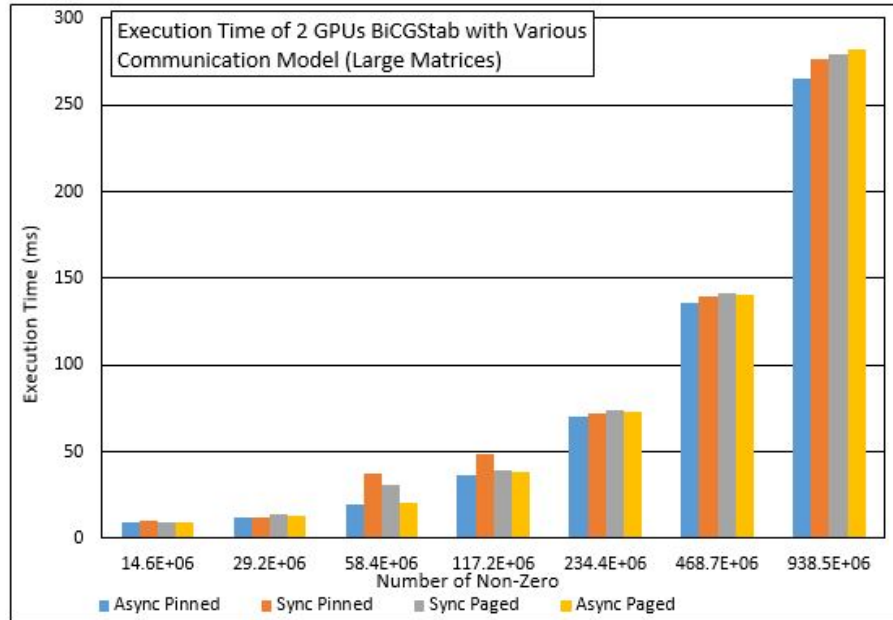


Figure 6.19: Execution Time of 2 GPUs BiCGStab with Various Communication Model (Large Size)

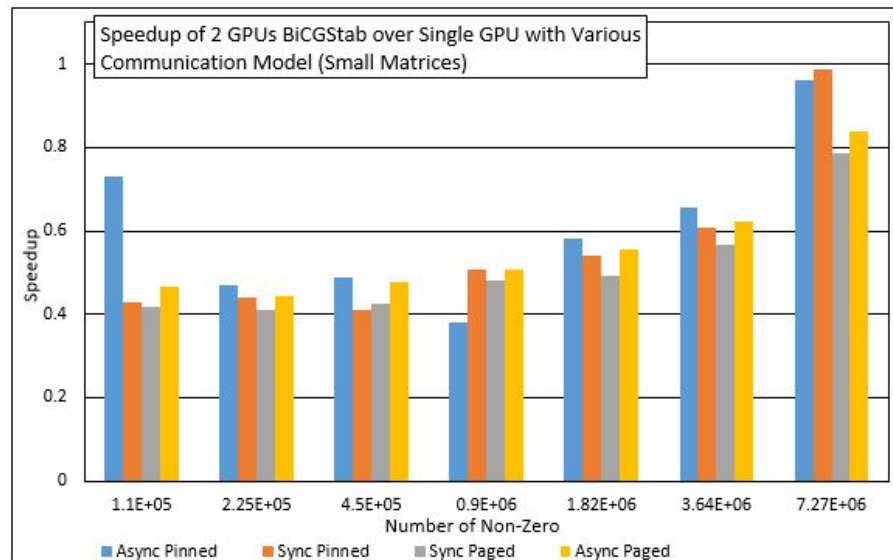


Figure 6.20: Speedup of 2 GPUs BiCGStab over Single GPU with Various Communication Model (Small Size)

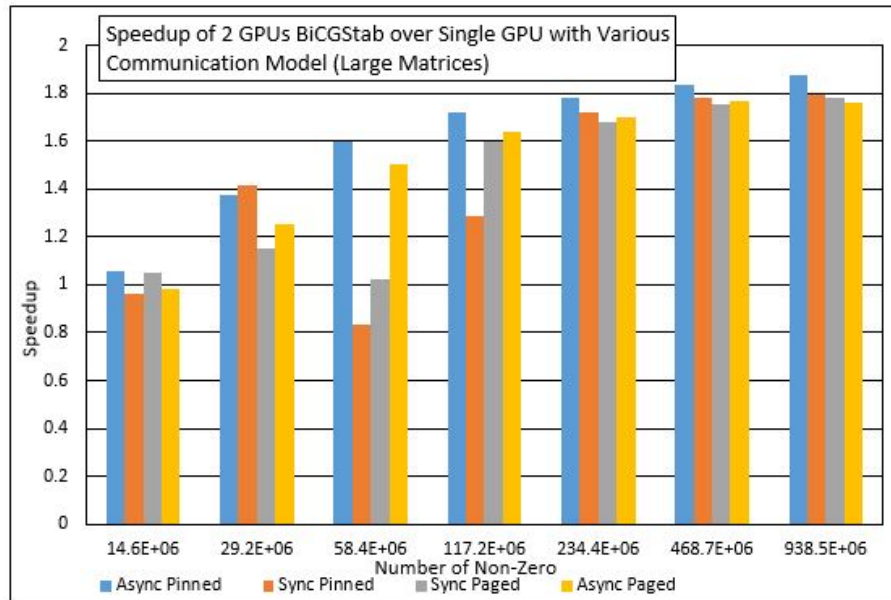


Figure 6.21: Speedup of 2 GPUs BiCGStab over Single GPU with Various Communication Model (Large Size)

