2015-09-28

A GMRES Solver with ILU(k) Preconditioner for Large-Scale Sparse Linear Systems on Multiple GPUs

Yang, Bo

UNIVERSITY OF CALGARY

A GMRES Solver with ILU(k) Preconditioner for Large-Scale Sparse Linear Systems on Multiple GPUs

by

Bo Yang

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN CHEMICAL AND PETROLEUM ENGINEERING

CALGARY, ALBERTA

SEPTEMBER, 2015

Abstract

Most time of reservoir simulation is spent on the solution of large-scale sparse linear systems. The Krylov subspace solvers and the ILU preconditioners are the most commonly used methods for solving such systems. Based on excellent parallel computing performance, GPUs have been a promising hardware architecture. The work of developing preconditioned Krylov solvers on GPUs is necessary and challengeable. We devote our efforts into the development of the GMRES and the ILU(k) preconditioner on a multiple-GPU architecture and achieve favorable speedup effects. Our GPU computation includes the algorithms such as SPMV, nested RAS, decoupled ILU(k) and parallel triangular solver, etc. The numerical experiments prove that our preconditioned GMRES algorithm is feasible and works well on a multiple-GPU workstation.

Acknowledgements

I would like to show my sincerest gratitude to my supervisor Dr. Zhangxing (John) Chen. He has rigorous attitude of scholarship and kind manner to all his students. I am very lucky to be one of his students and obtain the attractive research opportunity. Under my impression, he is always full of energy and eager to work with his full heart. With his detailed concern, direction and guidance, I made much progress in my study and research in the past two and a half years. He sets such a good example for me to follow in the future academic career.

My deepest thanks go to my academic collaborator and good friend Dr. Hui Liu. He is a talent scholar with outstanding academic knowledge. In my research process, I acquired a lot of direction and suggestion from him. He not only specializes in academic research fields but also has an aptitude for computer programming. It is one of the most comfortable things to study his state of the art codes. He is always kind and patient to answer my questions. It is very happy and lucky to work with him.

I am very grateful to my examining committee members, Drs. S.M. Farouq Ali, Hossein Hejazi, and Wenyuan Liao for their time, attention and comments.

I owe a lot to my wife. Without her comprehension, support and care, I would not have given my heart and soul into my study and research. She is also a positive and diligent person. I hope she will be success in pursuing her academic degree. I am indebted beyond hope of repayment to my family back home.

I also want to express my great thanks to all members of the Department of Chemical and Petroleum Engineering, NSERC/AIEE/Foundation CMG and AITF Chairs, and our Reservoir Simulation Group.

Finally, I deeply appreciate all my friends who have assisted me in my study, research and life.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
AMG	Algebraic Multigrid
API	Application Programming Interface
BiCG	Biconjugate Gradient
BiCGSTAB	Biconjugate Gradient Stabilized
BLAS	Basic Linear Algebra Subprograms
CPR	Constrained Pressure Residual
CPU	Central Processing Unit
CSR	Compressed Sparse Row
cuBLAS	The NVIDIA CUDA Basic Linear Algebra Subroutines
CUDA	Compute Unified Device Architecture
FDM	Finite Difference Method
FEM	Finite Element Method
FVM	Finite Volume Method
GFlops	One billion (10^9) floating point operations per second
GMRES	Generalized Minimal Residual Method
GPU	Graphical Processing Unit
GS	Gram-Schmidt
HEC	Hybrid of ELL and CSR
HYB	Hybrid of ELL and COO
ILU	Incomplete LU factorization
MGS	Modified Gram-Schmidt
nvcc	NVIDIA CUDA Compiler
OpenMP	Open Multi-Processing

PCIe	Peripheral Component Interface Express
PDE	Partial Differential Equation
RAS	Restricted Additive Schwarz
SPMV	Sparse Matrix-Vector Multiplication
TFlops	One Trillion (10^{12}) floating point operations per second

Chapter 1

OVERVIEW

1.1 Introduction

In the petroleum industry, reservoir simulation has a wide application. A piece of simulation software can be used for running reservoir development and production cases under various operation conditions. A classical reservoir simulation process can be divided into four parts which are physical modeling, nonlinear partial differential equations, nonlinear equations and linear algebraic equations [1]. Eventually, we need to solve the linear algebraic equations which form a linear system. Generally, a case of reservoir simulation runs for hours, days or even longer, which is a time-consuming process. In addition, over 70% of time is spent on the solution of linear systems derived from the Newton methods for the black oil model [2]. For some large and highly heterogeneous geological models [3, 4, 5, 6], their linear systems are even harder to solve and require more simulation time [2].

The coefficient matrices of linear systems in petroleum simulation are generally square and nonsingular. Moreover, they have two distinctive characteristics. One characteristic is their large scale. The number of their rows can be over one million. Fox example, a three-dimensional Poisson equation on a grid of $200 \times 200 \times 200$ has $8,000,000$ rows and columns. If it is stored into a computer in a CSR (Compressed Sparse Row) format, it may occupy 668 Mega Bytes in hard disk storage or in memory space. The other characteristic is their sparseness. This means that only a few nonzero elements exist in each row and most elements are zero. For the same example, the Poisson equation has seven nonzero elements in a row on average.

The Krylov subspace solvers and ILU (Incomplete LU factorization) preconditioners have been studied for decades. They are the most commonly used methods for solving large-scale spare linear systems [17, 7, 28, 37]. The Krylov subspace methods are iterative methods and have been

successfully used in numerical computation fields. The GMRES (Generalized Minimal Residual) method is developed by Saad et al. and used for unsymmetric linear systems [17, 7]. The BiCGSTAB (Biconjugate Gradient Stabilized) method is also a widely used solver in reservoir simulations. The ORTHOMIN (Orthogonal Minimum Residual) solver is originally designed by Vinsome for reservor simulations [18]. Both the robustness and efficiency of iterative solvers can be optimized by using preconditioners. Many advanced preconditioners are also developed, such as domain decomposition methods [18], multi-stage preconditioners [2, 20, 8], a residual (CPR) preconditioner [21, 22, 49] and a fast auxiliary space preconditioning method (FASP) [20]. The ILU preconditioner is one of the most commonly used preconditioners [17, 7, 26, 28, 37]. In our study, the solver is developed on the base of the GMRES algorithm. The preconditioner is developed on the base of ILU(0) and ILU(k).

Nowadays, GPU (Graphical Processing Unit) computing has been popular in various scientific computing applications due to its superiority over conventional CPU (Central Processing Unit) computing. For example, NVIDIA Tesla K80 GPU Accelerator has a peak performance of 2910 GFlops in double precision while a high performance CPU, Intel Core i7-5960X Processor Extreme Edition, has only a typical peak performance of 385 GFlops [57, 58]. In addition, the memory speed of a GPU architecture is also faster than that of a CPU architecture. Using the same example, NVIDIA Tesla K80 GPU Accelerator has 480 GB/s but the Intel Core i7-5960X Processor Extreme Edition has only 68 GB/s [57, 59]. Therefore, it is possible to design a program that is much faster on a GPU architecture than on a CPU carchitecture.

A typical large-scale sparse matrix may contain millions of rows and occupies much storage and memory. Although a single GPU can accelerate SPMV, its floating performance and memory space are still limited. A multiple-GPU architecture in a single node includes a CPU and several GPUs which are installed on the same motherboard. If a multiple-GPU architecture is employed, better performance and shorter calculation time can be expected. However, a multiple-GPU architecture is very different from a single GPU architecture. Because it is not just a simple addition of

several GPUs but has specified communicating mechanism, an algorithm for a single GPU cannot be transplanted into multiple GPUs directly. Therefore, specific algorithms must be investigated and written for multiple-GPU architecture. In our study, we devote our efforts to design the algorithm and implementation of a GMRES solver and an ILU(k) preconditioner on multiple GPUs.

The GMRES was developed by Saad and Schultz in 1986 [55]. It belongs to the Krylov subspace family, and is an iterative method for solving nonsymmetric linear systems. Its solution is approximated by a minimal residual vector which can be found by the Arnoldi iteration in a Krylov subsapce. A GMRES algorithm can be divided into five types of basic operations to implement, which are: (1) Preconditioner setup; (2) Matrix-vector multiplications; (3) Vector updates; (4) Dot products; and (5) Preconditioning operations [7]. The vector updates and dot products are relatively easy to implement. The rest three operations are the crucial parts of the preconditioned GMRES solver. The implementation of these parts on multiple-GPU architecture, which is our major contribution, is described in detail in this thesis.

In our study, the SPMV and corresponding communication mechanism are designed according the characteristics of a multiple-GPU architecture. A multiple-GPU architecture needs the GPUs and the CPU to cooperate mutually. It imports much communication loading caused by GPUs receiving tasks from the CPU and then sending results back to it. These communications pass through a PCIe interface. Though the communication speed of PCIe is always high, the latency of PCIe is the virtual bottleneck of the transmission speed. Therefore, unnecessary communications between GPUs and the CPU must be avoided. This leads to that each GPU must store enough data on its memory space and try to reduce the communication with the CPU. However, based on the limited memory space of a GPU, it is hard for each GPU to possess an entire copy of a matrix and a vector to compute a SPMV. A partition should be a feasible scheme to break through dilemma. In our study, we partition a SPMV into entirely parallel subtasks. Both a matrix and a vector are divided into sub-parts and distributed onto multiple GPUs. We design a local compact method to give each submatrix a set of local column indices and obtain a compact effect which can reduce

GPU memory usage. We design a special communication mechanism for subvectors to share data among multiple GPUs through a buffer on the CPU. The data transmission between the GPUs and the CPU are avoided as much as possible at the running time and only slight communication is needed. A series of numerical experiments are carried out to test the efficiency of the SPMV algorithm. These experimental results show that the SPMV implementation mechanism is efficient and suitable for general sparse matrices on multiple-GPU architecture. The results also indicate that the algorithm has favorable scalability with the number of GPUs.

In order to use an ILU preconditioner on a multiple-GPU architecture, we implement a nested RAS framework by domain decomposition. Multiple GPUs architecture can provide coarse-grained parallelization at multiple-GPU level and fine-grained parallelization at GPU-thread level. It consists of a two-level nested RAS which is composed of an outer RAS and an inner RAS. In the outer RAS, the original coefficient matrix of a linear system is divided into outer domain matrices by domain decomposition. Because most nonzero elements are concentrated on the diagonal blocks of outer domain matrices, other nonzero elements outside the diagonal blocks are sparse and can be discarded according to the RAS theory [24]. The degree of parallelization is improved by RAS because the diagonal blocks are independent and the relationship between them is cut by discarding the nonzero elements outside them. However, the solution accuracy of solving each block may decrease due to the loss of elements. In order to compensate for the calculation accuracy reduction, a multi-layer overlap technique is employed on these adjusted blocks. Each overlapped block carries enough data for calculation without communication with others. These outer overlapped blocks are distributed on multiple GPUs and can be solved simultaneously on them.

Each GPU owns an outer overlapped block. The block can be solved at fine-grained GPU-thread level. If some rows of the block contain a portion of the same unknowns, they have a tight relationship and cannot be solved simultaneously. The purpose of using GPUs is to maximize parallel computing capability. In order to enhance the parallel performance of solving an outer

overlapped block on a single GPU, an inner RAS is necessary. First, domain decomposition is applied to divide an outer overlapped block into inner domain matrices. The nonzero elements outside the inner diagonal blocks represent the relationship among these inner domain matrices and they can also be removed because they are sparse. The remaining diagonal blocks are all square matrices and can be solved concurrently. The accuracy of solving preconditioned systems at each iteration decreases again because some nonzero elements are removed. The multi-layer overlap technique can also be used to recover the accuracy. Therefore, each overlapped inner block carries enough data for calculation and can be calculated simultaneously with other inner overlapped blocks at GPU-thread level. This procedure is called the inner RAS.

The nested RAS framework performance can be configured by four parameters. They are the number of outer RAS (the number of outer blocks or the number of GPUs), the number of inner RAS (the number of inner blocks), the number of outer overlap layers and the number of inner overlap layers. A nested RAS framework is established only once when solving a linear system. The whole framework is deployed on multiple GPUs. Sub-tasks carry proper data and each GPU does not need to communicate with the CPU or other GPUs during the running time. Therefore, the data traffic loads are reduced to the minimum.

The ILU preconditioner is one of the most commonly used preconditioners. We use a decoupled ILU(k) mechanism to implement the ILU(k) preconditioner for a GMRES solver. Assuming that A represents the coefficient matrix of a linear system, a preconditioner system $M\vec{x} = \vec{b}$ is supposed to be solved at least once in each iteration of a Krylov solver. The preconditioner matrix can be expressed as $M = LU$ by ILU, where L and U are a lower and an upper triangular matrix, respectively. Then the preconditioner system can be easily decomposed into $L\vec{y} = \vec{b}$ and $U\vec{x} = \vec{y}$ which will be solved by a triangular solver. L and U can be obtained through an incomplete factorization from the matrix A. The ILU factorization algorithm derives from Gaussian elimination by applying a nonzero pattern P as a factorization filter, which is rather inexpensive to compute when P is sparse. If P has the same nonzero pattern as A, the factorization is called ILU(0). Because A is

a sparse matrix, sometimes the nonzero pattern cannot provide enough nonzero positions for L and U, which will decrease the rate of convergence. An improved method is ILU(k) that allows additional fill-in positions added to P and more accurate factorizations can be acquired. The variable level k is designed to control the extent of fill-in. A higher k allows more fill-in.

The ILU(k) algorithm consists of two tasks. One is the creation of a fill-in nonzero pattern P. The other is the ILU factorization using the pattern P. We call the first task a symbolic phase and the second task a factorization phase. Because the procedure of establishing P has nothing to do with the concrete data values of the original matrix A, a better way is to separate the symbolic phase from the ILU(k) algorithm and design it solely. After extracting the symbolic phase, the rest of the algorithm looks like ILU(0), which uses the fill-in nonzero pattern P to make an ILU factorization on the original matrix A. The ILU(0) algorithm requires the nonzero pattern comes from A. In our algorithm, matrix A is saved by data structure CSR format which only contains the nonzero data and positions of them. If we use zero values to fill the fill-in positions in the matrix A, it will have the same non-zero pattern as P without any essential value changes. Thus the ILU(0) algorithm can be applied to the modified A directly.

Each GPU owns a set of ILU outcomes of the nested RAS and a set of corresponding overlapped right-hand side domain vectors. They form the preconditioner systems to be solved. We use a parallel triangular solver to solve it [51]. The principle of the parallel triangular solver is the level schedule method [7, 28]. Its idea is to group unknowns x_i (ith unknown) into different levels so that all the unknowns within the same level can be computed simultaneously [7, 28]. Because the parallel triangular solver utilizes the fine-grained parallelism on a single GPU, we need to combine all the inner overlapped lower triangular matrices head to tail together to form an entire lower triangular matrix for the single GPU. The same procedure is also done for the inner overlapped upper triangular matrices. Thus these preconditioner systems can be solved on each GPU in parallel.

As stated above, the preconditioned GMRES solver is completed after all the basic operations are implemented. In our test experiments, the preconditioned GMRES solver on multiple-GPU

architecture shows high speedup performance against a single GPU or a CPU.

The layout of the thesis is as follows: In Chapter 2, the background about this research is presented. In Chapter 3, the GMRES method is described in detail and the basic operations of its implementation is analyzed. In Chapter 4, the GPU computation and corresponding algorithms of the GMRES with ILU(k) are described. In Chapter 5, numerical experiments and result analysis are presented. In Chapter 6, conclusions and future work are provided.

1.2 Literature Review

There have been a number of previous studies related to investigating the numerical solutions of large-scale sparse systems, which include various topics such as solver algorithms, preconditioners, matrix formats and their implementations on different hardware architectures. The Krylov iterative linear solvers and different preconditioners have been studied for decades. The GMRES and BiCGSTAB are the two commonly used Krylov subspace methods. Many advanced preconditioners are also developed, such as domain decomposition methods, multi-stage preconditioners, a residual (CPR) preconditioner and a fast auxiliary space preconditioning method (FASP). The Incomplete LU factorization (ILU) preconditioner is one of the most commonly used preconditioners. As GPUs have more superiority over CPUs, some solvers, preconditioners and matrix formats for solving large-scale linear systems have been developed on GPUs. This section gives a review of them.

The Lanczos algorithm is an iterative algorithm designed by Lanczos [68] in 1950. There are many variations of the Lanczos algorithm widely used. The minimal residual method (MINRES) is a variant of the Lanczos method, which can be applied to symmetric indefinite systems. It is developed by Paige and Saunders in 1975 [67]. Saad and Schultz generalized the MINRES method to the GMRES method in 1986 [55]. The GMRES can be applied to a nonsymmetric system. It is one of the most common iterative methods in industry nowadays. The detailed GMRES can be found in [7, 17].

In 1952, Hestenes et al. proposed the conjugate gradient method (CG) [64] which can be derived from the Lanczos algorithm. It is often implemented as an iterative method and suitable for solving a large-scale sparse symmetric and positive-definite matrix. Fletcher generalized the CG to BiCG for non-symmetric matrices in 1976 [65]. Then van der Vorst developed the BiCGSTAB method for nonsymmetric linear systems in 1992 [66]. As a variant of the BiCG, the BiCGSTAB has faster and smoother convergence than the original BiCG and other variants of BiCG, such as the Conjugate Gradient Squared method (CGS). The BiCGSTAB is also a sort of Krylov subsapce method and widely used solver in reservoir simulations.

Multigrid methods are effectively used for systems which have positive definite coefficient matrices. They are categorized into Algebraic Multigrid methods (AMG) and Geometric Multigrid methods. They can provide optimal convergence rates for such matrices [71, 72, 73, 74]. Ruge and Stüben developed a classical AMG solver [71, 72, 73, 74, 75]. Their `RS` (Ruge-Stüben) coarsening strategy established the foundation for the development of many other AMG solvers.

Iterative solvers can be optimized by applying preconditioners. The domain decomposition preconditioners were developed in 1999, which are cheaper and faster than the classical Additive Schwarz preconditioners for general sparse linear systems. Both iteration and CPU time can be saved [24]. Constraint Pressure Residual (CPR) preconditioners (multistage preconditioners) also have a broad application [2, 20, 21, 49]. Cao et al. described a multistage parallel linear solver framework for reservoir simulation with a two-stage CPR scheme [21]. The first stage is a highly efficient Parallel Algebraic Multigrid (PAMG) preconditioner and the second stage is a parallel ILU-type preconditioner. Hu et al. devised the Fast Auxiliary Space Preconditioning (FASP) method which is a general framework for constructing effective preconditioners [20]. The FASP can transform a complicated problem into a sequence of simpler solver-friendly systems by constructing appropriate auxiliary spaces. This method can be easily generalized to complicated models in reservoir simulation, such as the modified black oil model for simulating polymer flooding.

The ILU preconditioner is one of the most commonly used preconditioners [17, 7, 38, 37]. On a platform composed of NVIDIA TESLA C1060 and a CPU Intel Xeon E5504 Processor, Li et al. implemented a GPU-accelerated ILU factorization preconditioned GMRES method and achieved a speedup of nearly 4 with respect to a CPU. They also implemented a GPU-accelerated Incomplete Cholesky (IC) factorization preconditioned CG method which is faster than its CPU counterpart up to 3 times. Chen et al. implemented a block-wise ILU preconditioner on a single GPU platform which shows a faster acceleration effect on a GPU against a CPU [9]. Klie et al. proposed a novel poly-algorithmic solver for realistic black oil and compositional flow scenarios on a multicore CPU and GPU platform. They exploited the data parallelism of a GPU to implement preconditioner options such as BILU(k), BILUT and multicoloring SSOR. Their computational experiments revealed that a preconditioned solver yields significant speedups against conventional CPU multicore solver implementations [37].

Nowadays, parallel computing has been more and more important and a lot of work has been done in a variety of scientific fields, such as the reservoir simulation [10, 11]. In the study of AMG solvers, Luby et al. designed a parallel coarsening strategy `CLJP` for parallel computers [76, 77]. Henson and Yang proposed parallel coarsening strategies PMIS and HMIS [78, 79]. Besides, Yang and her collaborators developed a famous parallel AMG solver BoomerAMG [76, 78, 80, 79], which is the most famous parallel AMG solver/preconditioner for parallel computers.

As GPUs become a popular parallel hardware architecture, some solvers for solving linear systems have been developed on GPUs, such as Krylov linear solvers [49, 37, 38, 40]. Naumov [52] and Chen et al. [39] studied parallel triangular solvers for GPUs which focus on solving lower or upper triangular matrices derived from linear solvers or preconditioners in parallel. Haase et al. developed a parallel AMG solver using a GPU cluster [36]. That is a multi-GPU implementation of the Preconditioned Conjugate Gradient algorithm with an Algebraic Multigrid preconditioner (PCG-AMG) for an elliptic model problem on a 3D unstructured grid. They also developed an efficient parallel SPMV scheme underlying the PCG-AMG algorithm. With eight GPUs, about

100 speedup is obtained against a typical server CPU core [36]. AMG solvers are one of the most effective solvers in reservoir simulation. Chen et al. designed classical AMG solvers on a single GPU [42]. Their solvers are based on NVIDIA Tesla GPUs and up to 12.5 times speedup is obtained compared to the corresponding CPU-based AMG solver. Bell et al. investigated fine-grained parallelism of AMG solvers on a single GPU [50]. They developed a parallel AMG method which employs substantial fine-grained parallelism in both the construction of the multigrid hierarchy as well as the cycling or solution stage. The resulting solver achieves an average speedup of 1.8 in the setup phase and 5.7 in the cycling phase when compared to a representative CPU implementation [50]. Bolz, Buatois, Goddeke, Bell, Wang, Brannick, Stone and their collaborators also studied GPU-based parallel Algebraic Multigrid solvers, and details can be acquired in the references [43, 44, 45, 46, 53].

NVIDIA developed a hybrid matrix format HYB (Hybrid of ELL and COO) and sparse Krylov subspace solvers [49], which are used for general sparse matrices [31, 32]. The HYB sparse storage format is composed of a regular part, usually stored in ELL format, and an irregular part, usually stored in COO format. The HYB requires the use of a conversion operation to store a matrix in it, which partitions the original matrix into the regular part and the irregular part [69].

Chen et al. designed a hybrid matrix format named HEC (Hybrid of ELL and CSR). An HEC matrix consists of two parts which are an ELL matrix and a CSR matrix. The ELL matrix is regular and each row has the same length. The matrix is in column-major order when being stored on GPUs. The CSR matrix stores the irregular part of a given matrix. The advantages of the HEC matrix is that it is convenient to design SPMV algorithms and implement ILU preconditioners. Based on the HEC format, Chen et al. designed the SPMV algorithm on GPUs [33] and Krylov solvers [34, 35, 39, 40, 47, 41].

Saad et al. developed a JAD (jagged diagonal) matrix format for GPU architecture. They also developed the corresponding SPMV algorithm [28, 37]. The JAD format can be viewed as a generalization of the Ellpack-Itpack format which removes the assumption on fixed-length rows

[7]. First, it requires sorting the rows by a non-increasing order according to the number of non-zero elements per row. Then all the first element of each row constitutes the first JAD; all the second element of each row constitutes the second JAD, and so on. Thus the number of JADs is the largest number of non-zero elements per row. The JAD format has performance superiority over the CSR format. In the JAD format, consecutive threads can access contiguous memory address, which follows the suggested memory access pattern of GPU. The usage of memory bandwidth is improved by coalescing memory access [38].

The Fast Fourier Transform is an efficient algorithm for computing discrete Fourier transforms and their inverses and is widely used in numerical computational areas. NVIDIA implemented the scientific computing library cuFFT (CUDA Fast Fourier Transform) which provides a simple interface for computing FFTs up to 10x faster [29]. NVIDIA also developed some other mathematical calculation libraries for GPUs, such as the CUDA Basic Linear Algebra Subroutines (cuBLAS) library which is a GPU-accelerated version of the complete standard BLAS library [31, 32, 29], the CUDA Sparse Matrix (cuSPARSE) library which is a collection of basic linear algebra subroutines used for sparse matrices, and the CUDA Math library which provides a collection of standard mathematical functions [70]. All these CUDA libraries can be used on GPUs directly.

In summary, the investigation of preconditioned solvers for large-scale linear systems on GPU achitecture and related subjects have been a specialized research area. With the hardware techniques advancing rapidly, high performance parallel computing must have more and more real applications. In our research, we focus on the implementation of the GMRES solver with the ILU(k) preconditioner on multiple GPU architecture, which will benefit the computation speed of a reservoir simulator and other numerical calculation demands in industry.

Chapter 2

BACKGROUND

2.1 Reservoir Simulation and Large-Scale Sparse Linear Systems

2.1.1 The Procedure of Reservoir Simulation

A petroleum reservoir is a porous medium in which the fluids (typically, oil, water, and gas) can flow. The procedure of a reservoir simulation can be divided into four stages [1]; see Figure 2.1.

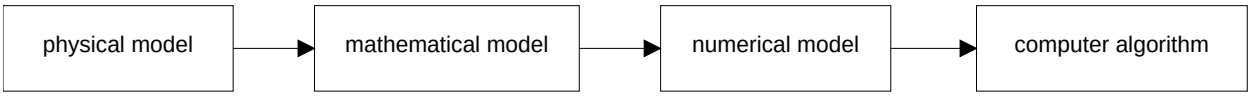


Figure 2.1: Reservoir simulation stages

At the first stage, a physical model is established according to the physical characteristics of a field reservoir. These characteristics include the size of the reservoir, the properties of rock, and the properties of fluids. Sometimes, a series of chemical reactions are also contained, for example, in a combustion process.

Physical phenomena of a reservoir can be modeled by a set of equations at the second stage. These equations can be categorized into three major categories, a set of mass conservation equations for fluid components, a set of Darcy equations for fluid phases and a set of state equations. The mass conservation and Darcy equations are partial differential equations. All these equations form a complete mathematical system and can be solved under certain boundary conditions and initial conditions. An energy equation is included when temperature varies.

The PDEs (Partial Differential Equations) are often nonlinear equations which cannot be solved by analytical methods. Their solution can be acquired by numerical methods (approximate methods). The third stage is the process of generating numerical models by discretizations of PDEs.

The approaches of discretization include the Finite Difference Method (FDM), the Finite Element Method (FEM) and the Finite Volume Method (FVM). Eventually, the numerical models are all linear systems, and the coefficient matrices of such systems are large and sparse.

Because a large-scale sparse linear system often contains millions of rows and unknowns, computer algorithms must be used for solving such a system. Most simulation time is spent on solving linear systems when running a simulator. For example, over 70% of time is consumed on the solution of these systems for the black oil model [2]. The algorithms can be basic iterative methods, Krylov subspace methods, and AMG methods. How to develop the algorithms and implement them on modern computer hardware is critical and challenging work for speeding up the solution process. It is the study area of this thesis.

2.1.2 PDEs for Reservoir Simulation

According to the physical characteristics of a reservoir and the concrete demands of a simulation, different physical and mathematical models are designed, such as a classical black oil model, an extended black oil model, a compositional flow model, and a thermal recovery model. These models must obey the mass conservation law, Darcy's law and other physical laws. In this subsection, a classical black oil model is presented as an example of PDEs. It contains three phases (the water, oil, and gas phases) and three components (the water, oil, and gas components). The components are defined at the standard conditions (60 F and 14.7 psia). There is no component exchange between any pair of phases except that the gas component is allowed to dissolve in the oil phase. The mathematical model is described by the following equations where the lowercase subscripts w, o, and g represent the water phase, oil phase, gas phase, respectively, and the uppercase subscripts W, O, and G represent the water component, oil component, gas component, respectively.

Component mass conservation

$$\frac{\partial}{\partial t}(\phi\rho_w S_w) = -\nabla\cdot(\rho_w \boldsymbol{u_w}) + q_W \tag{2.1}$$

$$\frac{\partial}{\partial t}(\emptyset \rho_{O_o} S_o) = -\bigtriangledown \cdot (\rho_{O_o} \boldsymbol{u_o}) + q_O \tag{2.2}$$

$$\frac{\partial}{\partial t}(\emptyset[\rho_{G_o} S_o + \rho_g S_g]) = -\bigtriangledown \cdot (\rho_{G_o} \boldsymbol{u_o} + \rho_g \boldsymbol{u_g}) + q_G \tag{2.3}$$

where

- $\emptyset$: porosity; $\boldsymbol{u_w}$, $\boldsymbol{u_o}$, $\boldsymbol{u_g}$: Darcy velocity
- ρ_w: water density; ρ_g: gas density
- ρ_{O_o}: partial densities of the oil components in oil phase
- ρ_{G_o}: partial densities of the gas components in oil phase
- q_W , q_O , q_G: source/sink of component

Darcy's law

$$\boldsymbol{u}_\alpha = -\frac{\boldsymbol{k}_\alpha}{\mu_\alpha}(\bigtriangledown p_\alpha - \rho_\alpha g \bigtriangledown z) \tag{2.4}$$

where

- $\alpha = w, o, g$
- p: pressure of fluid phase; $\boldsymbol{k}$: effective permeability of fluid phase
- z: value of depth; g: gravitational acceleration

Fluid saturation

$$S_w + S_o + S_g = 1 \tag{2.5}$$

where

- S_w: water saturation; S_o: oil saturation; S_g: gas saturation

Capillary pressure

$$p_{cow} = p_o - p_w \tag{2.6}$$

$$p_{cgo} = p_g - p_o \tag{2.7}$$

where

- p_{cow}: capillary pressure between oil phase and water phase for water-wet rock
- p_{cgo}: capillary pressure between gas phase and oil phase

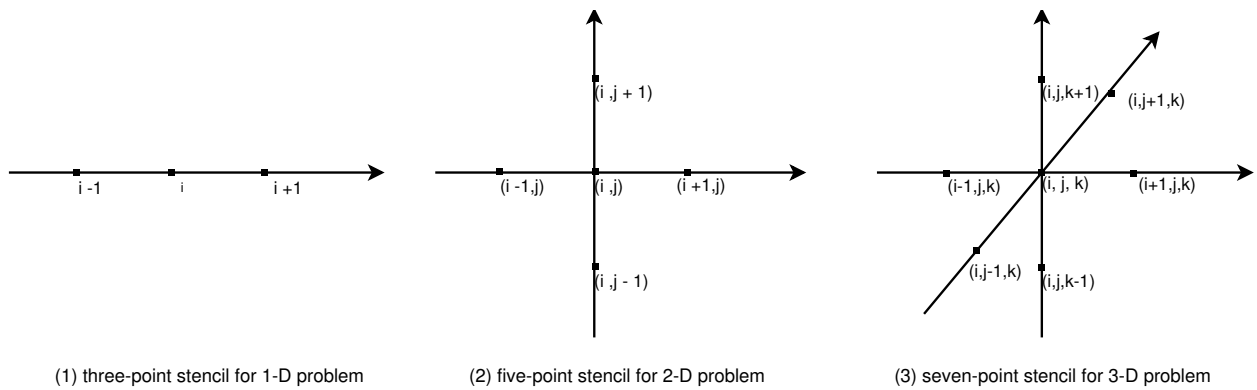
There are nine equations from (2.1) to (2.7). The fluid effective permeability and capillary pressure can be obtained by experiments. The fluid density can be expressed as functions of pressure. Therefore, a complete system is formed with nine unknowns (S_w, S_o, S_g, p_w, p_o, p_g, $\boldsymbol{u}_w$, $\boldsymbol{u}_o$, $\boldsymbol{u}_g$), which can be solved under given boundary conditions and initial conditions.

2.1.3 Discretization of PDEs

Generally, there are three methods to discretize a Partial Differential Equation, which are Finite Difference Methods (FDM), Finite Element Methods (FEM), and Finite Volume Methods (FVM). Because the FDM is most commonly used in reservoir simulation, this subsection describes the main principle of the FDM. The FDM uses finite difference equations to approximate derivatives. The partial derivatives in a Partial Differential Equation are derived by low order Taylor series expansions. The FDM requires that a physical domain is divided into subregions, for example, rectangles. The coefficient matrix of a finite difference equation is often regularly structured and composed of nonzero diagonals.

Since different-dimensional FDMs have a similar principle, the one-dimensional FDM is analyzed and a simple example is given in the following part. For a function u, Taylor's formulas can be expressed as in (2.8) and (2.9) where h represents a positive spatial step $\triangle x$.

$$u(x+h) = u(x) + h\frac{du}{dx} + \frac{h^2}{2}\frac{d^2u}{dx^2} + O(h^2) \tag{2.8}$$



(1) three-point stencil for 1-D problem (2) five-point stencil for 2-D problem (3) seven-point stencil for 3-D problem

Figure 2.2: Finite difference method

$$u(x-h) = u(x) - h\frac{du}{dx} + \frac{h^2}{2}\frac{d^2u}{dx^2} + O(h^2) \tag{2.9}$$

Based on Taylor's formulas, the following approximations can be acquired. The structure of the stencil can be represented by Figure 2.2-(1).

Forward difference

$$\frac{du(x)}{dx} \approx \frac{u(x+h)-u(x)}{h} \tag{2.10}$$

Backward difference

$$\frac{du(x)}{dx} \approx \frac{u(x)-u(x-h)}{h} \tag{2.11}$$

Centered difference

$$\frac{du(x)}{dx} \approx \frac{u(x+h)-u(x-h)}{2h} \tag{2.12}$$

Centered difference of second derivative

$$\frac{d^2u(x)}{dx^2} \approx \frac{u(x+h)-2u(x)-u(x-h)}{h^2} \tag{2.13}$$

Centered difference of second order operator

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[a(x)\frac{\mathrm{d}u}{\mathrm{d}x}\right] \approx \frac{a_{i+1/2}(u_{i+1}-u_i) - a_{i-1/2}(u_i-u_{i-1})}{h^2} \tag{2.14}$$

Consider a simple one-dimensional equation example [7]:

$$-\frac{\partial^2(x)}{\partial x^2} = f(x) \quad for\ x \in (0,1) \tag{2.15}$$

$$u(0) = u(1) = 0 \tag{2.16}$$

If the interval (0,1) is discretized into $n+1$ equal parts by setting $h = 1/(n+1)$, $n+2$ points of x_i can be created:

$$x_i = ih, \quad i = 0,1,...,n+1 \tag{2.17}$$

By applying the centered difference approximation (2.12) to equation (2.15), the discretized equation can be obatined; see equation (2.18).

$$\frac{1}{h^2}(-u_{i-1} + 2u_i - u_{i+1}) = f_i \tag{2.18}$$

For $n = 6$, a linear system (2.19) is obtained.

$$A\vec{x} = \vec{f} \tag{2.19}$$

where

$$A = \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & -1 & 2 & -1 & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix} \tag{2.20}$$

Obviously, matrix (2.20) has three diagonals. By analogy, a matrix of five diagonals can be obtained for a two-dimensional problem because the stencil contains five unknowns; see Figure 2.2-(2). A matrix of seven diagonals can be obtained for a three-dimensional problem, which has a stencil of seven unknowns; see Figure 2.2-(3).

In practice, a mathematical model and equations of a reservoir are more complicated than the above example. The detailed knowledge about the use of the FDM in reservoir simulation can

be referenced to the books written by Peaceman (1977A), Aziz and Settari(1979), Ertekin et al. (2001), Chen et al. (2005) and Chen (2007) [12, 13, 14, 2, 1].

2.1.4 Characteristics of Large-Scale Sparse Linear Systems

A reservoir can be divided into many grid blocks. Each block contains one or several unknowns. For example, a $120 \times 120 \times 120$ grid has 1,728,000 grid blocks. If each grid block contains one unknown, such as the oil pressure, and a stencil like Figure 2.2-(3) is employed, a linear system will have 1,728,000 unknowns and equations. Because its coefficient matrix is quare, it also contains the same number of columns. Therefore, each row contains 1,728,000 elements and only at most seven of them are nonzero elements. Apparently, the matrix is a large-scale sparse matrix. A large-scale spare matrix has some properties which benefit its solution.

First, a parallel algorithm can be applied to a large-scale sparse matrix. An extreme instance for a sparse matrix is a diagonal matrix; see Figure 2.3-(3). Each row contains only one unknown and can be solved independently. Thus all these equations can be solved simultaneously. The other extreme instance is a dense matrix; see Figure 2.3-(1). The equations have tight relations and they cannot be computed at the same time. Generally, a sparse matrix is situated between the two extreme cases. Figure 2.3-(2) gives a schematic of a common sparse matrix which has three diagonals. Some rows of it can be computed simultaneously. Therefore, based on the characteristic of its sparseness, parallel algorithms can be designed for solving such a sparse matrix.

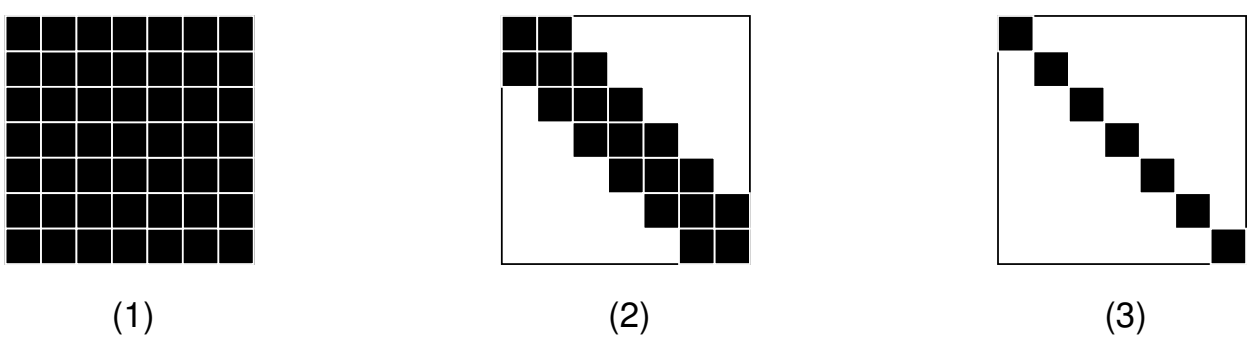


Figure 2.3: Matrix parallelization comparison

Second, a partition is necessary to a matrix in parallel computing. The original matrix is separated into some partitions and each partition has a set of rows. The rows in the same partition have a tight relation but the rows in different partitions should have relations as little as possible. In order to partition a matrix, its rows must be reordered. The rows in the same partition should be put together and then each partition can be cut out as a whole. The reorder of rows is called a row permutation operation. For example [7], a linear system is given; see equation (2.21).

$$\begin{pmatrix} A_{11} & 0 & A_{13} & 0 \\ 0 & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & 0 \\ 0 & A_{42} & 0 & A_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \tag{2.21}$$

If the second row and the third row are switched, the corresponding elements in the right-hand side vector $\vec{b}$ should also be permutated. The linear system will keep the solution unchanged; see (2.22). The diagonal elements A_{22} and A_{33} are removed from the main diagonal. Based the process of the FDM, the matrix is nonsingular and the main diagonal is dense. The elements on the main diagonal of the original matrix are all pivot elements. According to a linear algebra theory, pivot elements are kept on the main diagonal and help solve the linear system conveniently. Therefore, it is necessary to keep all the pivot elements staying on the main diagonal. In addition to a row permutation, a similar column permutation is executed on the matrix and the elements A_{22} and A_{33} come back to the main diagonal; see (2.23). Due to the columns sequence change, the unknown vector is also reordered to keep the linear system having the same solution as in (2.21). After permutation operations, the nonzero elements lie closely along the main diagonal and the matrix is more suitable for partition operations.

$$\begin{pmatrix} A_{11} & 0 & A_{13} & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ 0 & A_{22} & A_{23} & A_{24} \\ 0 & A_{42} & 0 & A_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_3 \\ b_2 \\ b_4 \end{pmatrix} \tag{2.22}$$

$$\begin{pmatrix} A_{11} & A_{13} & 0 & 0 \\ A_{31} & A_{33} & A_{32} & 0 \\ 0 & A_{23} & A_{22} & A_{24} \\ 0 & 0 & A_{42} & A_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_3 \\ b_2 \\ b_4 \end{pmatrix} \tag{2.23}$$

Third, as a large-scale sparse matrix will be solved on a computer, the sparse characteristic can be utilized to save the storage space and memory space. Because most of the elements are zero, it is not necessary to store all the elements in a computer. A popular approach is to store only the nonzero elements and their positions. The other positions are considered as zero spontaneously. A good approach is the Compressed Sparse Row (CSR) format. The equation (2.24) gives a simple example of a sparse matrix. The CSR format contains three one-dimensional arrays; see (2.25). The array Ax is used to store all the nonzero elements row by row. The array Aj is used to store the corresponding column indices of each element in Ax. The array Ap stores the start index of each row in arrays Aj and Ax. One thing to note is that the number of nonzero elements is stored as the last element in Ap. In order to facilitate the coding implementation by the C language, all the indices start from zero.