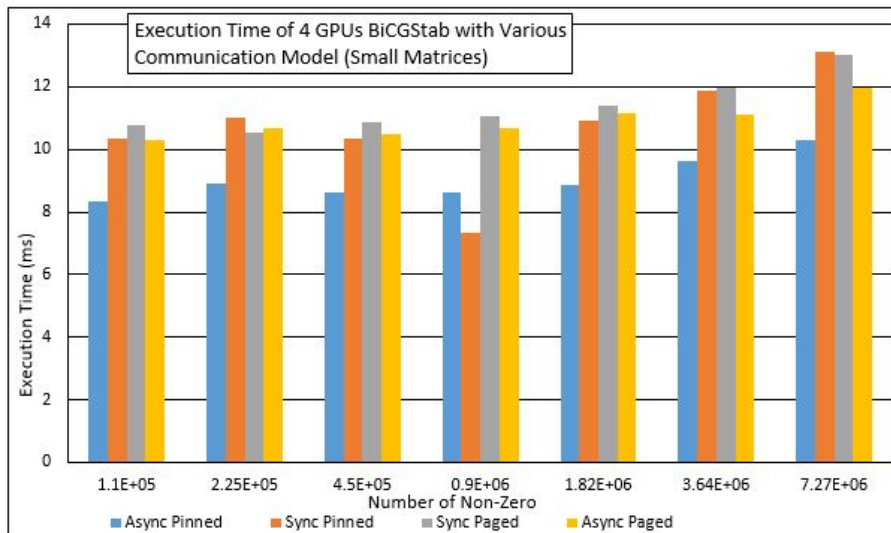


Figure 6.22: Execution Time of 4 GPUs BiCGStab with Various Communication Model (Small Size)

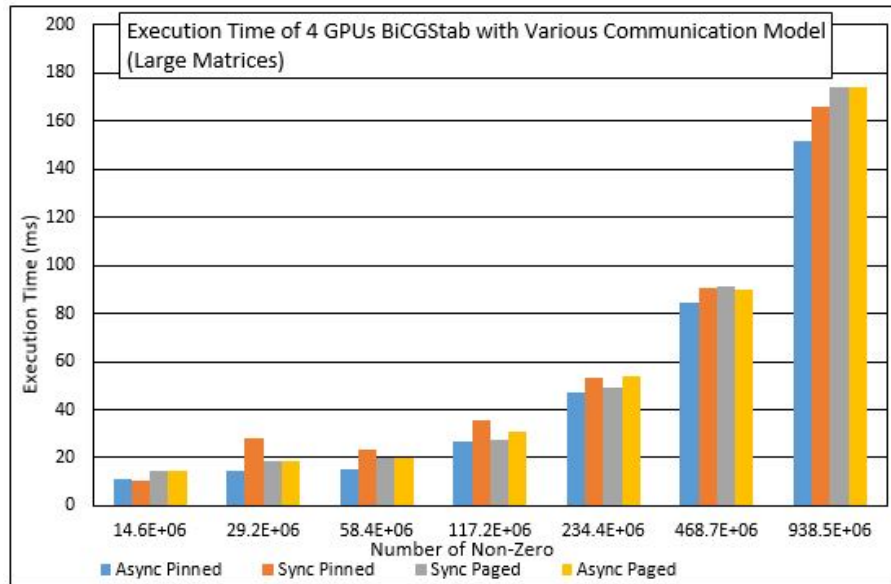


Figure 6.23: Execution Time of 4 GPUs BiCGStab with Various Communication Model (Large Size)