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 7 \end{pmatrix} \tag{2.24}$$

$$
\begin{aligned}
Ap &\quad \{0,2,4,6,7\} \\
Aj &\quad \{0,1,1,2,2,3,3\} \\
Ax &\quad \{1,2,3,4,5,6,7\}
\end{aligned} \tag{2.25}
$$

As stated above, the characteristics of a large-scale sparse matrix show that it is suitable to be solved by a parallel algorithm. The permutation operations can be employed to partition a matrix for parallel computing. Furthermore, the CSR format is designed for storing the matrix by saving much space resources of a computer.

2.2 Linear Solvers

2.2.1 Krylov Subspace Method

The Krylov subspace method is one of the most popular iteration methods in solving large-scale sparse linear systems. For a linear system as equation (2.26)

$$A\vec{x} = \vec{b} \tag{2.26}$$

obviously the solution can be expressed as $A^{-1}\vec{b}$. The principle for the Krylov subspace is derived from the Cayley-Hamilton theorem which says that the inverse of a matrix can be found in terms of a linear combination of its powers [15, 16]. That is, $A^{-1}\vec{b}$ can be expressed by $p(A)\vec{b}$, where p is an appropriate polynomial [7].

A Krylov subspace is defined as (2.27)

$$\kappa(A,\vec{r}_0) = span\{\vec{r}_0,\ A\vec{r}_0,\ A^2\vec{r}_0,\ ...,\ A^{m-1}\vec{r}_0\} \tag{2.27}$$

where $\vec{r}_0 = \vec{b} - A\vec{x}_0$ and $\vec{x}_0$ is an arbitrary initial guess to the solution.

Let $q_{m-1}(A)$ be a polynomial of order $m-1$ generated by the Krylov subspace. According to an approximation theory, $A^{-1}\vec{b}$ can be approximated as (2.28).

$$A^{-1}\vec{b} \approx \vec{x}_m = \vec{x}_0 + q_{m-1}(A)\vec{r}_0 \tag{2.28}$$

To calculate $\vec{x}_m$, only a series of matrix-vector multiplications are required, which is easier than calculating A^{-1} directly.

Another kind of the Krylov subspace method can be designed based on A^T. That is, the Krylov subspace is defined as (2.29)

$$\kappa(A^T, \vec{r}_0) = span\{\vec{r}_0,\ (A^T)\vec{r}_0,\ (A^T)^2\vec{r}_0,\ ...,\ (A^T)^{m-1}\vec{r}_0\} \tag{2.29}$$

Based on various concrete Krylov subspaces, different iterative methods can be designed. There are some mature Krylove subsapce methods that are widely used, such as the Arnoldi, Lanczos, GMRES (Generalized Minimum Residual), CG (Conjugate Gradient), BiCGSTAB (Biconjugate Gradient Stabilized), and QMR (Quasi Minimal Residual).

2.2.2 ILU Preconditioner

As mentioned, the performance of iterative solution techniques can be improved by preconditioners. Equation (2.30) has the same solution as the original equation (2.26), where matrix M is called a preconditioner. Matrix M has the same dimension as matrix A and it is also nonsingular. For a proper M, equation (2.30) is easier to solve than equation (2.26) because $M^{-1}A$ has a smaller condition number than A. The function of M can be explained by the following two extreme situations. If M is an identity matrix I, equation (2.31) is obtained and it is the same as equation (2.26). The matrix M has no effect in this situation. The other situation is that M equals A^{-1}. Equation (2.32) is obtained and the solution can be calculated directly by $A^{-1}\vec{b}$. However, acquiring A^{-1} has the same difficulty as solving the original system directly. Therefore, a compromise situation is to find a M^{-1} which is similar to A^{-1}.

$$M^{-1}A\vec{x} = M^{-1}\vec{b} \tag{2.30}$$

$$I^{-1}A\vec{x} = I^{-1}\vec{b} \tag{2.31}$$

$$A^{-1}A\vec{x} = A^{-1}\vec{b} \tag{2.32}$$

Fortunately, there is no need to look for M^{-1}. All the preconditioned Krylov subspace algorithms only need to solve linear systems like equation (2.33) at each step[7].

$$M\vec{y} = \vec{f} \tag{2.33}$$

A common situation is that the preconditioner M is available in the factorized form (2.34) where L and U are a lower triangular matrix and an upper triangular matrix, respectively [7]. Apparently, equation (2.34) can be separated into two triangular matrix equations to solve; see equations (2.35) and (2.36), which have very low complexity.

$$M = LU \tag{2.34}$$

$$L\vec{z} = \vec{f} \tag{2.35}$$

$$U\vec{y} = \vec{z} \tag{2.36}$$

As mentioned above, M needs to be close to A. That is, LU is close to A. If A can be factorized into L and U approximately, the preconditioner M is obtained. This process is called an incomplete LU factorization (ILU); see equation (2.37). The matrix M is called an ILU preconditioner.

$$A \approx LU \tag{2.37}$$

In practice, the ILU preconditioners, such as ILU(0), ILU(k), and ILU(T), are popularly used.

2.3 Hardware Platform

2.3.1 CPU

A CPU (Central Processing Unit) plays an important role in a computer. From the aspect of its function, it looks like the brain of a computer. The instructions, such as computation, logic, control and I/O (Input/Output), are all performed by a CPU. Modern CPUs have a strong ability of computing. For example, an Intel Core i7-5960X Processor Extreme Edition, which is the latest CPU by April 2015, can provide 3 GHz Processor Base Frequency (3.5 GHz with Turbo Boost turned on) and 68 GB/s Max Memory Bandwidth. It can even reach 384 GFlops in double precision [59, 58].

A CPU supports multiple threads. An i7-5960X Extreme Edition CPU has 8 cores and it can support 16 threads under the Intel Hyper-Threading Technology. However, traditionally, a CPU is designed as a central controller for a computer. Its multiple threads function is limited.

2.3.2 Parallel Computing

Nowadays, parallel computing has been supported by most hardware platforms. It provides computation models which allow many calculation tasks to be carried out simultaneously. Parallel algorithms have also been studied for many years, especially in the field of high performance computing. The queue of instructions used to execute an individual task is called a thread. Multiple threads can run concurrently to complete multiple tasks.

Most hardware supports parallelism, which can be categorized into two categories roughly. One is a single machine (a single node), such as a CPU with multiple cores, a CPU plus a GPU, and a CPU plus multiple GPUs. Each core runs as an individual thread to execute a task. The other one is a multiple-machine (multiple nodes) platform, such as clusters, MPPs, and grids. All the nodes perform tasks by coordinating computations in parallel.

Parallel computer programs have some characteristics and are harder to develop than sequential ones [63]. First, a synchronization mechanism of threads is necessary because the common

resources are always limited and all the threads need to compete the resources at the same time. Second, a communication mechanism is also unavoidable. Threads need to share data or transmit data at running time but the interface latency, banwidth of memory and network are often a bottleneck. Last but not the least, the design of parallel algorithms for a specific problem is difficult. Sometimes only parts of a problem can be designed as parallelization. Though there are many challenging problems for parallel computing, its superiority over sequential computing still attracts much attention.

2.3.3 GPU

Originally, a GPU is designed to accelerate the creation of images for displaying on a screen. It can be found in a video card in a personal computer. It is also widely used in mobile phones and workstations. A GPU plays a critical role for a video card just like the importance of a CPU for a computer. It specializes in enormous parallel computing. Nowadays, applications of a GPU have been more and more popular in scientific computing areas due to its outstanding floating-point calculation performance in parallel. For example, a NVIDIA Tesla K40 GPU Accelerator contains 2880 CUDA cores and has a peak performance of 1.43 TFlops (Base Clocks) and 1.66 TFlops (GPU Boost Clocks) in double precision. Its memory bandwith is 288 G/sec [56]. These performance parameters are much better than those of a CPU. The 2880 CUDA cores mean that Tesla K40 can provide 2880 threads to run simultaneously, which is a very strong parallel ability. The architecture difference between a GPU an a CPU can be schematically shown in Figure 2.4.

2.3.4 Multiple GPUs in a Single Node

In a single node, multiple GPUs can be installed on the same motherboard, which form a multiple-GPU platform. This architecture can provide higher parallel performance than a single GPU. For example, a NVIDIA Tesla K80 GPU Accelerator, which has two GPUs, contains 4992 CUDA cores. The peak performance is 1.87 TFlops (Base Clocks) and 2.91 TFlops (GPU Boost Clocks) in double precision. The memory bandwith is 480 GB/sec. Obviously, more GPUs provide high-

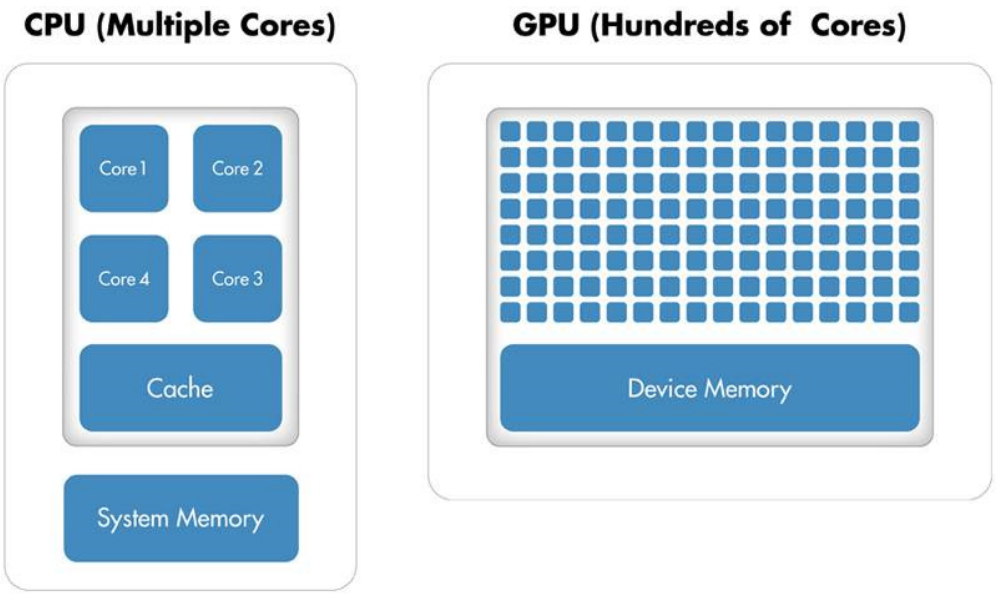


Figure 2.4: Architecture comparison of CPU and GPU

er performance in parallel. In our study, NVIDIA multiple-GPU architecture is selected as the hardware platform. A detailed analysis of it is introduced below.

A multiple GPUs architecture has a host (CPU) and several devices (GPUs). It consists of two parallel levels. The first level is formed by GPUs. Each GPU can be controlled by an OpenMP (Open Multi-Processing) thread which lies on the host. The original task is divided by the host into coarse-grained subtasks. Each device acquires a subtask from the host. Because OpenMP threads run simultaneously on the host, devices can receive tasks and perform them simultaneously; see Figure 2.5.

The processor of a typical NVIDIA GPU is categorized into two levels: a streaming multiprocessor (SM) and a streaming processor (SP or CUDA core). A GPU may have a different number of SMs. According to the architecture of a GPU, the SM may have 32, 128 and 192 SPs (CUDA cores). The Tesla C2070 has 14 SMs and a total of 448 CUDA cores. The Tesla K20X has 14 SMs (or SMXs) and each SM has 192 SPs, which has 2688 CUDA cores in total. The Tesla K40 has 15 SMs (or SMXs) and 2880 CUDA cores in all.

The NVIDIA GPUs have hierarchical memory architecture which includes a register, L1 cache,

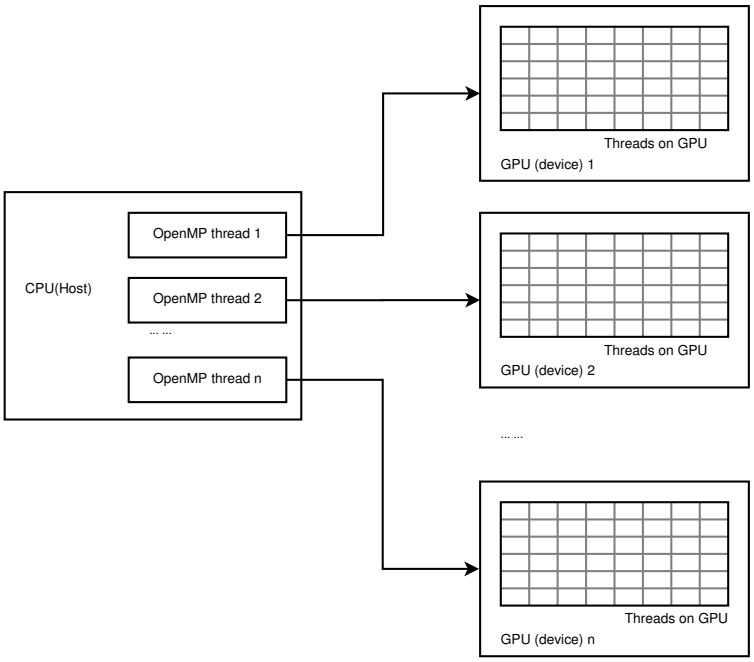


Figure 2.5: Multiple GPUs platform architecture

L2 cache, shared memory and global memory. The size of the shared memory is small, and each multiprocessor owns a small fraction of the shared memory, such as 48 KB. It is used to communicate among threads in a block. The global memory has a large size and fast speed. In our study, the global memory is employed to store the input data, such as matrices and vectors, and it is also used for data transfer between the host and the devices.

2.4 Development Environment

2.4.1 C Language

C is a popular known and widely used advanced programming language. Flexible pointer functionality is one of its grammar characteristics. Pointers are convenient for controlling memory, which leads to high-efficiency program practice. In our study, pointers are employed to manipulate matrices and vectors in a form of dynamic arrays. The other grammar characteristics, such as structure and enumeration, are also used in our codes. By means of portability of the C language,

our program can be compiled and run on different operating systems.

2.4.2 OpenMP

OpenMP (Open Multi-Processing) is an API (Application Programming Interface) which can be used in languages, such as C, C++, and Fortran. It provides a portable, scalable model with simple and flexible API for developing parallel applications. Multiple threads can be created by OpenMP and performed simultaneously. In our study, we employ mutiple OpenMP threads on the host to control each device; see Figure 2.5. We also use the OpenMP programs on a loop structure to change it into a parallel form in the algorithms which run on the host.

2.4.3 CUDA

CUDA is a programming model invented by NVIDIA. It is also a parallel computing platform [60]. The purpose of designing CUDA is for coding on NVIDIA GPUs. Since its introduction in 2006, CUDA has been widely used on thousands of applications [62].

A CUDA program consists of several phases that are executed on either the host or a device. The phases that exhibit little or no data parallelism are implemented in the host code. The phases that exhibit rich data parallelism are implemented in the device code [19]. The host code is written by ANSI C language. It is compiled with a standard C compiler and executed on the host. The device code is written by extended ANSI C (CUDA C) which has keywords for labeling parallel functions to perform on devices. The parallel functions are called kernel functions. The device code in kernel functions is compiled by the nvcc (NVIDIA CUDA Compiler) and executed on the devices. Figure 2.6 gives a simple example of a CUDA C program where the kernel function is labeled by the keyword "__global__".

Figure 2.7 shows an example of a CUDA processing flow [61], which includes four steps. First, data from the host memory is copied to the device memory. Second, the host instructs a process to the devices. Third, the devices execute in parallel in each core. Fourth, the results from the device memory are copied to the host memory. The processing flow is used in our programs. The original

```
__global__ void kernel ( void ){
    //device codes
}

int main ( void ){
    //host codes
    kernel<<<1,1>>>();
    //host codes
    return 0;
}
```

Figure 2.6: Example of simple CUDA C program

coefficient matrix and the right-hand side vector of a large-scale linear system is divided into subparts on the host. The data of each set of parts are copied from the host memory to the device memory. Then calculations are performed on each device in parallel following the instructions of the host. After all the sub-results are completed, the result data is copied from the devices to the host to constitute a final solution. More development about CUDA can be found in the references [29, 30]

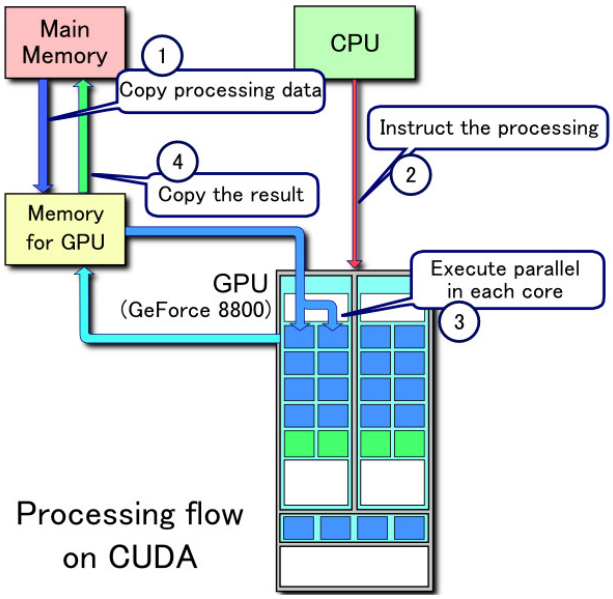


Figure 2.7: Example of CUDA processing flow

Chapter 3

GMRES METHOD

The GMRES solver is an iterative solution method for nonsymmetric linear systems developed by Saad and Schultz [7]. This method approximates a solution by a vector in a Krylov subspace with a minimal residual. In practice, the preconditioned restarted GMRES(m) is used commonly. By reference to *iterative methods for sparse linear systems* (*2nd edition*) [7], a summary derivation of the preconditioned restarted GMRES(m) is explained in this chapter. An analysis of basic operations about how to implement this algorithm is introduced in the last part.

3.1 Gram-Schmidt Process

3.1.1 Inner Product, Euclidean Norm and Orthogonality

Given two column vectors $\vec{v}$ and $\vec{w}$, the *inner product* is defined as

$$(\vec{v},\vec{w}) = \vec{v}^T\vec{w} = [v_1, v_2, \ldots, v_n]\begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = v_1w_1 + v_2w_2 + \cdots + v_nw_n. \tag{3.1}$$

The length of a vector can be measured by many ways. The *Euclidean norm* is the most commonly used, which is defined as

$$\|\vec{v}\|_2 = (\vec{v},\vec{v}) = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}. \tag{3.2}$$

The *orthogonality* of two vectors $\vec{v}$ and $\vec{w}$ are defined as $(\vec{v},\vec{w}) = 0$.

3.1.2 Projection

Given two vectors $\vec{v}$ and $\vec{i}$ where $\vec{i}$ is a unit vector, the projection of $\vec{v}$ onto $\vec{i}$ can be calculated by (3.3); see Figure 3.1:

$$\vec{q} = (\vec{i}, \vec{v})\vec{i} \tag{3.3}$$

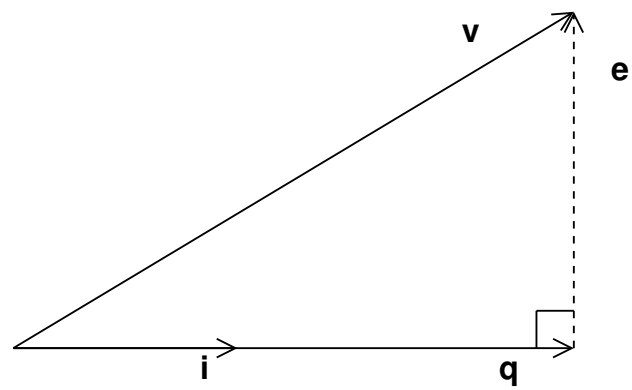


Figure 3.1: Projection triangle

3.1.3 Gram-Schmidt Process

Assume that we have a subspace S spanned by an orthonormal basis $\{\vec{q}_1, \vec{q}_2, \ldots, \vec{q}_r\}$ and a vector $\vec{v}$ which lies outside of the subspace. To construct a new subspace extended from S by including an additional linearly independent vector $\vec{v}$, there are two solutions explained below.