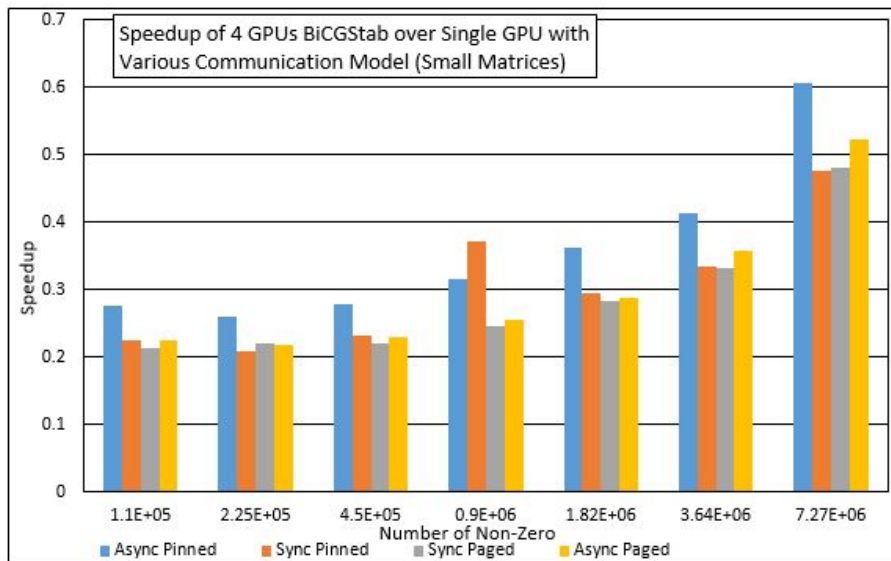


Figure 6.24: Speedup of 4 GPUs BiCGStab over Single GPU with Various Communication Model (Small Size)

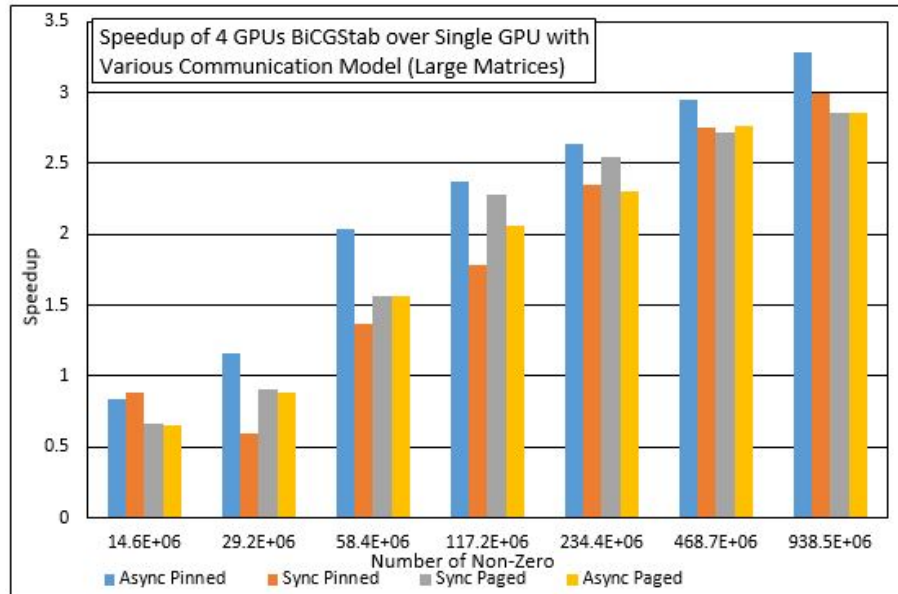


Figure 6.25: Speedup of 4 GPUs BiCGStab over Single GPU with Various Communication Model (Large Size)

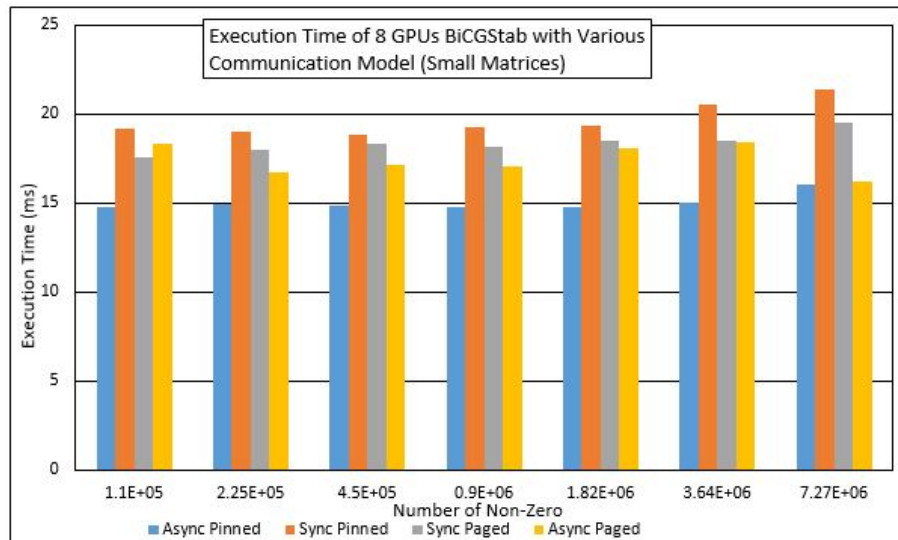


Figure 6.26: Execution Time of 8 GPUs BiCGStab with Various Communication Model (Small Size)

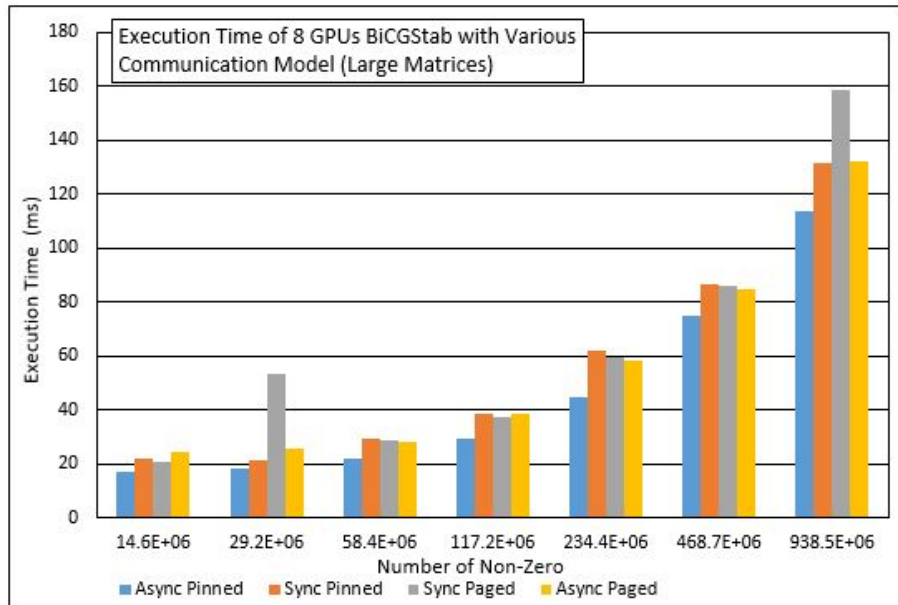


Figure 6.27: Execution Time of 8 GPUs BiCGStab with Various Communication Model (Large Size)

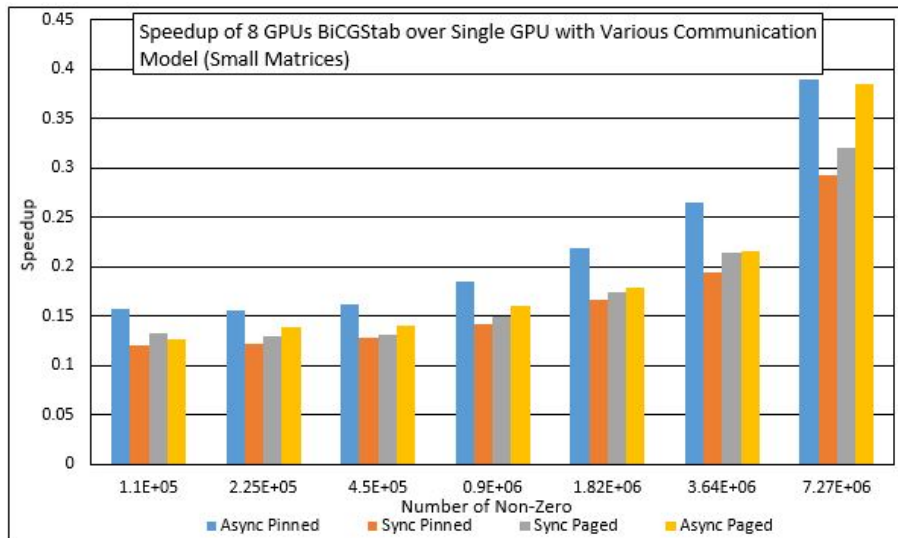


Figure 6.28: Speedup of 8 GPUs BiCGStab over Single GPU with Various Communication Model (Small Size)

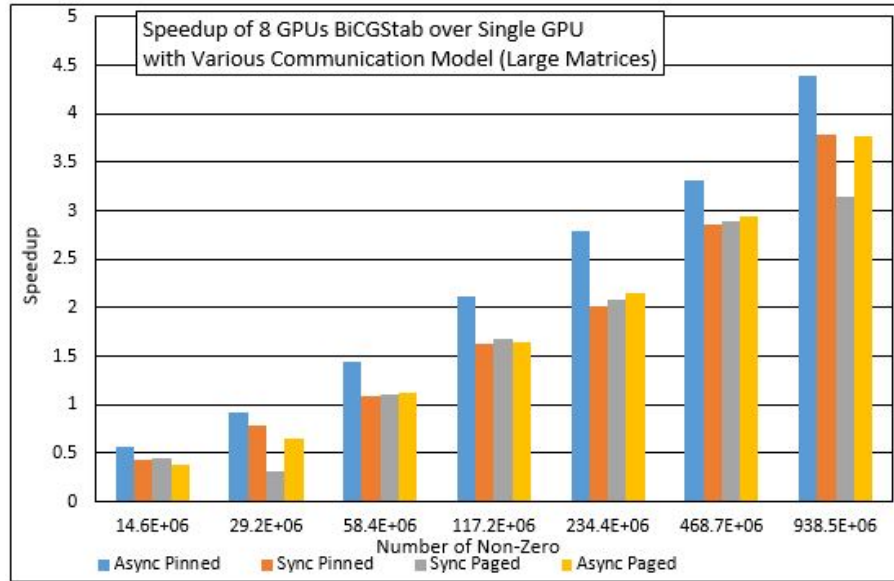


Figure 6.29: Speedup of 8 GPUs BiCGStab over Single GPU with Various Communication Model (Large Size)