Solution 1: From Figure 3.2-(1), $\{\vec{q}_1, \vec{q}_2\}$ is an orthonormal basis which forms a subspace represented by the plane $EFGH$. $\vec{v}$ is a vector outside of the plane.

$$\overrightarrow{EF} = (\vec{q}_1, \vec{v})\vec{q}_1$$

$$\overrightarrow{EH} = (\vec{q}_2, \vec{v})\vec{q}_2$$

$$\overrightarrow{EG} = \overrightarrow{EF} + \overrightarrow{EH} = (\vec{q}_1, \vec{v})\vec{q}_1 + (\vec{q}_2, \vec{v})\vec{q}_2 = \sum_{i=1}^{2} (\vec{q}_i, \vec{v})q_i$$

$$\overrightarrow{GC} = \vec{v} - \overrightarrow{EG} = \vec{v} - \sum_{i=1}^{2} (\vec{q}_i, \vec{v})q_i$$

$$\vec{q}_3 = \overrightarrow{GC}/\|\overrightarrow{GC}\|_2$$

Therefore, the orthonormal basis $\{\vec{q}_1, \vec{q}_2, \vec{q}_3\}$ forms a new subspace and the vector $\vec{v}$ lies in this subspace. The Gram-Schmidt process can be constructed by the idea of solution 1. Given a

vector set $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_r\}$ where all the vectors are linearly independent, the first vector $\vec{q}_1$ of the orthonormal basis can be obtained by normalizing $\vec{x}_1$. Then the other orthonomal vectors can be acquired by the mathematical induction; see Algorithm 1. However, there is a drawback of this algorithm because of finite-precision on computers. Line 8 to Line 11 introduces a cumulative error. Thus some modifications must be applied to correct this problem.

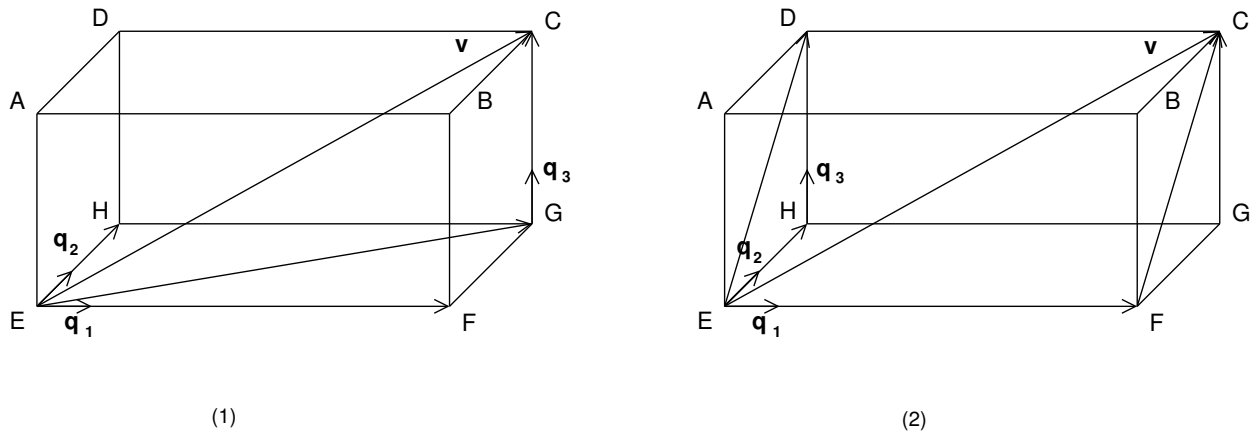


Figure 3.2: Orthonormal basis extension

Solution 2: From Figure 3.2-(2), $\{\vec{q}_1, \vec{q}_2\}$ is an orthonormal basis which forms a subspace represented by the plane $EFGH$. $\vec{v}$ is a vector outside of the plane.

Let $\vec{q} = \vec{v}$

$$\begin{aligned}
\overrightarrow{EF} &= (\vec{q}_1, \vec{q})\vec{q}_1 \\
\overrightarrow{FC} &= \vec{q} - \overrightarrow{EF} = \vec{q} - (\vec{q}_1, \vec{q})\vec{q}_1 \\
\overrightarrow{ED} &= \overrightarrow{FC} = \vec{q} - (\vec{q}_1, \vec{q})\vec{q}_1
\end{aligned} \tag{3.4}$$

Let $\vec{q} = \overrightarrow{ED}$

$$\begin{aligned}
\overrightarrow{EH} &= (\vec{q}_2, \vec{q})\vec{q}_2 \\
\overrightarrow{HD} &= \vec{q} - \overrightarrow{EH} = \vec{q} - (\vec{q}_2, \vec{q})\vec{q}_2 \\
\vec{q}_3 &= \overrightarrow{HD} / \|\overrightarrow{HD}\|_2
\end{aligned} \tag{3.5}$$

Algorithm 1 Gram-Schmidt

```
r_11 := ||x_1||_2
if r_11 = 0 then
    Stop
else
    q_1 := x_1 / r_11
end if
for j = 2: r do
    for i = 1: j - 1 do
        r_ij := (x_j, q_i)
    end for
    q := x_j - Σ_{i=1}^{j-1} r_ij q_i
    r_jj := ||q||_2
    if r_jj = 0 then
        Stop
    else
        q_j := q / r_jj
    end if
end for
```

Therefore, the orthonormal basis $\{\vec{q}_1, \vec{q}_2, \vec{q}_3\}$ forms a new subspace and the vector $\vec{v}$ lies in this subspace. In general, given a vector set $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_r\}$ where all the vectors are linearly independent, the first vector $\vec{q}_1$ of the orthonormal basis can be obtained by normalizing $\vec{x}_1$ and the other orthonomal vectors can be acquired by the mathematical induction; see Algorithm 2. Based on solution 2, this algorithm gives the same result as the Algorithm 1 but introduces smaller errors in finite-precision arithmetic. It is known as the Modified Gram-Schmidt (MGS).

Algorithm 2 Modified Gram-Schmidt

```
r_11 = ||x_1||_2
if r_11 = 0 then
    Stop
else
    q_1 = x_1 / r_11
end if
for j = 2: r do
    q = x_j
    for i = 1: j - 1 do
        r_ij = (q, q_i)
        q = q - r_ij q_i
    end for
    r_jj = ||q||_2
    if r_jj = 0 then
        Stop
    else
        q_j = q / r_jj
    end if
end for
```

3.2 Arnoldi Iteration

3.2.1 Cayley-Hamilton Theorem

In linear algebra, for a $n \times n$ nonsingular matrix A, the Cayley-Hamilton theorem is represented by the identity (3.6). The coefficients c_k can be expressed in terms of traces of powers of the matrix A. However, the calculations of coefficients become increasingly complicated and impossible

practically for large-scale matrices.

$$p(A) = A^n + c_{n-1}A^{n-1} + \cdots + c_1 A + (-1)^n det(A) I_n = 0 \tag{3.6}$$

Rewrite (3.6) as (3.7),

$$-(-1)^n det(A) I_n = A(A^{n-1} + c_{n-1}A^{n-2} + \cdots + c_1 I_n). \tag{3.7}$$

Multiply both sides by A^{-1},

$$A^{-1} = \frac{(-1)^{n-1}}{detA}(A^{n-1} + c_{n-1}A^{n-2} + \cdots + c_1 I_n). \tag{3.8}$$

Therefore, A^{-1} must belong to the subspace (3.9),

$$\kappa_n(A) = span\{1, A, A^2, \cdots, A^{n-1}\}. \tag{3.9}$$

3.2.2 Krylov Subspace

$$A\vec{x} = \vec{b} \tag{3.10}$$

In linear algebra, given a $n \times n$ matrix A and a n-dimensional vector $\vec{v}$, the order n Krylov subspace is defined as $\kappa_n(A, \vec{v}) \equiv span\{\vec{v}, A^2\vec{v}, \cdots, A^{n-1}\vec{v}\}$. The system to solve is expressed by (3.10). An initial guess is given as $\vec{x}_0$. The following deduction shows that the exact solution $\vec{x}_*$ can be found through a Krylov subspace.

$$\begin{aligned}
&A\vec{x}_0 = \vec{b} \\
&\vec{r}_0 = \vec{b} - A\vec{x}_0 \\
&A^{-1}\vec{r}_0 = A^{-1}\vec{b} - A^{-1}A\vec{x}_0 \\
&A^{-1}\vec{r}_0 = \vec{x}_* - \vec{x}_0 \\
&\vec{x}_* = \vec{x}_0 + A^{-1}\vec{r}_0
\end{aligned} \tag{3.11}$$

Obviously, $A^{-1}\vec{r}_0$ is in the Krylov subspace (3.12).

$$\kappa_n(A, \vec{r}_0) = span\{\vec{r}_0, A\vec{r}_0, A^2\vec{r}_0, \cdots, A^{n-1}\vec{r}_0\}. \tag{3.12}$$

Because the dimension n is very large for a large-scale matrix, it is impossible to search for the accurate $A^{-1}\vec{r}_0$ from such a subspace in practice. The reasonable way is to shrink this subspace to m dimension where m is much smaller than n, for example, $m = 20$. Thus, in the objective subspace (3.13), we search for an nearest solution under the required conditions.

$$\kappa_m(A,\vec{r}_0) = span\{\vec{r}_0, A\vec{r}_0, A^2\vec{r}_0, \cdots, A^{m-1}\vec{r}_0\}. \tag{3.13}$$

3.2.3 Arnoldi's Method

Though the vectors in a Krylov subspace are actually linearly independent, the extent of linear dependence becomes more and more serious with an increase in the power of a matrix. Based on this, the coordinate values of a vector will only have a slight difference; see Figure 3.3-(1). Because the digital numbers in computers are limited, an orthonormal basis is a better way for the vectors in the subspace, which expresses the coordinates in an efficient way; see Figure 3.3-(2).

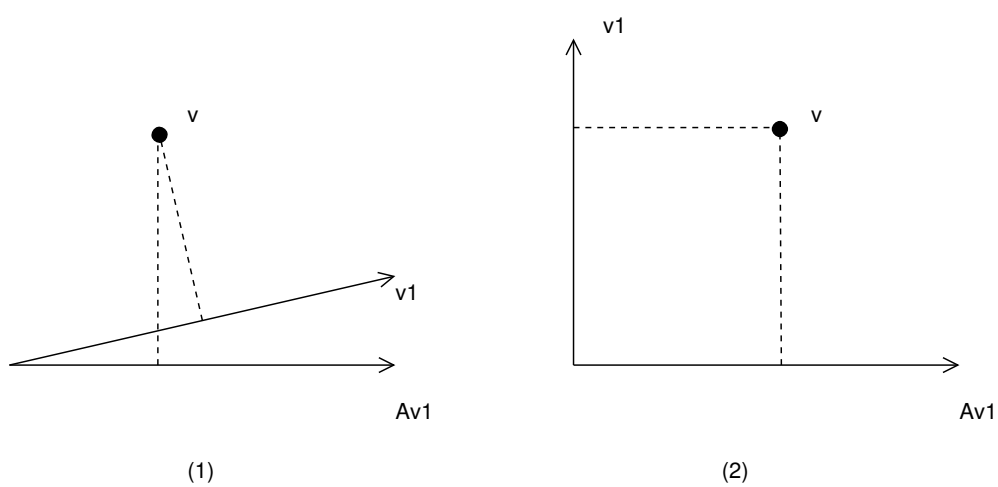


Figure 3.3: Two-dimensional example for using orthonormal basis

Given a vector $\vec{v}_1 = \vec{r}_0/\|\vec{r}_0\|_2$, the corresponding m-order Krylov subspace is $\kappa_m(A,\vec{v}_1)$; see (3.14). V_m represents the orthogonal matrix composed by the orthonormal basis vectors which span the same subspace $\kappa_m(A,\vec{v}_1)$; see (3.15).

$$\kappa_m(A,\vec{v}_1) = span\{\vec{v}_1, A\vec{v}_1, A^2\vec{v}_1, \cdots, A^{m-1}\vec{v}_1\} \tag{3.14}$$

$$V_m = [\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_m] \tag{3.15}$$

The procedure of generating an orthonormal basis V_m for the subspace $\kappa_m(A,\vec{v}_1)$ can be implemented by the Gram-Schmidt process, which is called the Arnoldi iteration. Algorithm 3 gives an Arnoldi iteration by using the standard Gram-Schmidt algorithm 1.

Algorithm 3 Arnoldi

1: $v_1 = r_0/\|r_0\|_2$
2: **for** j = 1: m **do**
3: **for** i = 1: j **do**
4:   $h_{ij} := (Av_j, v_i)$
5: **end for**
6: $\vec{w} = Av_j - \sum_{i=1}^{j} h_{ij}\vec{v}_i$
7: $h_{j+1,j} = \|\vec{w}\|_2$
8: **if** $h_{j+1,j} = 0$ **then**
9:   Stop
10: **else**
11:   $v_{j+1} = \vec{w}/h_{j+1,j}$
12: **end if**
13: **end for**

From line 11 and line 6 of Algorithm 3, equations (3.16) and (3.17) can be obtained.

$$\vec{w} = \vec{v}_{j+1}h_{j+1,j} \tag{3.16}$$

$$\vec{w} = A\vec{v}_j - \sum_{i=1}^{j} h_{ij}\vec{v}_i \tag{3.17}$$

Apparently, (3.18) is obtained.

$$A\vec{v}_j = \sum_{i=1}^{j+1} h_{ij}\vec{v}_i, \quad j = 1,2,\ldots,m \tag{3.18}$$

Thus we have the following derivation process:

$$\begin{aligned}
AV_m &= A[\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_m] \\
&= [A\vec{v}_1, A\vec{v}_2, \cdots, A\vec{v}_m] \\
&= [\vec{v}_1 h_{1\ 1} + \vec{v}_2 h_{2\ 1}, \vec{v}_1 h_{1\ 2} + \vec{v}_2 h_{2\ 2} + \vec{v}_3 h_{3\ 2}, \cdots, \vec{v}_1 h_{1\ m} + \vec{v}_2 h_{2\ m} + \cdots + \vec{v}_{m+1} h_{m+1\ m}] \\
&= \begin{pmatrix} \vec{v}_1 & \vec{v}_2 & \cdots & \vec{v}_{m+1} \end{pmatrix} \begin{pmatrix} h_{1\ 1} & \cdots & \cdots & h_{1\ m} \\ h_{2\ 1} & h_{2\ 2} & \cdots & \vdots \\ & \ddots & \ddots & \vdots \\ & & h_{m\ m-1} & h_{m\ m} \\ & & & h_{m+1\ m} \end{pmatrix}
\end{aligned}$$

Defining two Hessenberg matrices H_m as in (3.19) and $\bar{H}_m$ as in (3.20), we get equation (3.21) which will be used to deduce the GMRES.

$$H_m = \begin{pmatrix} h_{1\ 1} & \cdots & \cdots & h_{1\ m} \\ h_{2\ 1} & h_{2\ 2} & \cdots & \vdots \\ & \ddots & \ddots & \vdots \\ & & h_{m\ m-1} & h_{m\ m} \end{pmatrix} \tag{3.19}$$

$$\bar{H}_m = \begin{pmatrix} H_m \\ h_{m+1\ m} \end{pmatrix} \tag{3.20}$$

$$AV_m = V_{m+1}\bar{H}_m \tag{3.21}$$

As stated in the Gram-Schmidt process section, the Arnoldi iteration process derived from the Modified Gram-Schimidt process will be more practical than from the standard Modified Gram-Schimidt process, which is listed by Algorithm 4.

Algorithm 4 Arnoldi-Modified Gram-Schmidt

```
1:  v⃗_1 = r⃗_0/‖r⃗_0‖_2
2:  for j = 1: m do
3:      w⃗ = A v⃗_j
4:      for i = 1: j do
5:          h_ij = (w⃗, v⃗_i)
6:          w⃗ = w⃗ − h_ij v⃗_i
7:      end for
8:      h_{j+1,j} = ‖w⃗‖_2
9:      if h_{j+1,j} = 0 then
10:         Stop
11:     else
12:         v⃗_{j+1} = w⃗/h_{j+1,j}
13:     end if
14: end for
```

3.3 GMRES

3.3.1 GMRES Algorithm

Rewrite (3.11) in (3.22) by using the orthonormal basis.

$$\vec{x} = \vec{x}_0 + V_m \vec{y} \tag{3.22}$$

where $\vec{y}$ is a vector with m elements. Because V_m is only a m-dimensional subspace, only an approximate solution $\vec{x}$ instead of the exact solution $\vec{x}_*$ can be found in this subspace in general.

Assume that $\vec{r}_m$ is the least residual which can be obtained by solving a minimization problem; see (3.23).

$$\|\vec{r}_m\|_2 = argmin_{\vec{x}}\|\vec{b} - A\vec{x}\|_2 = argmin_{\vec{y}}\|\vec{b} - A(\vec{x}_0 + V_m\vec{y})\|_2 \tag{3.23}$$

$$\beta = \|\vec{r}_0\|_2 \tag{3.24}$$

$$\vec{v}_1 = \vec{r}_0/\|\vec{r}_0\|_2 \tag{3.25}$$

$$\vec{e}_1 = (1, 0, \cdots, 0)_{(m+1)\times 1} \tag{3.26}$$

$$\vec{v}_1 = V_{m+1}\vec{e}_1 \tag{3.27}$$

Using (3.24), (3.25), (3.26), (3.27) and (3.21), we have the deduction:

$$\begin{aligned}
\vec{b} - A(\vec{x}_0 + V_m\vec{y}) &= \vec{b} - A\vec{x}_0 - AV_m\vec{y} \\
&= \vec{r}_0 - V_{m+1}\bar{H}_m\vec{y} \\
&= \beta\vec{v}_1 - V_{m+1}\bar{H}_m\vec{y} \\
&= \beta V_{m+1}\vec{e}_1 - V_{m+1}\bar{H}_m\vec{y} \\
&= V_{m+1}(\beta\vec{e}_1 - \bar{H}_m\vec{y}).
\end{aligned} \tag{3.28}$$

.

Because V_m is an orthonormal basis, $\vec{r}_m$ is expressed by (3.29):

$$\|\vec{r}_m\|_2 = argmin_{\vec{y}}\|V_{m+1}(\beta\vec{e}_1 - \bar{H}_m\vec{y})\|_2 = argmin_{\vec{y}}\|\beta\vec{e}_1 - \bar{H}_m\vec{y}\|_2 \tag{3.29}$$

In order to solve this minimization problem, a reasonable approach is to transfer the Hessenberg matrix $\bar{H}_m$ to a matrix $\bar{R}_m$ which is formed by an upper triangular matrix R_m with a zero row; see (3.30).

$$\bar{R}_m = \begin{pmatrix} R_m \\ 0 \end{pmatrix} \tag{3.30}$$

A series of rotation matrices Ω_i can be employed to implement the transformation; see (3.31).

$$\bar{R}_i = \Omega_i \Omega_{i-1} \cdots \Omega_1 \bar{H}_m \tag{3.31}$$

$$\Omega_i = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & s_i & & & \\ & & & -s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix} \begin{array}{l} \\ \\ \\ \leftarrow row\ i \\ \leftarrow row\ i+1 \\ \\ \\ \\ \end{array} \tag{3.32}$$

$$s_i = \frac{h_{i+1\ i}}{\sqrt{(h_{i\ i}^{(i-1)})^2 + h_{i+1\ i}^2}} \tag{3.33}$$

$$c_i = \frac{h_{i\ i}^{(i-1)}}{\sqrt{(h_{i\ i}^{(i-1)})^2 + h_{i+1\ i}^2}} \tag{3.34}$$

where $i = 1, 2, \cdots, m$. $\bar{R}_m$ is obtained after m steps. The superscript $(i-1)$ of an entry denotes its value from $\bar{R}_{i-1}$.