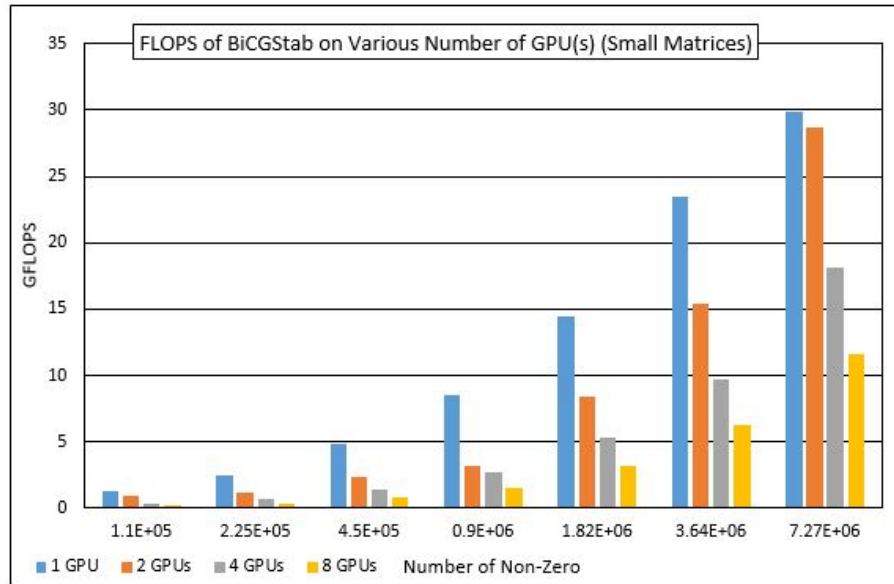


Figure 6.30: FLOPS of BiCGStab on Various Number of GPU(s) (Small Size)

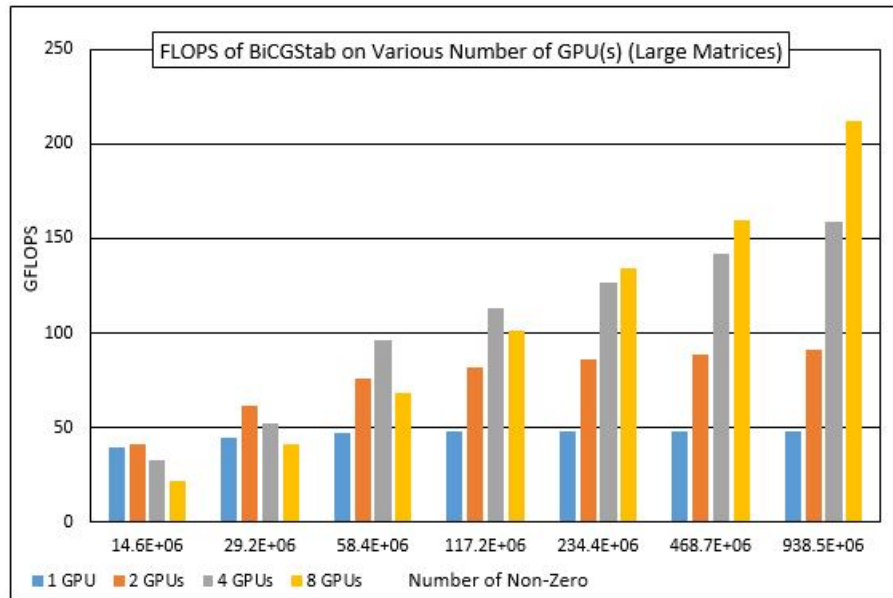


Figure 6.31: FLOPS of BiCGStab on Various Number of GPU(s) (Large Size)

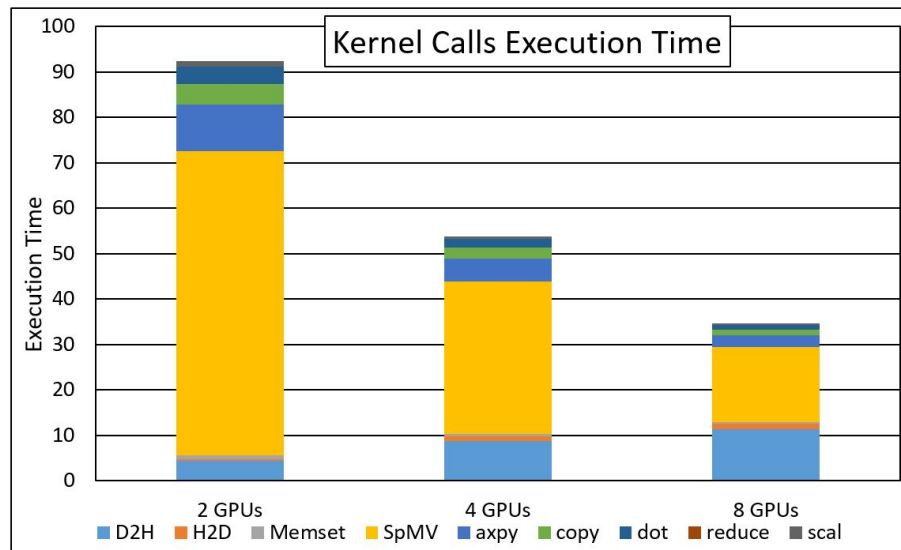


Figure 6.32: Kernel Calls Execution Time of BiCGStab on various number of GPUs



## CHAPTER 7

# CONCLUSION

In this thesis a new sparse matrix storage format called BDIA-GH has been proposed as a storage scheme for sparse matrix on reservoir simulation. This matrix format is proved as the best storage scheme compared to other storage format that available on CUSPARSE library, such as CSR, BSR, and HYB. The sparse matrix vector multiplication for the corresponding format also has been proposed and it showed that BDIA-GH SpMV perform better than other storage formats SpMV using single GPU in CUDA. This better performance on SpMV leads to better performance on BiCGStab algorithm. A novel implementation on Multi-GPU BiCGStab also has been proposed with several optimizations like optimizing shared memory usage, using a best parameter for number of thread size in a block, and choosing memory communication scheme that leads to the better performance on Multi-GPU BiCGStab with CUDA.

# REFERENCES

- [1] Y. Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.
- [2] A.-K. C. Ahamed and F. Magoules, “Iterative Methods for Sparse Linear Systems on Graphics Processing Unit,” *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 836–842, jun 2012. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6332256>
- [3] W. Abu-Sufah and A. A. Karim, “An Effective Approach for Implementing Sparse Matrix-Vector Multiplication on Graphics Processing Units,” *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 453–460, 2012. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6332207>
- [4] J. Í. Aliaga, J. Pérez, E. S. Quintana-Orti, and H. Anzt, “Reformulated conjugate gradient for the energy-Aware solution of linear systems on GPUs,”

- Proceedings of the International Conference on Parallel Processing*, pp. 320–329, 2013.
- [5] H. Anzt, S. Tomov, and J. Dongarra, “Implementing a Sparse Matrix Vector Product for the SELL-C / SELL-C- $\sigma$  formats on NVIDIA GPUs,” 2014.
- [6] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “GPU computing,” *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008. [Online]. Available: <http://ieeexplore.ieee.org/ielx5/5/4490117/04490127.pdf?tp=&arnumber=4490127&isnumber>
- [7] C. B. Moler, *Numerical computing with MATLAB*. SIAM, 2004.
- [8] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *AMC Queue*, vol. 6, no. April, pp. 40–53, 2008. [Online]. Available: [www.acmqueue.com](http://www.acmqueue.com)
- [9] L. Brown, “Deep Learning With GPUs,” *Geoint Symposium*, no. June, 2015.
- [10] NVIDIA, “NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110/210,” pp. 1–24, 2012. [Online]. Available: <https://www.google.co.kr/%5Cnpapers3://publication/uuid/944F91D5-AEDB-4F62-B6F4-BC>
- [11] M. A. Martinez-del Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, J. M. Cecilia, G. D. Guerrero, and J. M. Garcia, “Simulating active membrane systems using gpus,” in *Tenth Workshop on Membrane Computing (WMC10)*. Citeseer, 2009, p. 369.

- [12] A. Acosta, V. Blanco, and F. Almeida, “Towards the Dynamic Load Balancing on Heterogeneous Multi-GPU Systems.” *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 646–653, 2012. [Online]. Available: <http://ieeexplore.ieee.org/document/6280356/>
- [13] S. Schaetz and M. Uecker, “A multi-GPU programming library for real-time applications,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7439 LNCS, no. PART 1, pp. 114–128, 2012. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-33078-0\\_9](http://dx.doi.org/10.1007/978-3-642-33078-0_9)
- [14] H. V. D. A. van der Vorst, “Bi-cgstab: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992. [Online]. Available: <http://epubs.siam.org/doi/abs/10.1137/0913035>
- [15] T. a. Davis and Y. Hu, “The university of Florida sparse matrix collection,” *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.
- [16] J. H. Wilkinson and J. H. Wilkinson, *The algebraic eigenvalue problem*. Clarendon Press Oxford, 1965, vol. 87.
- [17] N. Bell and M. Garland, “Efficient Sparse Matrix-Vector Multiplication on CUDA,” *Nvidia Technical Report*, pp. 1–32, 2008. [Online]. Available: <http://sbel.wisc.edu/Courses/ME964/Literature/techReportGarlandBell.pdf>