Because each rotation matrix Ω_i is unitary, it does not affect the Euclidean norm. We have the following deduction:

$$\begin{aligned} \|\beta\vec{e}_1 - \bar{H}_m\vec{y}\|_2^2 &= \|\Omega_m\Omega_{m-1}\cdots\Omega_1(\beta e_1 - \bar{H}_m\vec{y})\|_2^2 \\ &= \|\Omega_m\Omega_{m-1}\cdots\Omega_1\beta e_1 - (\Omega_m\Omega_{m-1}\cdots\Omega_1\bar{H}_m)\vec{y}\|_2^2 \\ &= \|\Omega_m\Omega_{m-1}\cdots\Omega_1\beta e_1 - \bar{R}_m\vec{y}\|_2^2. \end{aligned}$$

.

Obviously, $\Omega_m\Omega_{m-1}\cdots\Omega_1\beta\vec{e}_1$ is a $(m+1)\times 1$ vector. Let $\vec{g}_m$ represent this vector and $\vec{g}'_m$ represents the vector excluding the last element γ_{m+1}; see (3.35).

$$\vec{g}_m = \Omega_m\Omega_{m-1}\cdots\Omega_1\beta\vec{e}_1 = \begin{pmatrix} \vec{g}'_m \\ \gamma_{m+1} \end{pmatrix} \tag{3.35}$$

Thus,

$$\begin{aligned} \|\beta\vec{e}_1 - \bar{H}_m\vec{y}\|_2^2 &= \|\vec{g}_m - \bar{R}_m\vec{y}\|_2^2 \\ &= \left\| \begin{pmatrix} \vec{g}'_m \\ \gamma_{m+1} \end{pmatrix} - \begin{pmatrix} R_m \\ 0 \end{pmatrix} \vec{y} \right\|_2^2 \\ &= \left\| \begin{pmatrix} \vec{g}'_m - R_m\vec{y} \\ \gamma_{m+1} \end{pmatrix} \right\|_2^2 \\ &= |\gamma_{m+1}|^2 + \|\vec{g}'_m - R_m\vec{y}\|_2^2. \end{aligned} \tag{3.36}$$

The minimum value of (3.36) can be obtained when the second part is zero which is an upper triangular system and can be solved easily; see (3.37).

$$\vec{y}_m = R_m^{-1}\vec{g}'_m \tag{3.37}$$

From (3.29) and (3.36), the norm of the minimization residual is obtained by (3.38). If it meets the requirement, we can use (3.22) to get the solution vector $\vec{x}$.

$$\|\vec{r}_m\|_2 = |\gamma_{m+1}| \tag{3.38}$$

In practice, m is often assigned with a small integer, such as 20 or 30. But the residual $\vec{r}_m$ may not satisfy the requirement after the above calculations. If a larger m is employed, the accuracy of the solution can be improved. However, the memory and computational resources would become unacceptable quickly with m increasing. A suitable approach is to restart the process and use the

solution $\vec{x}_m$ as an initial vector $\vec{x}_0$. The restarted process is repeated until a satisfactory solution is arrived. Based on the ideas of this subsection, the GMRES algorithm with restarted number m is given in Algorithm 5.

Algorithm 5 GMRES(m)

```
Given A, b, x_0
for k = 1, 2, ··· do
    r_0 = b − A x_0, β = ||r_0||_2, v_1 = r_0/β
    Generate V_m, H̄_m by modified Arnoldi iteration with A, r_0; see Algorithm 4.
    Compute r_m and y_m of the minimization problem ||r_m||_2 = argmin_ȳ ||β e_1 − H̄_m y||_2; see (3.29).
    Compute x_m = x_0 + V_m y_m; see (3.22).
    if satisfied then
        Stop
    else
        x_0 = x_m
    end if
end for
```

3.3.2 Preconditioned GMRES

As stated in subsection 2.2.2, a preconditoner can be used to improve the performance of iterative solvers. A linear system with a left-preconditioner M is written as (3.39) which has the identical solution to the original equation (3.10).

$$M^{-1}A\vec{x} = M^{-1}\vec{b} \tag{3.39}$$

Obviously, the residual vector $\vec{r}$ is given by (3.40).

$$\begin{aligned} \vec{r} &= M^{-1}\vec{b} - M^{-1}A\vec{x} \\ \vec{r} &= M^{-1}(\vec{b} - A\vec{x}) \\ M\vec{r} &= \vec{b} - A\vec{x} \end{aligned} \tag{3.40}$$

Thus the Krylov subspace used for searching for an approximate solution is defined as (3.41).

$$\kappa_m(M^{-1}A, \vec{r}_0) = span\{\vec{r}_0, M^{-1}A\vec{r}_0, (M^{-1}A)^2\vec{r}_0, \cdots, (M^{-1}A)^{m-1}\vec{r}_0\}. \tag{3.41}$$

The complete algorithm for the restarted GMRES(m) solver with left-preconditioner M is given in Algorithm 6 which is implemented in our research.

3.4 Basic Operations

From Algorithm 6, the most operations except the solution of a preconditioner system are matrix-vector multiplication and vector operations as described below, where α , β ,γ and r represent scalars.

$$\vec{y} = \alpha A\vec{x} + \beta\vec{y}, \tag{3.42}$$

$$\vec{y} = \alpha\vec{x} + \beta\vec{y}, \tag{3.43}$$

$$\vec{z} = \alpha\vec{x} + \beta\vec{y}, \tag{3.44}$$

$$\gamma = (\vec{x}, \vec{y}), \tag{3.45}$$

$$r = \|\vec{x}\|_2 = \sqrt{(\vec{x}, \vec{x})}. \tag{3.46}$$

The algorithms for vector operations, such as scalar multiplication, dot product and vector addition, are easy to implemented. The NVIDIA also provides BLAS operations through NVIDIA cuBLAS library. However, the SPMV (Sparse Matrix-Vector Multiplication) operation is much more complicated than other vector operations. It will be analyzed in the first section of Chapter 4. The preconditioner systems need to be solved at least once at each iteration; for instance, see line 3 and line 6 in Algorithm 6. The second and third sections of Chapter 4 detail the mechanism of establishing a preconditioner matrix on multiple-GPU architecture and the approach for solving it.

Algorithm 6 The preconditioned GMRES(m) algorithm

```
Given A, b⃗, x⃗_0
for k = 1, 2, ··· do
    Solve r⃗_0 from M r⃗_0 = b⃗ − A x⃗_0
    β = ‖r⃗_0‖_2, v⃗_1 = r⃗_0/β
    for j = 1: m do
        M w⃗ = A v⃗_j
        for i = 1: j do
            h_ij = (w⃗, v⃗_i)
            w⃗ = w⃗ − h_ij v⃗_i
        end for
        h_{j+1,j} = ‖w⃗‖_2
        if h_{j+1,j} = 0 then
            Stop
        else
            v⃗_{j+1} = w⃗/h_{j+1,j}
        end if
    end for
    Define V_m, H̄_m.
    Compute r⃗_m and y⃗_m of the minimization problem ‖r⃗_m‖_2 = argmin_{y⃗} ‖β e⃗_1 − H̄_m y⃗‖_2; see (3.29).
    Compute x⃗_m = x⃗_0 + V_m y⃗_m; see (3.22).
    if satisfied then
        Stop
    else
        x⃗_0 = x⃗_m
    end if
end for
```

Chapter 4

GPU COMPUTATION

This chapter is made of three sections which concentrate on the major implementation mechanism of GMRES with ILU(k). The first section explains the SPMV mechanism in which the domain decomposition, the reordering-compact method and the communication mechanism are detailed. The second section analyzes the nested parallel characteristic of a multiple-GPU platform, where the RAS principle and then the nested RAS framework are presented. The last section describes the decoupled algorithms about ILU(k). The principle of a parallel triangular solver is also introduced in this section.

4.1 Sparse Matrix-Vector Multiplication Mechanism

4.1.1 Domain Decomposition

Generally, SPMV is expressed by equation (4.1),

$$\vec{y} = A\vec{x}. \tag{4.1}$$

where $\vec{y}$ is the result vector, A is the original matrix, and $\vec{x}$ is the vector for multiplication.

In our study, we assume that A is nonsingular. Figure 4.1 uses four GPUs as a schematic diagram. Figure 4.1 (1)-(3) explains the process of partitioning a matrix. Because each GPU owns a domain matrix, the number of domain matrices equals the number of GPUs. As the SPMV algorithm in linear algebra is based on the computation of rows, it is natural to divide matrix A by rows into domain matrices. We assume that m GPUs are employed. For a given $N \times N$ matrix A, its row indices form a set $S = \{1, 2, \cdots, N\}$. Each GPU stores some rows of A and a corresponding subset of S. Then we have m subsets of S, denoted by S_1, S_2, $\cdots$, and S_m, which satisfy the following two equations:

$$S_i \cap S_j = \emptyset, i \neq j, \tag{4.2}$$

$$S_1 \cup S_2 \cup \cdots \cup S_m = S. \tag{4.3}$$

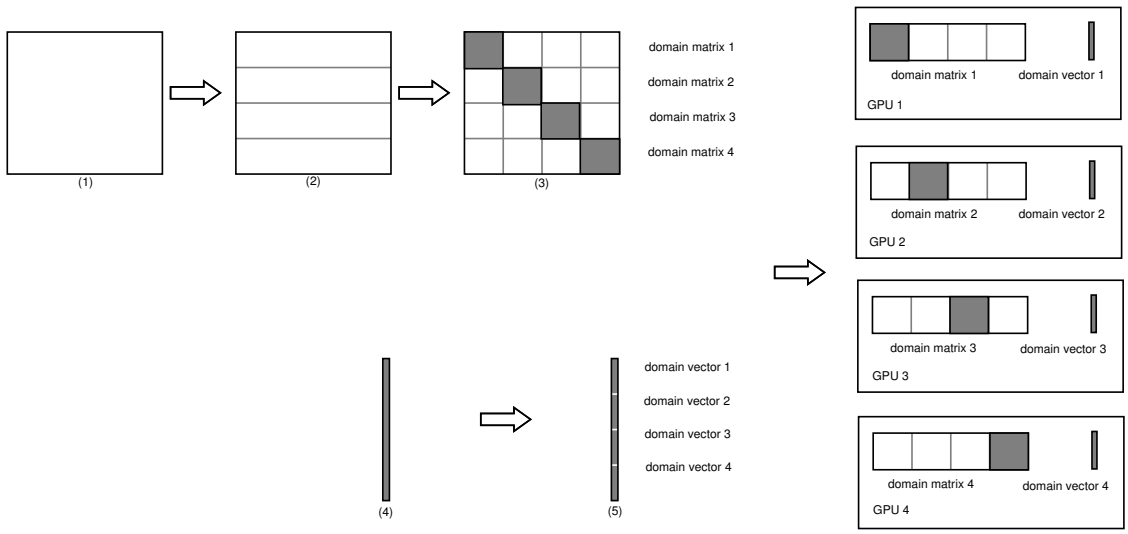


Figure 4.1: Matrix and vector domain decomposition

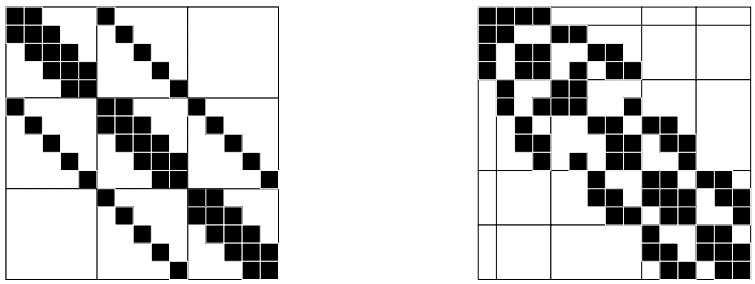
In Figure 4.1-(2), each horizontal rectangle stands for a domain matrix. The row indices form a subset S_i. According to the graph theory, the pivot element of each row can be looked at as a node. The nodes in the same domain matrix form a set. The other non-zero elements of a row are looked at as communication lines. From Figure 4.1-(3), if an element is located in the pivot diagonal block, the communication is within the same GPU. However, if an element is located outside of the pivot diagonal block, there is a communication between two different GPUs. Because the cost of data transmission within the same GPU is much less than that between two different GPUs, the data transmission among GPUs should be reduced as much as possible. Thus the relation of rows within the same domain matrix should be as tight as possible. We can explain this principle by equation (4.4),

$$y_i = \sum_{j=1}^{n} A_{ij} x_j. \tag{4.4}$$

where A_{ij} is the ijth nonzero element of matrix A, x_j is the jth element of vector $\vec{x}$ and y_i is the ith element of the SPMV result vector.

Although matrix A is sparse, it is possible for a nonzero element A_{ij} to appear in any column in row i. This means that x_j, the corresponding element used for SPMV in vector $\vec{x}$, may emerge at any position of vector $\vec{x}$. We establish the communication mechanism among rows according to the nonzero structure of A. For element A_{ij}, if i and j both belong to the same subset S_k, the vector element x_j must locate in the domain vector related to the domain matrix. Obviously, there is no communication because they are on the same GPU. On the other hand, if i and j belong to different subsets S_q and S_p, respectively, a communication exists between two different GPUs. This principle shows that the load of communication between different GPUs is determined by the subsets S_1, S_2, ..., S_m. Our aim is to reduce the communication load as much as possible by employing a proper partition method.

If matrix A has a regular structure which is usually obtained from the FDM (Finite Difference Method) or FVM (Finite Volume Method), we can use a sequence partition method as illustrated in Figure 4.2-(1). For an unstructured matrix A which is derived from the FVM or FEM (Finite Element Method), special partition methods should be applied to matrix A. There are some existing methods available. In our study, METIS is selected as the partition method, which can provide a quasi-optimal partition and minimize communication cost [48]. A schematic diagram of this partition method is illustrated by Figure 4.2-(2).



(1) regular matrix A (2) irregular matrix

Figure 4.2: Schematic of regular and irregular matrix

After operations of partition, most elements of a domain matrix lie in the dense pivot diagonal blocks; see Figure 4.1-(3). The row index for each row may change. The index for each element of vector $\vec{x}$ should also change in the same way for the SPMV to have a correct result. We create a mapping *pm* to record the permutation expressed by equation (4.5). We also establish a mapping *pa* to reflect the relation of each row and its partition expressed by equation (4.6).

$$i' = pm(i) \tag{4.5}$$

where i is the old row number and i' is the new row number.

$$k = pa(i') \tag{4.6}$$

where i' is the new row number and k is the domain number.

By equations (4.5) and (4.6), each row can be distributed to a domain matrix and the original positions of row data can be recorded and traced. To implement a matrix-vector product, the data of vector $\vec{x}$ should be stored in GPUs. It is not reasonable to store a whole copy of vector $\vec{x}$ in each GPU because the memory of a GPU is always limited and vector $\vec{x}$ is dense in most cases. The solution is to store only a segment of vector $\vec{x}$ on each GPU. This means that a vector also needs to be partitioned into domain vectors, which is shown in Figure 4.1-(5). Because the pivot diagonal block of a domain matrix is its only dense part, the row indices of a domain vector stored in each GPU should correspond to the column indices of the pivot diagonal block of the domain matrix. Apparently, most calculations of SPMV for a domain matrix are guaranteed on its local GPU.

4.1.2 Data Structure

In the phase of domain decomposition, a domain matrix is stored by data structure *mat_csr_t* which is the CSR (Compressed Sparse Row) format. The property Ax is used for storing the data of nonzero elements row by row. The property Aj is used for storing the column index corresponding each element in Aj. The property Ap is used for storing the starting indices of each row in Aj and

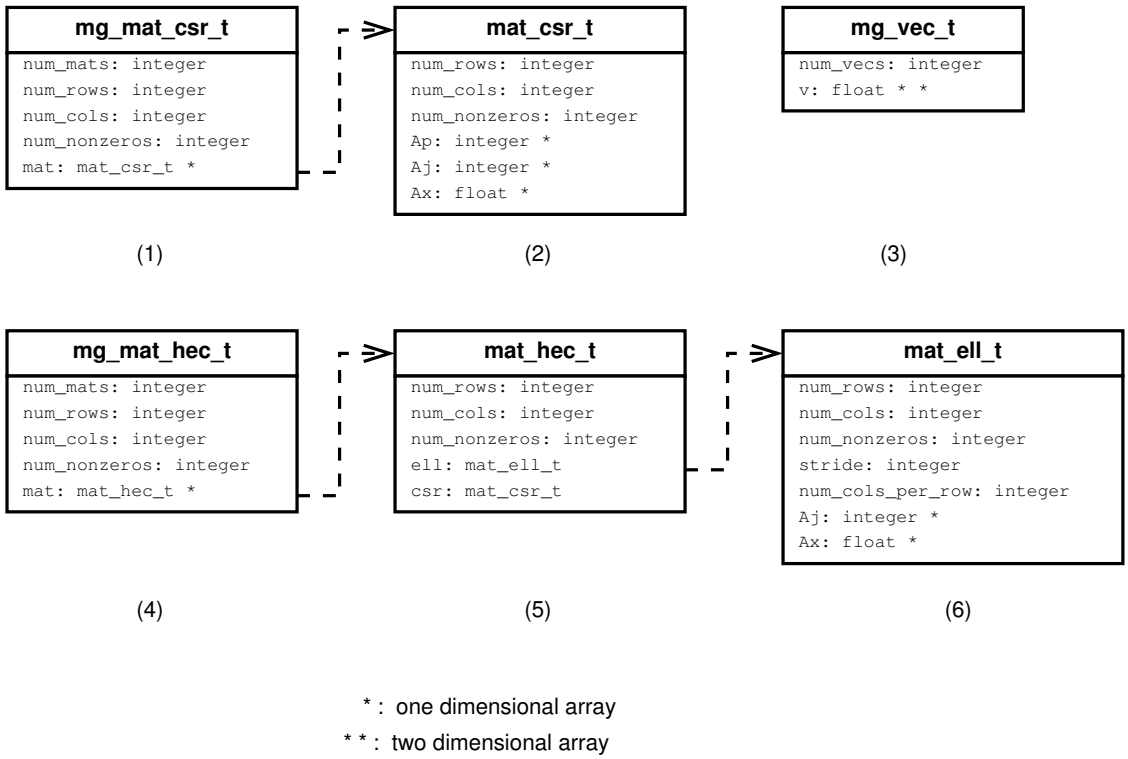


Figure 4.3: Data structure

Ax. The data structure of *mg_mat_csr_t* is designed for managing all domain matrices. The property *mat* is a array which contains pointers of all the domain matrices located in different GPUs. The property *num_mats* represents the number of the domain matrices which equals the number of the GPUs. The data structure *mg_vec_t* is designed to store domain vectors. The property *num_vecs* is the number of the domain vectors which equals the number of the GPUs. A two-dimensional pointer *v* is used to manage the set of the domain vectors. Each domain vector is stored in a corresponding GPU. The first level of *v* points to vectors. The second level of *v* points to the elements of each vector.

Before distributing the domain matrices onto different GPUs, we need to transform them into the HEC format which is friendly to design a SPMV algorithm [33]; see Figure 4.3-(5). An original domain matrix is cut into two parts. One part is regular by storing some necessary zero elements. The other part is an irregular part. Figure 4.4 gives a schematic of the HEC matrix format. The data structure *mat_ell_t* is designed for the regular part. The property *stride* represents the number

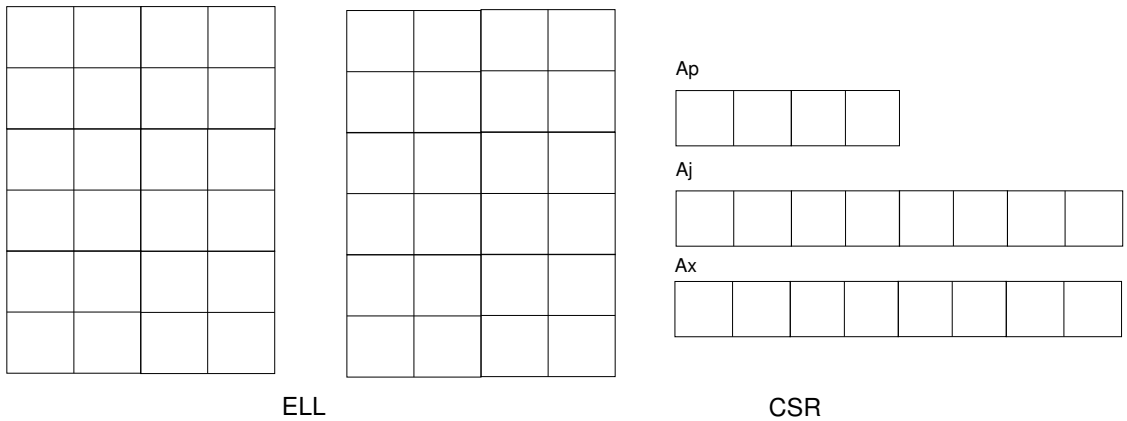


Figure 4.4: HEC matrix format

of the rows of the matrix. The property *num_cols_per_row* is used for controlling the length of each row. In this part, all the data are stored in a matrix transpose form. In other words, the data stream read from the array *Ax* is stored by a column-major order in the *ell* part. Thus it is suitable for fitting the data stream from the vector. In addition, GPU threads can process them batched. The data structure *mat_csr_t* is used for the irregular part. The data structure *mg_mat_hec_t* is designed for managing the domain matrices.