- [18] —, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, no. 1, p. 1, 2009. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=1654059.1654078>
- [19] C. Toolkit, “4.0, CUSPARSE Library,” Available at <http://developer.nvidia.com/cuda-toolkit-40>, 2011.
- [20] N. Bell and M. Garland, “Cusp: Generic parallel algorithms for sparse matrix and graph computations,” *Version 0.3. 0*, vol. 35, 2012.
- [21] M. Naumov, L. S. Chien, P. Vandermersch, and U. Kapasi, “Cusp sparse library,” in *GPU Technology Conference*, 2010.
- [22] B. Neelima, G. R. M. Reddy, and P. S. Raghavendra, “Predicting an optimal sparse matrix format for SpMV computation on GPU,” in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 1427–1436.
- [23] K. K. Matam and K. Kothapalli, “Accelerating sparse matrix vector multiplication in iterative methods using GPU,” *Proceedings of the International Conference on Parallel Processing*, pp. 612–621, 2011.
- [24] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore

- platforms,” *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009. [Online]. Available: <http://dx.doi.org/10.1016/j.parco.2008.12.006>
- [25] A. Monakov and A. Avetisyan, “Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs,” in *International Workshop on Embedded Computer Systems*. Springer, 2009, pp. 289–297.
- [26] G. Huan and Z. Qian, “A new method of Sparse Matrix-Vector Multiplication on GPU,” in *Computer Science and Network Technology (ICCSNT), 2012 2nd International Conference on*. IEEE, 2012, pp. 954–958.
- [27] D. Luebke, “CUDA: Scalable parallel programming for high-performance scientific computing,” *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, Proceedings, ISBI*, pp. 836–838, 2008.
- [28] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming - PPOPP '08*, p. 73, 2008. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1345220%5Cnhttp://portal.acm.org/citation.cfm?doid=1345220>
- [29] K.-T. Cheng and Y.-C. Wang, “Using mobile GPU for general-purpose computing—a case study of face recognition on smartphones,” *International*

- Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–4, 2011.
- [30] J. Mak, P. Choboter, and C. Lupo, “Numerical ocean modeling and simulation with CUDA,” *Oceans 2011*, 2011. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=6107199](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6107199)
- [31] N. Lopes and B. Ribeiro, “GPUMLib: An Efficient Open-source GPU Machine Learning Library,” ... *Journal of Computer Information Systems and ...*, vol. 3, no. July 2016, pp. 355–362, 2011. [Online]. Available: <http://www.ece.neu.edu/groups/nucar/NUCARTALKS/GPUMLib.pdf>
- [32] G. Andrade and F. Viegas, “Parallel Implementation and Performance Analysis of a 3D Oil Reservoir Data Visualization Tool on the Cell Broadband Engine and CUDA GPU,” ... *(Sbac-Pad), 2013 ...*, pp. 4–9, 2013. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=6702594](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6702594)
- [33] T. Ertekin, J. Abou-Kassem, and G. King, “Basic practical reservoir simulation,” *SPE Textbook Series*, vol. 7, 2001.
- [34] R. S. Wright Jr, N. Haemel, G. M. Sellers, and B. Lipchak, *OpenGL SuperBible: comprehensive tutorial and reference*. Pearson Education, 2010.
- [35] F. Dunn and I. Parberry, *3D math primer for graphics and game development*. CRC Press, 2015.
- [36] I. Kerr, ““Camera Class Tutorial,” *Nehe Productions: OpenGL Article*, vol. 8, 2003.

- [37] H. Klie, H. Sudan, and R. Li, “Exploiting Capabilities of Many Core Platforms in Reservoir Simulation,” *SPE Reservoir Simulation*, 2011. [Online]. Available: <http://www.onepetro.org/mslib/servlet/onepetroreview?id=SPE-141265-MS&soc=SPE>
- [38] R. Li and Y. Saad, “GPU-accelerated preconditioned iterative linear solvers,” *Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.
- [39] J. Bergstra, O. Breuleux, F. F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, “Theano: a CPU and GPU math compiler in Python,” *Proceedings of the Python for Scientific Computing Conference (SciPy)*, no. Scipy, pp. 1–7, 2010. [Online]. Available: [http://www-etud.iro.umontreal.ca/wardefar/publications/theano\\_scipy2010.pdf](http://www-etud.iro.umontreal.ca/wardefar/publications/theano_scipy2010.pdf)
- [40] Y. Lin, C. Lin, and D. Lou, “Efficient Parallel RSA Decryption Algorithm for Many-core GPUs with CUDA,” *Koala.Cs.Pub.Ro*. [Online]. Available: [https://koala.cs.pub.ro/redmine/attachments/download/1746/cuda\\_rsa\\_decrypt.pdf](https://koala.cs.pub.ro/redmine/attachments/download/1746/cuda_rsa_decrypt.pdf)
- [41] P. Micikevicius, “Multi-GPU Programming,” 2011.
- [42] —, “3D finite difference computation on GPUs using CUDA,” *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*, pp. 79–84, 2009. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1513895.1513905>