4.1.3 Reordering-Compact Method for Domain Matrices

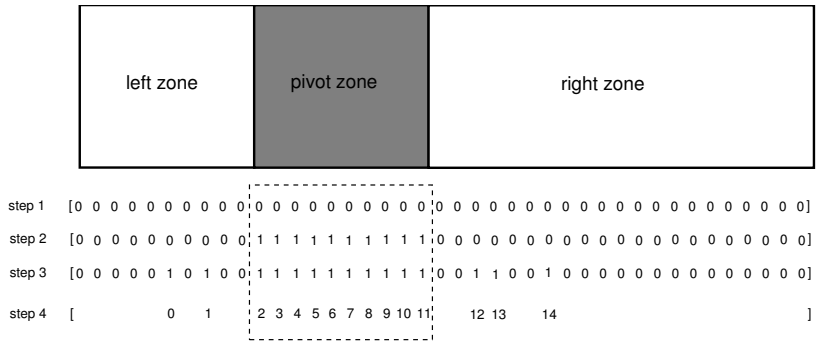


Figure 4.5: Example for reorder-compact method

A domain matrix can be divided into three parts; see the upper part in Figure 4.5. The pivot diagonal block is called the pivot zone which is the only dense part in a domain matrix. There are

sparse nonzero elements in the left and right zones. As stated above, a domain matrix is stored in the HEC format which is composed of *mat_ell_t* and *mat_csr_t*. All the nonzero elements have their original column indices which are discontinuous numbers. However, a vector uses a one-dimensional array as the data structure with continuous indices, which has a consecutive memory region and fast visiting speed. These two kinds of indices do not match. The solution is to use a local order for domain matrices and make all columns reach a compact effect. The reordering-compact method is described in Algorithm 7; also see the lower part in Figure 4.5. Because the nonzero elements in the left and right zones are sparse, the reorder-compact process costs less computing resources. It can be executed on the host after the domain decomposition phase.

Algorithm 7 Reordering-compact method

1: Establish an integer array with an initial value as zero. Its length is the same number of the columns in a domain matrix.

2: All elements corresponding to the pivot zone are assigned with one because the pivot zone is dense.

3: For the left or right zone, if a nonzero element appears in any rows, the corresponding array element is updated to one.

4: All the array elements with value one are renumbered sequentially starting from zero.

4.1.4 Communication Mechanism for Domain Vectors

Each GPU owns a domain matrix and a domain vector. Though a domain matrix is dense in its pivot block, it is possible for the domain matrix to have nonzero elements in any columns. Thus the whole vector data should be prepared for multiplication according to the SPMV algorithm in linear algebra. However, there is only one domain vector in each GPU, which is a segment and only contains elements used for the pivot block. If there are some nonzero elements appearing outside of the pivot block, extra vector elements must be fetched from other domain vectors which are located on other GPUs. The challenge concentrates on how to fetch necessary vector data

from other GPUs with the least communication. We design a communication mechanism which is described by the data structure and algorithm euqations. Through this mechanism, each GPU can fetch the vector data it needs from other GPUs. Conversely, each GPU can provide vector data for other GPUs to fetch.

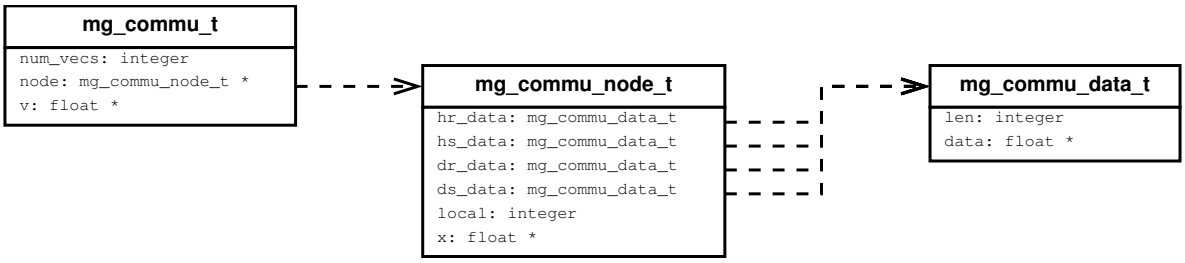


Figure 4.6: Data structure for communication mechanism

4.1.4.1 Data Structure

The data structure used for communication is shown in Figure 4.6. Each GPU owns a *mg_commu_node_t*. The properties *hr_data* and *hs_data* are the data receiving and data sending proxies on the host. The properties *dr_data* and *ds_data* are the data receiving and data sending proxies on the device. All data packages are designed as *mg_commu_data_t* and transmitted by a share cache on the CPU. The property *x* is the extended vector which is a local domain vector plus extra parts fetched from other GPUs. *mg_commu_t* is designed for managing the communication nodes. The property *v* acts as the share cache.

4.1.4.2 Assembling Communication Mechanism on Multiple GPU Architecture

The assembling chart of the communication mechanism is shown in Figure 4.7. An extended vector is combined by two parts; see figure 4.8. The first part comes from a local domain vector which lies in the middle of the extended vector. The start position is identified by property *local* in *mg_commu_node_t*. The second part contains the two sides of the extended vector, which are fetched from other domain vectors. The extended vector will be used for SPMV calculation on each GPU. We assume that *m* GPUs are employed. The extended vectors can be assembled by the

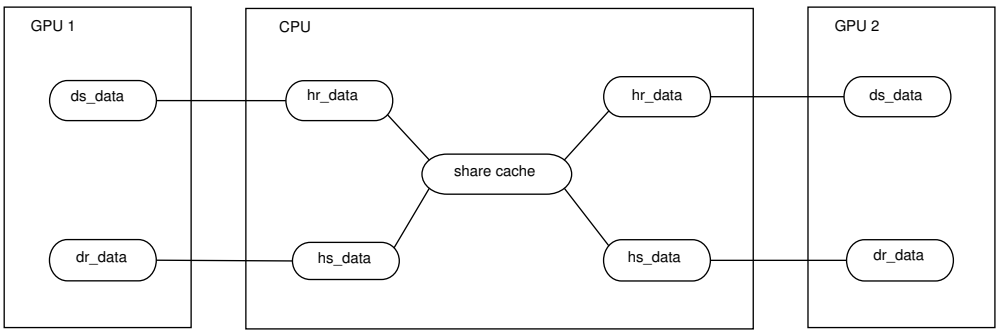


Figure 4.7: Communication chart

following equations:

$$D_{ri} = S_{li} \cup S_{ri} \tag{4.7}$$

$$\cup_{i=1}^{m} D_{ri} = \cup_{i=1}^{m} D_{si} \tag{4.8}$$

$$x = S_{li} \cup S_{pi} \cup S_{ri} \tag{4.9}$$

where

- m: the number of GPUs
- S_{pi} : the domain vector on ith GPU
- S_{li} : the left-hand side of the extended vector on ith GPU
- S_{ri} : the right-hand side of the extended vector on ith GPU
- D_{ri}: dr_data on ith GPU
- D_{si}: ds_data on ith GPU
- x: the extended vector on ith GPU

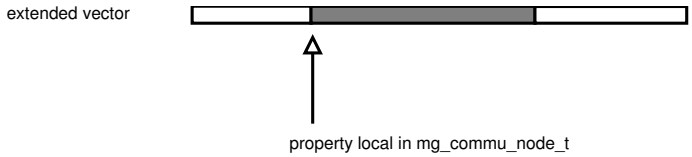


Figure 4.8: Extended vector

When determining dr_data by equation (4.7), the elements in S_{li} can be counted and stored as property $local$ in $mg\_commu\_node\_t$. By now, the SPMV on a multiple-GPU architecture has been divided into subtasks and distributed onto multiple GPUs.

4.2 Nested Restricted Additive Schwarz Framework

4.2.1 Theory Review

The nested RAS framework devotes to providing a highly parallel framework for assembling ILU preconditoner matrices. This subsection gives a brief review of the RAS theory and related ILU precondioner equations, which will be used in the following contents.

4.2.1.1 Restricted Additive Schwarz

$A = (A_{ij})$ is a nonsingular $n \times n$ sparse matrix. We define an graph $G = \{W, E\}$, where the set of vertices $W = \{1, \ldots, n\}$ represents the n unknowns. The edge set $E = \{(i, j) : A_{ij} \neq 0,\ A_{ij} \in A\}$ represents the pairs of vertices [24]. Then the graph G is divided into k non-overlapping subsets, denoted by $W_1^0, W_2^0, \ldots, W_k^0$. For any subset W_i^0, a 1-overlap subset W_i^1 can be generated by including all the direct neighboring vertices in W [24]. By this analog, a δ-overlap subset W_i^δ can be created, and the resulting overlapping subsets are $W_1^\delta, W_2^\delta, \ldots, W_k^\delta$. W_i^δ and the edges among the vertices in W_i^δ form a sub $N_i \times N_i$ matrix, denoted by A_i. All these sub-matrices can be assembled and an enlarged system is obtained:

$$M = \mathrm{diag}(A_1, A_2, \ldots, A_k) \tag{4.10}$$

Obviously, M is a $(N_1 + N_2 + \cdots + N_k) \times (N_1 + N_2 + \cdots + N_k)$ matrix whose system can be solved by an ILU preconditioner, for example. All these sub-matrices, $A_1, A_2, \ldots, A_k$, can be obtained simultaneously.

4.2.1.2 Related Preconditioner Equations

The linear system we solve is written as (4.11).

$$A\vec{x} = \vec{b} \tag{4.11}$$

where A is a nonsingular $n \times n$ sparse matrix, $\vec{x}$ is an unknown vector, and $\vec{b}$ is a right-hand side vector. Preconditioning algorithms all need to solve a linear system (4.12) at least once at each iteration. A common situation is that the preconditioner is available in the factorized form (4.13) where L and U are triangular matrices [7].

$$M\vec{y} = \vec{f} \tag{4.12}$$

where M is a preconditioner which is a nonsingular $n \times n$ sparse matrix, $\vec{y}$ is an unknown vector, and $\vec{f}$ is a right-hand side vector.

$$M = LU \tag{4.13}$$

where M is the preconditioner, L is a lower triangular matrix, and U is an upper triangular matrix.

4.2.2 Data Structure

In the nested RAS implementation, all matrix formats are CSR (Compressed Sparse Row). The data structures of storing matrices and vectors are shown in Figure 4.3-(1),(2) and (3). *mat_csr_t* is the data structure for a single matrix. *mg_mat_csr_t* is used to manage multiple matrices. *mg_vec_t* is designed to manage a set of vectors. We do not establish a specific data structure for a single vector, because a one-dimensional array is a suitable format for it. A detailed explanation of CSR data structure is referred to subsection 4.1.2.

4.2.3 Domain Decomposition

In Figure 4.9-(2), each horizontal rectangle stands for a domain matrix. According to the graph theory, the pivot element of each row can be looked at as a node. The nodes in the same domain matrix form a node set. The other nonzero elements of a row are looked at as communication

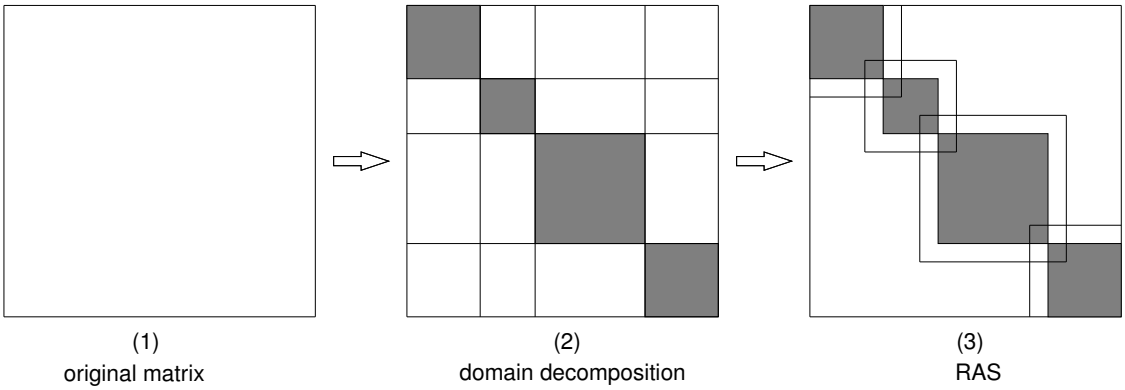


Figure 4.9: Domain decomposition and RAS

edges. If an element is located in a pivot diagonal block of a domain matrix, it has no influence on the parallelization between the current domain matrix and others. However, if an element is located outside of a pivot diagonal block, it represents a communication edge between two domain matrixes. According to the RAS theory, the nonzero elements outside of the diagonal blocks will be discarded and the diagonal blocks will be used for parallel computing. Removing the nonzero elements from the original matrix leads to some data information loss. Hence the nonzero elements outside of the diagonal blocks should be reduced to the least, which means that the relationship between the rows within the same domain matrix ought to be as tight as possible.

The partition methods are illustrated in Figure 4.2. As described in Section 4.1.1, a sequence domain partition method is used for a regular structure matrix that usually comes from the FDM (Finite Difference Method) or FVM (Finite Volume Method). The special partition method is used for an unstructured matrix derived from the FVM or FEM (Finite Element Method). We use METIS as a partition method, which is suitable for both structured and unstructured matrices [48]. Figure 4.9-(2) denotes the remaining diagonal blocks after discarding the sparse nonzero elements in the non-diagonal blocks.

4.2.4 Overlapped Diagonal Block Matrices

The first step of RAS creates a series of square block matrices by discarding a few nonzero elements, which improves the degree of parallelization but loses the calculation accuracy due to the missed data information. This can be compensated to a certain extent by the overlap mechanism discussed below.

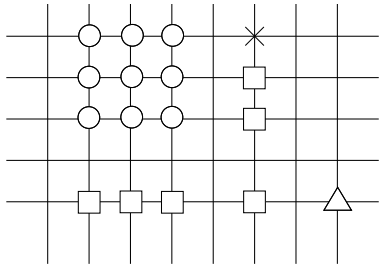


Figure 4.10: Overlap example

Figure 4.10 shows a schematic of one level overlap. The horizontal lines represent the rows of the original matrix. The vertical lines represent the columns of the original matrix. The circle symbol set represents a diagonal block. If a nonzero element is found outside of it, the block should be extended according to the new found element. For example, if the symbol x is a nonzero element, the diagonal block should be extended by both the column of x and the row which has the same index as the column. All circle symbols, the square symbols and the x symbols form the resulting block matrix of the first level overlap, which is an extended square matrix. Because new lines are added, it is possible for a few nonzero elements to appear outside of the extended matrix on these new lines in the original matrix. These nonzero elements will be used for the next overlap level. If the next overlap level is not needed, these nonzero elements are just discarded. In this example, only one overlap level is under consideration. As illustrated in Figure 4.10, from the new added line, a triangle symbol, which is a nonzero element that lies outside of the overlapped block, can be found. Because we only need the first level overlap, the triangle is discarded.

Figure 4.9-(3) shows the overlapped diagonal blocks. In other words, the domain matrices in Figure 4.9-(2) are all rectangles but all the domain matrices in Figure 4.9-(3) are cut into squares

and then enlarged by overlapping.

The diagonal blocks can be stored in the data structure *mat_csr_t*. The global matrix in Figure 4.9-(3) can be managed by *mg_mat_csr_t*. The procedure of generating overlapped blocks is described in Algorithm 8.

Algorithm 8 Generate overlapped blocks

```
input
level                                                    ▷ overlap levels
for i = 1 : level do
    for each extended row of i − 1 th level overlapped block do
        for each nonzero element outside block do
            Extend one row and one column
        end for
    end for
end for
```

4.2.5 Row Tracing Mappings

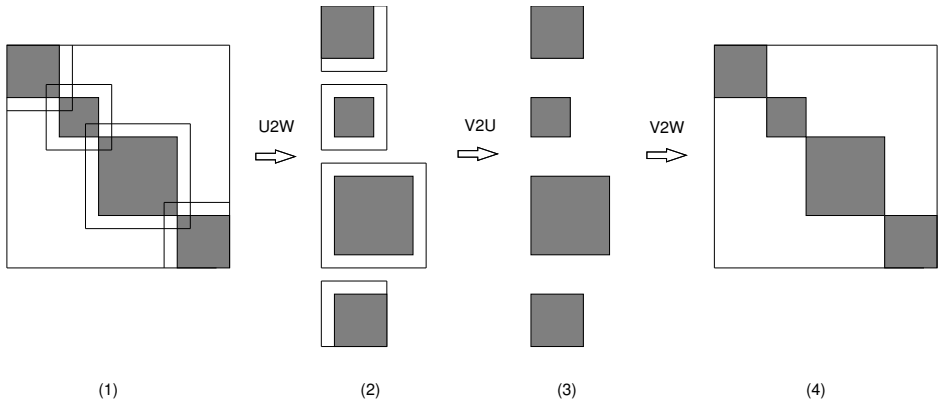


Figure 4.11: Row tracing mappings

Figure 4.11-(1) shows the matrix schematic after the above overlapping process. All these overlapped blocks will be distributed onto different GPUs to serve as the coefficient matrix of a precon-

ditioner system; see Figure 4.11-(2). This means that all these overlapped blocks should be separated. For an individual sub-matrix, the row indices are renumbered from zero. The new index of a row may be different from its index in the global matrix. Therefore, mappings of the row indices need to be created for tracing the row data. We define $U = \{i: \text{ row indices of overlapped block matrices}\}$, $V = \{j: \text{ row indices of block matrices}\}$, and $W = \{k: \text{ row indices of the original matrix}\}$. Here we assume that each set is sorted in ascending order according to the row indices. Now, three mappings can be defined as follows:

$$k = U2W(i) \tag{4.14}$$

$$i = V2U(j) \tag{4.15}$$

$$k = V2W(j) \tag{4.16}$$

where $i \in U, j \in V$ and $k \in W$.

Figures 4.11-(1) and 4.11-(2) show that the global matrix is divided into overlapped blocks. The original location of each row in an overlapped block can be found by mapping $U2W$. After solving a precondtioner system, the solution sub-vectors on multiple GPUs must be restored to the original location in the global vector. Therefore, the mapping of row indices between the non-overlapped blocks and the global matrix is necessary. We establish mapping $V2W$ between the non-overlapped blocks and the global matrix; see Figures 4.11-(3) and 4.11-(4). The row relationship between the overlapped blocks and the non-overlapped blocks is recorded by mapping $V2U$. All the mappings can be established in Algorithm 9.