- [43] T. Xu, T. Pototschnig, K. Kühnlenz, and M. Buss, “A high-speed multi-GPU implementation of bottom-Up attention using CUDA,” *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 41–47, 2009.
- [44] P. Vidal and E. Alba, “A multi-GPU implementation of a Cellular Genetic Algorithm,” *2010 IEEE World Congress on Computational Intelligence, WCCI 2010 - 2010 IEEE Congress on Evolutionary Computation, CEC 2010*, 2010.
- [45] J. C. Thibault and I. Senocak, “Accelerating incompressible flow computations with a Pthreads-CUDA implementation on small-footprint multi-GPU platforms,” *Journal of Supercomputing*, vol. 59, no. 2, pp. 693–719, 2012.
- [46] J. Guan, S. Yan, and J. M. Jin, “An OpenMP-CUDA implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-GPU computing systems,” *IEEE Transactions on Antennas and Propagation*, vol. 61, no. 7, pp. 3607–3616, 2013.
- [47] V. Boyer, D. El Baz, and M. Elkihel, “Dense dynamic programming on multi GPU,” *Proceedings - 19th International Euromicro Conference on Parallel, Distributed, and Network-Based Processing, PDP 2011*, pp. 545–551, 2011.
- [48] M. Sourouri, T. Gillberg, S. B. Baden, and X. Cai, “Effective multi-GPU communication using multiple CUDA streams and threads,” *Proceedings of*

- the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2015-April, pp. 981–986, 2014.
- [49] D. Q. Ren and R. Suda, “Investigation on the power efficiency of multi-core and GPU processing element in large scale SIMD Computation with CUDA,” *2010 International Conference on Green Computing, Green Comp 2010*, pp. 309–316, 2010.
- [50] Byunghyun Jang, D. Kaeli, Synho Do, and H. Pien, “Multi GPU implementation of iterative tomographic reconstruction algorithms,” in *2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. IEEE, jun 2009, pp. 185–188. [Online]. Available: <http://ieeexplore.ieee.org/document/5193014/>
- [51] M. E. Lalami, D. El-Baz, and V. Boyer, “Multi GPU implementation of the simplex algorithm,” *Proc.- 2011 IEEE International Conference on HPCC 2011 - 2011 IEEE International Workshop on FTDCS 2011 -Workshops of the 2011 Int. Conf. on UIC 2011- Workshops of the 2011 Int. Conf. ATC 2011*, pp. 179–186, 2011.
- [52] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, “Multi-GPU and multi-CPU parallelization for interactive physics simulations,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6272 LNCS, no. PART 2, pp. 235–246, 2010.

- [53] M. Ament, G. Knittel, D. Weiskopf, and W. Straßer, “A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform,” *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010*, pp. 583–592, 2010.
- [54] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*. NBS, 1952, vol. 49.
- [55] J. R. Shewchuk and Others, “An introduction to the conjugate gradient method without the agonizing pain,” 1994.
- [56] N. P. Karunadasa and D. N. Ranasinghe, “Accelerating high performance applications with CUDA and MPI,” *ICIIS 2009 - 4th International Conference on Industrial and Information Systems 2009, Conference Proceedings*, no. December, pp. 331–336, 2009.
- [57] A. Cevahir, A. Nukada, and S. Matsuoka, “High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning,” *Computer Science - Research and Development*, vol. 25, no. 1-2, pp. 83–91, 2010.
- [58] A. Zaza, A. A. Awotunde, F. A. Fairag, and M. A. Al-Mouhamed, “A CUDA based parallel multi-phase oil reservoir simulator,” *Computer Physics Communications*, vol. 206, pp. 2–16, 2016.

- [59] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, “Understanding the Performance of Sparse Matrix-Vector Multiplication,” *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pp. 283–292, 2008.
- [60] P. Micikevicius, “Multi-GPU programming,” *GPU Computing Webinars*, NVIDIA, 2011.

# Vitae

- Name: Lutfi Aziz Firdaus
- Nationality: Indonesia
- Date of Birth: 27<sup>th</sup> February 1990
- Email: *lutfi.afirdaus@gmail.com*
- Permenant Address: Jl. P.H.H. Mustopa no. 24, Bandung, Indonesia
- Academic Background: Bachelor Engineering in Electrical Engineering from Institut Teknologi Bandung, Bandung, Indonesia