4.2.6 Outer RAS and Inner RAS (Nested RAS)

In order to make the most use of the parallel capacity provided by a multiple GPUs platform, a nested RAS framework is designed, which contains two layers named an outer RAS and an inner RAS, respectively. The outer RAS corresponds to the multiple-GPU level which provides coarse-grained parallelization. The inner RAS is designed for fine-grained parallelization at a GPU-thread

Algorithm 9 Generation of mappings

```
1:  input
2:  block                                   ▷ non-overlapped block located in overlapped block
3:  overlappedBlock                                                          ▷ overlapped block
4:  for each row of overlappedBlock do
5:      Create a U2W mapping entry
6:      if row is in block then
7:          Create a V2W mapping entry
8:          Create a V2U mapping entry
9:      end if
10: end for
```

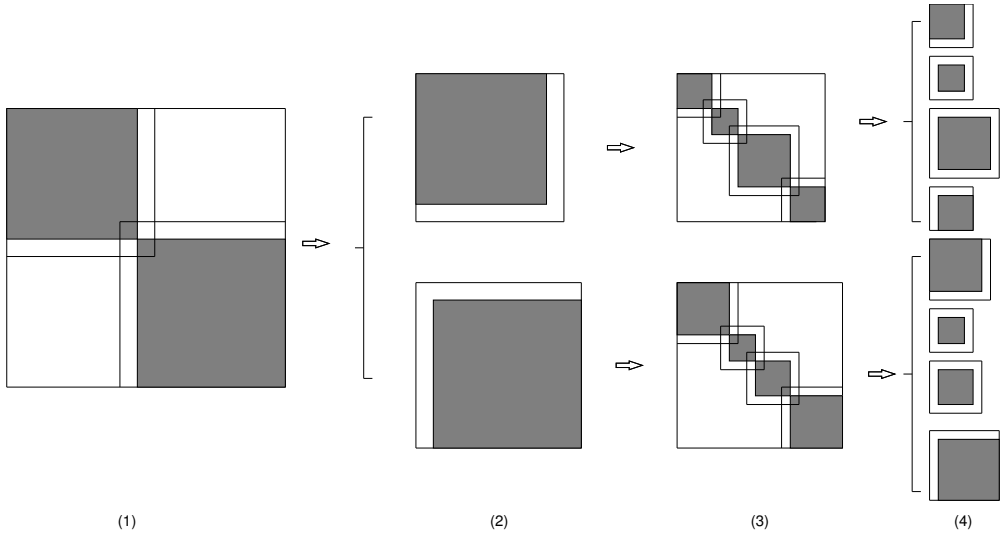


Figure 4.12: Nested RAS

level. The global matrix is shown in Figure 4.12-(1). Through the outer RAS procedure, the global matrix is divided into outer overlapped blocks which have the same number as GPUs. Each GPU acquires an outer overlapped block; see Figure 4.12-(2). These blocks are still nonsingular square matrices and do not have any communication between each other. Each GPU can solve its own outer overlapped block independently. The outer overlapped blocks could be factorized by ILU

and then solved by a parallel triangular solver. In order to make maximal use of the parallel capacity on each GPU, we apply the inner RAS to the outer overlapped blocks to improve the degree of parallelism. The principle of the inner RAS is the same as that of the outer RAS, as discussed above. Each outer overlapped block acts as a global matrix. The inner overlapped blocks produced from one outer overlapped block are on the same GPU and can be solved concurrently. The schematic is shown in Figures 4.12-(3) and 4.12-(4). Algorithm 10 states the process of the nested RAS.

Algorithm 10 Nested RAS

```
1:  input
2:  Global matrix
3:  Outer partition by METIS
4:  Generate outer overlapped blocks by Algorithm 8
5:  Generate outer mappings by Algorithm 9
6:  for each outer overlapped block do
7:      Inner partition by METIS
8:      Generate inner overlapped blocks by Algorithm 8
9:      Generate inner mappings by Algorithm 9
10: end for
```

After the nested RAS decomposition, the original global matrix is transferred to many groups of inner overlapped blocks. Each group of blocks is on a same GPU and has better capability for parallel computing. In order to apply a parallel triangular solver to solve the sub-systems on each GPU, the matrix on the GPU needs to be factorized into a lower triangular matrix and an upper triangular matrix by ILU; see equation (4.13). There are two ways to implement this factorization. One way is to factorize each inner overlapped block individually and then put all small lower and upper triangular matrices together to form the resulting lower and upper triangular matrices. The other way is first to put all inner overlapped blocks head-to-head and then perform an ILU

factorization on the resulting matrix directly. Because there is no relationship between any two inner overlapped blocks, either way is reasonable. We select the first way; see Figure 4.13.

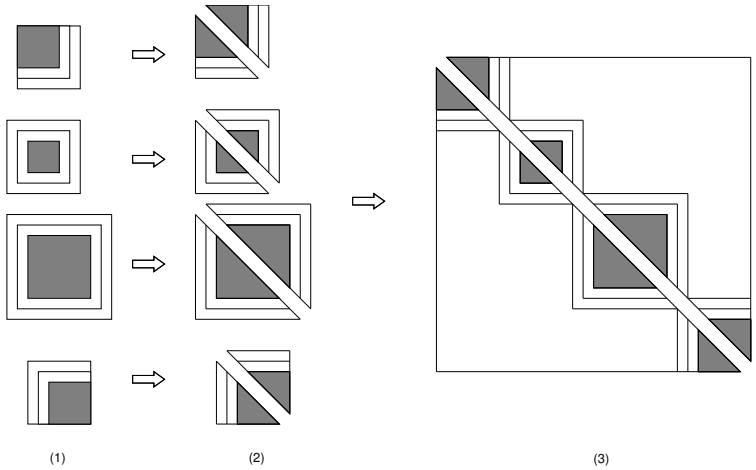


Figure 4.13: ILU for inner overlapped blocks

4.2.7 Right-Hand Side Vector Overlap and Solution Vector Recovery

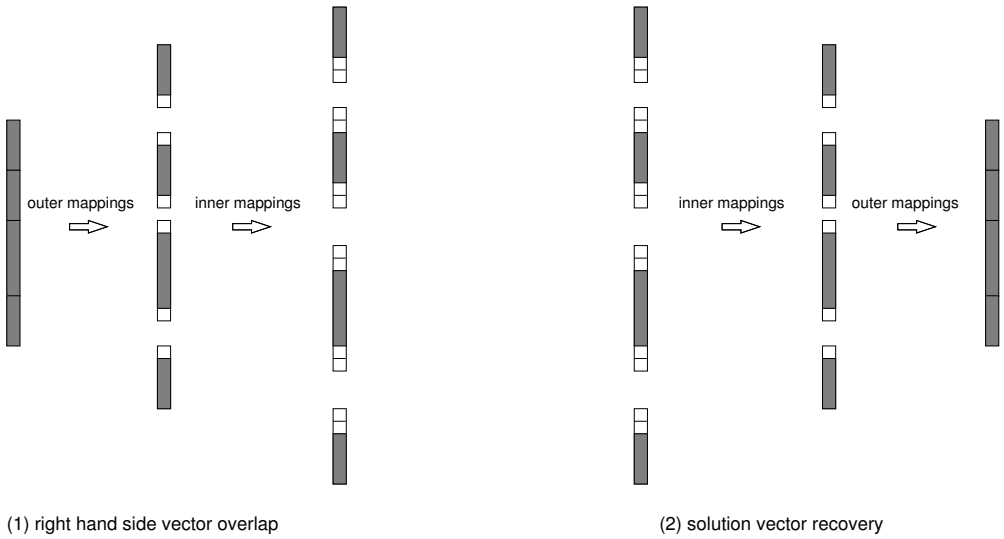


Figure 4.14: Right-hand side vector overlap and solution vector recovery

The preconditiner matrix M has been established and factorized into L and U; see equations (4.12) and (4.13). M is stored in *mg_mat_csr_t* format and each GPU has a sub-system to solve. To

solve this precondtioner system, the right-hand side vector must be divided into sub-vectors and each sub-vector is overlapped by the same outer and inner mappings as discussed above. Since a preconditioner system is solved on a single GPU, the right-hand side vector only needs to be divided into outer sub-vectors. It is not necessary to divide the right-hand side vector into inner ones because the vector elements do not have any relationship that affects parallel performance between each other as matrix rows have. But the inner mappings should still be employed to generate the overlapped right-hand side vector for the preconditoner system; see Figure 4.14-(1).

Because the matrix and right-hand side vector of a preconditioner system are enlarged by outer and inner overlaps, the solution vector is also overlapped. Therefore, after the preconditioner system is solved, a similar but inverse procedure needs to be performed on the solution vector. The recovery procedure for the solution vector is illustrated in Figure 4.14-(2).

By now, we have finished the construction of the nested RAS framework. In summary, we divide the original matrix A into small matrices to improve the parallel performance through a nested RAS process. These small matrices can be further factorized into triangular matrices by ILU or other preconditioner algorithms. We use the parallel triangular solver to solve triangular matrices on each GPU. In Section 4.3, we will give a detailed description of the ILU(k) algorithm and the parallel triangular solver.

4.3 Decoupled ILU(k) and Parallel Triangular Solver

4.3.1 Data Structure

For ILU(k) factorization, we still store matrices in the CSR format which is implemented by the data structure *mat_csr_t*; see Figure 4.3-(2). If ignoring the concrete values of entries in a matrix and only considering whether they equal zero or not, all the nonzero positions of a matrix will form a nonzero pattern. Obviously, an entry is a nonzero element if its row and column indices are involved in Ap and Aj; otherwise, it is a zero element. Therefore, Ap and Aj also represent the nonzero pattern of a matrix or the structure of a matrix.

Figure 4.15 shows data structure *imatcsr_t* which is specially designed for storing a nonzero pattern and will be applied in the symbolic phase of ILU(k). n represents the dimension of a matrix. nz is an array for the length of each row. The two-dimensional array Aj is used for storing column indices. As stated above *mat_csr_t* stores column indices in a one-dimensional array. If an element is deleted from the matrix, the entire elements after the current element in the memory space must be moved forward, which costs a large amount of computing resources, especially when the deletion operation happens frequently. However, because the column indices are stored in the two-dimensional array Aj of *imatcsr_t* row by row and each row contains sparse elements, a very low cost of moving data is required for the deleting operation. Therefore, *imatcsr_t* is more suitable for manipulating a nonzero pattern.

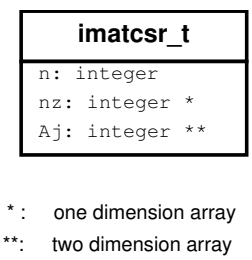


Figure 4.15: Data structure for nonzero pattern

4.3.2 Decoupled ILU(k) Algorithm

Algorithm 11 shows the ILU(0) algorithm. P represents the nonzero pattern of matrix A. If the code $(i, p) \in P$ on line 2 and $(i, j) \in P$ on line 4 are removed, this algorithm is Gaussian elimination. As we know, Gaussian elimination is costly on computing resources. In the ILU(0) algorithm, P serves as a filter where only the elements lying in the nonzero positions of P can be calculated by lines 3 and 5. Due to the sparse character of P, the complexity of this algorithm is very low, which can be performed at each iteration of a Krylov subspace solver. Matrix A is stored in *mat_csr_t* format. In this algorithm, the resulting matrix also uses the same memory as A, which means that the original elements of A will be replaced by the resulting elements calculated by lines 3 and 5.

Algorithm 11 ILU(0) factorization

1: **for** i = 2: n **do**
2: **for** p = 1: i - 1 & $(i, p) \in P$ **do**
3: $A_{ip} = A_{ip}/A_{pp}$
4: **for** j = p + 1: n & $(i, j) \in P$ **do**
5: $A_{ij} = A_{ij} - A_{ip}A_{pj}$
6: **end for**
7: **end for**
8: **end for**

The lower triangular matrix L from factorization will be stored in A's low triangular part except the diagonal which are all unit values. L is generated by line 3. The upper triangular matrix U from factorization will be stored in A's upper triangular part including the diagonal. U is generated by line 5. Figure 4.16 shows a schematic of ILU(0) factorization. Because P comes from the nonzero pattern of A, Figure 4.16-(2), which represents the resulting matrix, has the same nonzero pattern as Figure 4.16-(1).

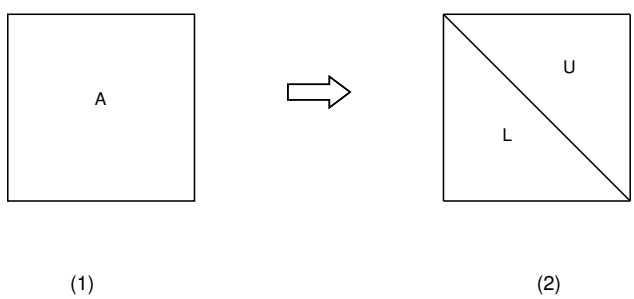


Figure 4.16: ILU factorization

Because A is a sparse matrix, the nonzero pattern P of A is also sparse. The accuracy of ILU(0) may be insufficient to yield an adequate rate of convergence. More accurate incomplete LU factorizations are often more efficient as well as more reliable. These more accurate factorizations differ from ILU(0) by allowing some fill-in. A level of k is defined to control the degree of fill-in [7]. A larger k allows more fill-ins in the nonzero pattern in addition to the original nonzero pattern

created by A. k is a nonnegative number; ILU(k) is ILU(0) when k is zero.

$$L_{ij} = \begin{cases} 0, & (i,j) \in P \\ \infty, & (i,j) \notin P. \end{cases} \tag{4.17}$$

$$L_{ij} = min\{L_{ij}, L_{ip} + L_{pj} + 1\}. \tag{4.18}$$

ILU(k) requires that a level is defined for each entry of matrix A. The initial level of each entry A_{ij} is defined by formula (4.17), where P represents the nonzero pattern of matrix A. The nonzero elements of A have a level zero; otherwise, it has a level of infinity. Formula (4.18) gives the level updated algorithm [7]. Apparently, formula (4.18) has no influence on the level of a nonzero element, because the level of a nonzero element always remains zero. This means that only zero element's level can be updated from infinity to a limited positive value. If the new level value is still greater than level k, this entry will be kept as zero and the position of this entry is not involved in the next factorization. In other words, only the positions that have a new level value smaller than level k can continue to a next calculation. If this position continues to the end, it will be considered as a fill-in position included in the extended nonzero pattern. Because the nonzero pattern generated by ILU(k) has more nonzero positions than ILU(0), the accuracy of solution for a preconditioner system improves. Algorithm 12 gives the whole procedure of ILU(k).

Apparently, the calculations of A_{ip}, A_{ij} and L_{ij} are mixed together in Algorithm 12, which complicates the algorithm. There is no direct relationship between A_{ip}, A_{ij} and L_{ij}. L_{ij} merely acts as a filter to control whether A_{ip} or A_{ij} is qualified to be calculated in the following steps. Because different functions are compounded into one subroutine to implement, the design of Algorithm 12 decreases the maintainability of the codes. From another perspective, line 6 is not necessary to be calculated when L_{ij} is greater than k because A_{ij} is set to zero on line 11. In other words, if an element A_{ij} cannot stay to the end to be a fill-in element, the calculation for A_{ij} on lines 6 is unnecessary. Thus line 6 wastes some computing resources in some situations. If the functions of the zero pattern creation and the ILU factorization can be separated, the problems stated above

Algorithm 12 ILU(k) factorization

```
1:  For all nonzero entries in nonzero pattern P define L_ij = 0
2:  for i = 2 : n do
3:      for p = 1 : i − 1 & L_ip ≤ k do
4:          A_ip = A_ip / A_pp
5:          for j = p + 1 : n do
6:              A_ij = A_ij − A_ip A_pj
7:              L_ij = min{L_ij, L_ip + L_pj + 1}
8:          end for
9:      end for
10:     if L_ij > k then
11:         A_ij = 0
12:     end if
13: end for
```

can be resolved. It is more clear and easier to process if Algorithm 12 is decoupled instead of implementing everything in one subroutine. Therefore, for both the aspects of designing a favorable maintainable program and a performance optimization algorithm, a decoupled ILU(k) implementation is necessary. The decoupled ILU(k) contains two steps. The first step is called the symbol phase focusing on establishing a fill-in nonzero pattern, which has no relation to the processing of any concrete data values. The second phase is responsible for the ILU factorization which uses the nonzero pattern established in the symbolic phase to factorize the original matrix into a lower triangular part and an upper triangular part.

The symbolic phase can be presented in Algorithm 13, where P' is designed to record the fill-in nonzero pattern. The data structure *tm_ilu_t* is applied to P', which can easily and efficiently remove entries. The logic for computing L_{ij} is the same as in Algorithm 12. Algorithm 13 has been separated totally from the processing of concrete data values. The factorization phase is

Algorithm 13 Symbolic phase

```
1:  For all nonzero entries in nonzero pattern P define L_ij = 0
2:  Define P' as n × n nonzero pattern
3:  Initiate P' by full filling with entries
4:  for i = 2 : n do
5:      for p = 1 : i − 1 & L_ip ≤ k do
6:          for j = p + 1 : n do
7:              L_ij = min{L_ij, L_ip + L_pj + 1}
8:          end for
9:      end for
10:     if L_ij > k then
11:         Remove entry ij from P'
12:     end if
13: end for
```

responsible for creating L and U according to the nonzero pattern P'. Because *tm_ilu_t* only stores the pattern without actual data values, in order to apply the ILU factorization, it is necessary to create a matrix A' by *mat_csr_t* from the nonzero pattern P', which contains both structure and data values. All entries are copied from A to A' since all the entry positions of A are remained in P'. The fill-in positions of A' are all filled with a value zero. Thus A' has the same structure as P' but has no essential data value difference from A. The ILU(0) algorithm can be directly applied to A' and so it is called the factorization phase; see Algorithm 11. The process of the factorization phase is also illustrated in Figure 4.16. The only difference from ILU(0) for A is that the number of nonzero elements in Figure 4.16-(2) is greater than the number of nonzero elements in A, because extra fill-in elements are imported by ILU(k).

4.3.3 Parallel Triangular Solver and Level Schedule Method

Each GPU owns a set of the ILU(k) outcomes of the nested RAS and a corresponding overlapped right-hand side vector. They form the preconditioner system. We use the parallel triangular solvers to solve it [51]. These parallel triangular solvers employ a level schedule method [7, 28]. The idea is to group unknowns $x(i)$ into different levels so that all the unknowns within the same level can be computed simultaneously [7, 28]. Because an upper triangular system can be easily converted to a lower triangular system, only a lower triangular system is analyzed bellow.

For the lower triangular linear system (4.19), the level of x_i $(1 \leq i \leq n)$ is defined in equation (4.20).

$$L\vec{x} = \vec{b} \tag{4.19}$$

where

- L : lower triangular matrix
- $\vec{x}$: unknown vector
- $\vec{b}$: right-hand side vector

$$l(i) = 1 + \max_{j} l(j) \tag{4.20}$$

where

- for all j such that $L_{ij} \neq 0, i = 1, 2, \ldots, n$
- L_{ij} : the (i, j)th entry of L
- $l(i)$: ith level, zero initially
- n : the number of rows.

The level schedule method is described in Algorithm 14. For GPU computing, each level in this algorithm can be parallelized.

The lower and upper triangular matrices come from a nested RAS resulting matrix by ILU(k). If the resulting matrix has higher parallel structure, so do the lower and upper triangular matrices.

Algorithm 14 Level schedule method for solving a lower triangular system

```
1: input
2: n                                                    ▷ the number of levels
3: for i = 1 : n do
4:     for each row in current level do
5:         Solve the current row
6:     end for
7: end for
```

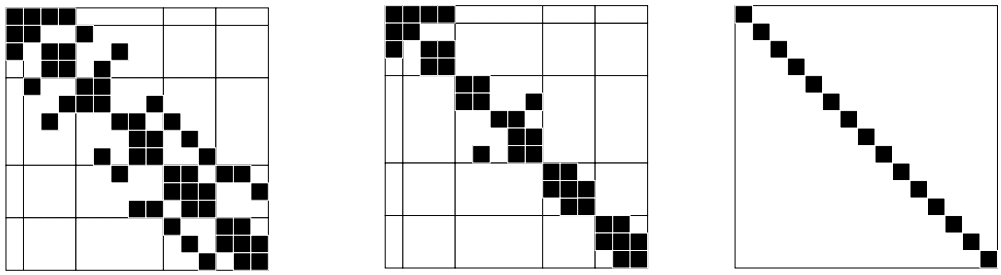


Figure 4.17: Inner RAS for level schedule method

If the level schedule method acquires more rows at each level, better parallel performance will be expected. Figure 4.17-(1) shows an example of a resulting matrix. Each row of this matrix represents the coefficients of an equation. Apparently, these equations have tight relationships and most of them cannot be solved simultaneously. In order to use the high parallel computing ability of GPU, some nonzero elements need to be removed from the matrix to improve the parallel structure. The extreme way is to discard all the elements except the diagonal elements as in Figure 4.17-(3). Ideal parallel performance can be acquired and all equations can be solved concurrently because there is only one level. However, it may not reach convergence or needs far more iterations to reach convergence due to the very poor accuracy. A compromise method divides a matrix into sub-domain matrices and discards the nonzero elements outside of the diagonal blocks as shown in Figure 4.17-(2). The number of sub-domains can be used to control the parallelization degree. This provides a theoretical basis for the inner RAS, as discussed in Section 4.2.

In this chapter, the crucial implementation mechanism and corresponding algorithms for the GMRES with ILU(k) are explained in detail. The SPMV is the core operation in the GMRES algorithm. The nested RAS constructs the framework for the ILU(k) preconditioner on the multiple GPU architecture. The decoupled ILU(k) algorithm is convenient for coding and improving efficiency. At last, we introduce the principle of the level schedule method and its implementation, namely, the parallel triangular solver. In the next chapter, a series of numerical experiments are presented and analyzed, which validate all these algorithms.

Chapter 5

NUMERICAL EXPERIMENTS

We designed three sections of numerical experiments to test the performance of our algorithms. They are experiments of SPMV, experiments of the nested RAS framework and experiments of GMRES with ILU(k). The workstation is established by an Intel Xeon X5570 CPU and four NVIDIA Tesla C2050/C2070 GPUs. The operating system is CentOS X86_64 with CUDA 5.1 and GCC 4.4. All cases are run in double precision. All data results are calculated by average on three runs. In order to determine the parallel performance of the algorithms on multiple GPUs, some contrastive algorithms are implemented for a CPU. The CPU codes are all compiled with -O3 option and only one thread is employed. An algorithm speedup on GPUs against a CPU is calculated by tc/tg where tc is the CPU time and tg is the GPU time, which shows how many times acceleration GPUs can achieve relative to a CPU.

5.1 SPMV

The SPMV algorithm performance on the multiple-GPU architecture is tested by a series of matrices which are listed in Table 5.1. The number of rows, the number of nonzero elements and the ratio of the number of nonzero elements to the number of rows are listed in the columns of $\# \; of \; rows$, $Nonzeros$ and NNZ/N, respectively. The matrix $3D\_Poisson$ comes from a three-dimensional poisson equation. The matrix $SPE10$ comes from a classical reservoir model. The other matrices are all downloaded from a matrix market [54]. All these matrices are nonsingular. A SPMV algorithm for a CPU is implemented as a reference. We use only one thread in the CPU algorithm for the reference purpose. Our SPMV algorithm uses the HEC format to store matrices on GPUs. In order to test the performance of the HEC format, a reference experiment about the HYB format is also presented, which uses the same SPMV algorithm except the matrix format.

Matrix	# of Rows	Nonzeros	NNZ/N
BenElechi1	245,874	6,698,185	27.24
af_shell8	504,855	9,042,005	17.91
parabolic_fem	525,825	2,100,225	3.99
tmt_sym	726,713	2,903,837	4.00
ecology2	999,999	2,997,995	3.00
thermal2	1,228,045	4,904,179	3.99
atmosmodd	1,270,432	8,814,880	6.94
atmosmodl	1,489,752	10,319,760	6.93
Hook_1498	1,498,023	30,436,237	20.32
G3_circuit	1,585,478	4,623,152	2.92
3D_Poisson	1,728,000	12,009,600	6.95
kkt_power	2,063,494	7,209,692	3.49
SPE10	2,188,851	29,915,573	13.67
memchip	2,707,524	13,343,948	4.93

Table 5.1: Matrices used for testing SPMV

5.1.1 SPMV Algorithm Performance

The running results are collected in Table 5.2. The $CPUtime$ lies in the first column. The $GPUtime$ for a different number of GPUs is listed in other columns. The corresponding speedup is calculated and shown in Table 5.3. Figure 5.1 provides a scatter diagram of speedup vs. the GPU number which is used for the study of the performance scalability when employing more GPUs. The rows of the tested matrices range from 240,000 to 2,700,000, which are a wide range of sparse matrices.

The matrices $BenElechi1$, af_shell8, $parabolic\_fem$, tmt_sym and $ecology2$ are matrices which have rows smaller than one million. Their CPU times are 0.01213, 0.01754, 0.00635, 0.00824 and 0.00877, respectively. When employing one GPU, their running times are reduced to 0.00084,

0.00167, 0.00048, 0.00053 and 0.00049. The speedups are 14.40, 10.53, 13.29, 15.67 and 17.85, respectively. When employing two GPUs, their running times are reduced to 0.00049, 0.00093, 0.00032, 0.00037 and 0.00037. The speedups are 24.60, 18.87, 19.73, 22.48 and 23.94. When employing three GPUs, their running times are reduced to 0.00035, 0.00067, 0.00024, 0.00027 and 0.00026. The speedups are 34.42, 26.31, 26.85, 30.61 and 33.48. When employing four GPUs, their running times are reduced to 0.00028, 0.00051, 0.00020, 0.00022 and 0.00021. The speedups are 43.36, 34.15, 31.93, 38.23 and 41.57, respectively.

Obviously, the speedup effects increase with the number of GPUs increasing. The maximal speedups are all over 30 when four GPUs are employed. From Figure 5.1, the curves of $BenElechi1$, af_shell8, $parabolic\_fem$, tmt_sym and $ecology2$ all display a favorable linear effect. This reveals our SPMV algorithm has good scalability for these matrices which are under one million rows.

The matrices $thermal2$, $atmosmodd$, $atmosmodl$, $Hook\_1498$, $G3\_circuit$, $3D\_Poisson$, kkt_power, $SPE10$ and $memchip$ are in the range from one million to nearly three million rows. Their CPU times are 0.01930, 0.01767, 0.02136, 0.05723, 0.01342, 0.02630, 0.02315, 0.05686 and 0.03342, respectively. When employing one GPU, their running times are reduced to 0.00203, 0.00125, 0.00152, 0.00642, 0.00099, 0.00156, 0.00366, 0.00475 and 0.00314. The speedups are 9.49, 14.10, 14.07, 8.91, 13.61, 16.85, 6.32, 11.96 and 10.63. When employing two GPUs, their running times are reduced to 0.00116, 0.00080, 0.00096, 0.00340, 0.00063, 0.00108, 0.00195, 0.00253 and 0.00180. The speedups are 16.64, 22.14, 22.31, 16.84, 21.21, 24.36, 11.85, 22.50 and 18.55. When employing three GPUs, their running times are reduced to 0.00082, 0.00072, 0.00080, 0.00247, 0.00045, 0.00099, 0.00138, 0.00190 and 0.00127. The speedups are 23.57, 24.49, 26.72, 23.19, 30.11, 26.48, 16.73, 29.89 and 26.30. When employing four GPUs, their running times are reduced to 0.00061, 0.00066, 0.00070, 0.00189, 0.00034, 0.00095, 0.00107, 0.00161 and 0.00103. The speedups are 31.44, 26.59, 30.63, 30.31, 38.93, 27.76, 21.58, 35.28 and 32.41, respectively.

For these matrices which have over one million rows, the SPMV algorithm also displays a

direct proportion effect with the number of GPUs employed. The maximal speedups can still reach over 20 when four GPUs are employed. The speedups for matrices under one million rows range from 31 to 43 on four GPUs, but for matrices over one million rows range from 21 to 38. This can be explained that the cost of communication among GPUs increases as the number of rows increases. From Figure 5.1, the curves of *thermal*2, *atmosmodl*, *Hook_*1498, *G3_circuit*, *kkt_power*, *SPE*10 and *memchip* are close to a straight line, which represents good performance of scalability. However, the matrices *atmosmodd* and 3*D_Poisson* have low speedup improvement when more than two GPUs are employed, which is caused by their concrete matrix patterns.

From the aspect of NNZ/N, the matrices *BenElechi*1, *af_shell*8, *Hook_*1498 and *SPE*10 have a high NNZ/N of 27.24, 17.91, 20.32 and 13.67 and can be sped up to 43.36, 34.15, 30.31 and 35.28 on four GPUs, respectively. Another set of matrices *parabolic_fem*, *tmt_sym*, *ecology*2, *thermal*2, *G3_circuit*, *kkt_power* and *memchip*, which have a low NNZ/N less than 5, can also be sped up to 31.93, 38.23, 41.57, 31.44, 38.93, 21.58 and 32.41 on four GPUs. These show that the SPMV algorithm can be adapted to different values of NNZ/N.

In Figure 5.1, the speedup vs. GPU number curves are drawn according to all the matrices and each matrix provides a group of four points. Obviously, most groups of points fit a straight line well. Due to the difference of matrix patterns and sizes, the slopes have a little difference but the general tendency of them is almost consistent. Therefore, Figure 5.1 demonstrates that the SPMV algorithm has good scalability for the current four GPUs available.

In summary, all these matrices represent a common scope of large-scale matrices. The lowest speedup happens on the matrix *kkt_power* on four GPUs, which is 21.58. The highest speedup 43.36 is obtained on matrix *BenElechi*1 on four GPUs. Though the speed of SPMV is decided by a combination effect of the number of rows, the number of nonzeros and the matrix pattern, the experiments show that all the matrices can be sped up to 20 to 40 times faster than they are run on a traditional CPU.

Matrix	CPU time (second)	1 GPU time (second)	2 GPUs time (second)	3 GPUs time (second)	4 GPUs time (second)
BenElechi1	0.01213	0.00084	0.00049	0.00035	0.00028
af_shell8	0.01754	0.00167	0.00093	0.00067	0.00051
parabolic_fem	0.00635	0.00048	0.00032	0.00024	0.00020
tmt_sym	0.00824	0.00053	0.00037	0.00027	0.00022
ecology2	0.00877	0.00049	0.00037	0.00026	0.00021
thermal2	0.01930	0.00203	0.00116	0.00082	0.00061
atmosmodd	0.01767	0.00125	0.00080	0.00072	0.00066
atmosmodl	0.02136	0.00152	0.00096	0.00080	0.00070
Hook_1498	0.05723	0.00642	0.00340	0.00247	0.00189
G3_circuit	0.01342	0.00099	0.00063	0.00045	0.00034
3D_Poisson	0.02630	0.00156	0.00108	0.00099	0.00095
kkt_power	0.02315	0.00366	0.00195	0.00138	0.00107
SPE10	0.05686	0.00475	0.00253	0.00190	0.00161
memchip	0.03342	0.00314	0.00180	0.00127	0.00103

Table 5.2: SPMV algorithm running time for HEC format

5.1.2 Comparison with HYB Format

The HYB format is a hybrid of ELL and COO, which is designed by Bell and Garland [31, 32]. We use the HYB format as a reference to test the effect of the HEC format. The SPMV algorithm on multiple GPUs remains the same with only the HYB format instead of the HEC format. All the matrices listed in Table 5.1 are tested by this algorithm. Table 5.4 collects the running time on a CPU or GPUs. Table 5.5 gives the speedup results. Figure 5.2 provides a scatter diagram of speedup vs. the GPU number.

Because the algorithm remains unchanged, the difference only comes from the matrix format.

Matrix	1 GPU (speedup)	2 GPUs (speedup)	3 GPUs (speedup)	4 GPUs (speedup)
BenElechi1	14.40	24.60	34.42	43.36
af_shell8	10.53	18.87	26.31	34.15
parabolic_fem	13.29	19.73	26.85	31.93
tmt_sym	15.67	22.48	30.61	38.23
ecology2	17.85	23.94	33.48	41.57
thermal2	9.49	16.64	23.57	31.44
atmosmodd	14.10	22.14	24.49	26.59
atmosmodl	14.07	22.31	26.72	30.63
Hook_1498	8.91	16.84	23.19	30.31
G3_circuit	13.61	21.21	30.11	38.93
3D_Poisson	16.85	24.36	26.48	27.76
kkt_power	6.32	11.85	16.73	21.58
SPE10	11.96	22.50	29.89	35.28
memchip	10.63	18.55	26.30	32.41

Table 5.3: SPMV algorithm speedup for HEC format

By the comparison of Tables 5.3 and 5.5, a quarter of the running time has a significant gap. The white cells indicate that the speedup of the HYB format is approximately equivalent to that of the HEC format. The gray cells denote the speedup of HYB format which is less than that of the HEC format. For the matrix af_shell8, the speedups on three GPUs and four GPUs are 12.69 and 10.54 with the HYB format. By comparison, the speedups on three GPUs and four GPUs are 26.31 and 34.15 with the HEC format. For the matrix $thermal2$, the speedups on two GPUs, three GPUs and four GPUs are 12.54, 13.70 and 11.18 with the HYB format. By comparison, the speedups on two GPUs, three GPUs and four GPUs are 16.64, 23.57 and 31.44 with the HEC format. For the matrix $Hook\_1498$, the speedups on two GPUs, three GPUs and four GPUs are 13.88, 17.20 and 15.58

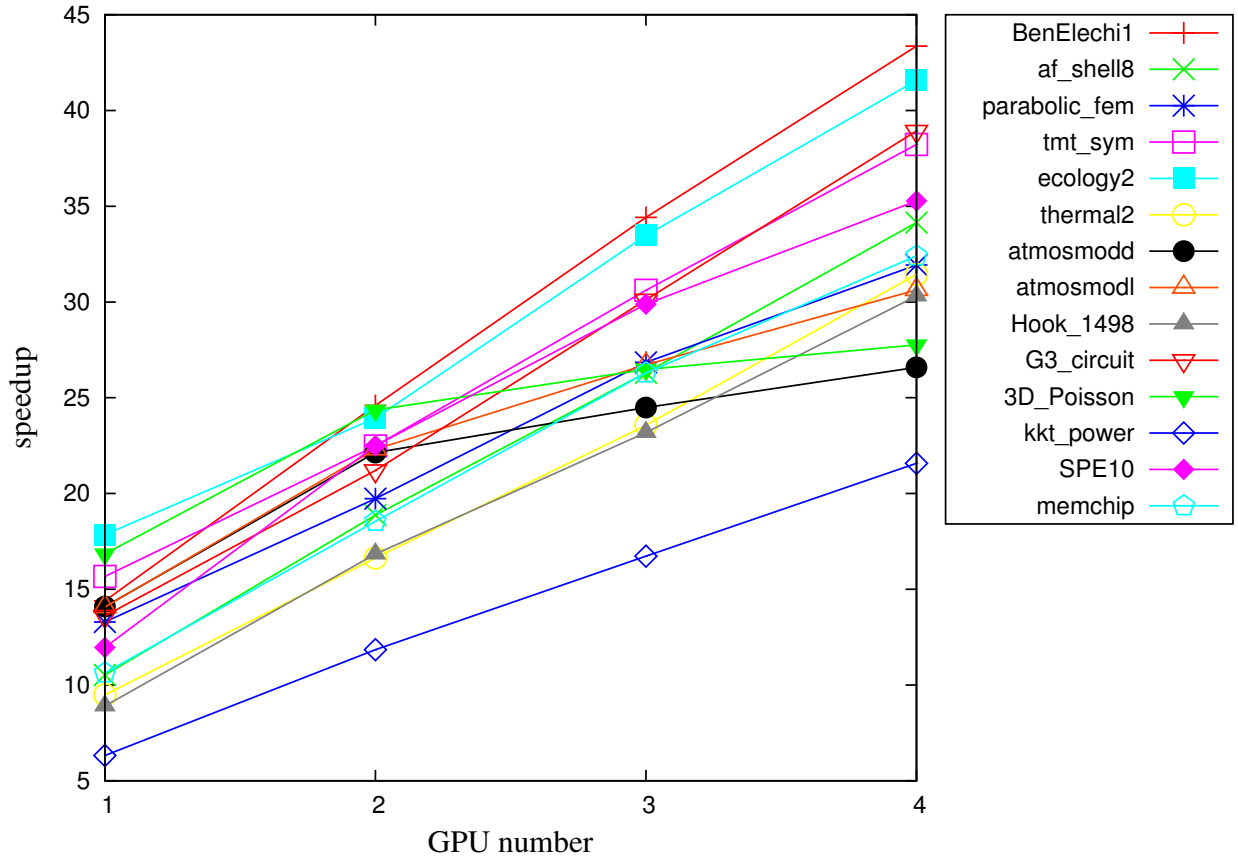


Figure 5.1: SPMV algorithm speedup vs. GPU number. for HEC format

with the HYB format. By comparison, the speedups on two GPUs, three GPUs and four GPUs are 16.84, 23.19 and 30.31 with the HEC format. For the matrix *kkt_power*, the speedups on three GPUs and four GPUs are 10.20 and 9.97 with the HYB format. By comparison, the speedups on three GPUs and four GPUs are 16.73 and 21.58 with the HEC format. For the matrix *SPE*10, the speedups on three GPUs and four GPUs are 22.45 and 27.00 with the HYB format. By comparison, the speedups on three GPUs and four GPUs are 29.89 and 35.28 with the HEC format. For the matrix *memchip*, the speedups on two GPUs and four GPUs are 15.38 and 20.53 with the HYB format. By comparison, the speedups on two GPUs and four GPUs are 18.55 and 32.41 with the HEC format. On the whole, the HEC matrix format shows a more friendly effect of speedup about the SPMV algorithm.

From Figure 5.2, a linear relationship does not exist between the speedup and the GPU number

for some matrices, such as af_shell8, $thermal2$, $atmosmodd$, $Hook\_1498$, $3D\_Poisson$, kkt_power, $memchip$ and $SPE10$, compared to the HEC format which has only two matrices $atmosmodd$ and $3D\_Poisson$ with an apparent nonlinear relation. This demostrates that the HEC format has more stable scalability to the SPMV algorithm than the HYB format.

We have given a detailed comparison of the performance between the HYB format and the HEC format. In both the aspects of speedup and scalability, the HEC format has a favorable performance on the whole.

Matrix	**CPU time (second)**	**1 GPU time (second)**	**2 GPUs time (second)**	**3 GPUs time (second)**	**4 GPUs time (second)**
BenElechi1	0.01213	0.00084	0.00049	0.00035	0.00028
af_shell8	0.01754	0.00183	0.00097	0.00138	0.00166
parabolic_fem	0.00635	0.00048	0.00032	0.00023	0.00019
tmt_sym	0.00824	0.00053	0.00036	0.00026	0.00021
ecology2	0.00877	0.00050	0.00036	0.00026	0.00020
thermal2	0.01930	0.00227	0.00154	0.00141	0.00173
atmosmodd	0.01767	0.00125	0.00079	0.00076	0.00065
atmosmodl	0.02136	0.00152	0.00094	0.00078	0.00072
Hook_1498	0.05723	0.00670	0.00412	0.00333	0.00367
G3_circuit	0.01342	0.00099	0.00062	0.00044	0.00034
3D_Poisson	0.02630	0.00156	0.00107	0.00100	0.00093
kkt_power	0.02315	0.00402	0.00228	0.00227	0.00232
SPE10	0.05686	0.00473	0.00259	0.00253	0.00211
memchip	0.03342	0.00333	0.00217	0.00127	0.00163

Table 5.4: SPMV algorithm running time for HYB format

Matrix	**1 GPU** (speedup)	**2 GPUs** (speedup)	**3 GPUs** (speedup)	**4 GPUs** (speedup)
BenElechi1	14.40	24.55	34.82	43.89
af_shell8	9.60	18.08	12.69	10.54
parabolic_fem	13.15	20.09	27.87	33.38
tmt_sym	15.63	22.75	31.35	39.70
ecology2	17.57	24.48	34.40	43.42
thermal2	8.51	12.54	13.70	11.18
atmosmodd	14.10	22.44	23.29	27.05
atmosmodl	14.06	22.74	27.28	29.81
Hook_1498	8.54	13.88	17.20	15.58
G3_circuit	13.61	21.54	30.63	39.89
3D_Poisson	16.82	24.65	26.41	28.25
kkt_power	5.76	10.17	10.20	9.97
SPE10	12.02	21.94	22.45	27.00
memchip	10.05	15.38	26.25	20.53

Table 5.5: SPMV algorithm speedup for HYB format

5.2 Nested RAS

The performance of the nested RAS framework can be adjusted by four parameters which are Outer RAS, Inner RAS, Outer overlap and Inner overlap. In order to obtain the effect affected by these parameters, we use ILU(0) as a fixed triangular factorization algorithm. The linear solver is GMRES(20). We select the restarted number as 20 because it is commonly used. In addition to the nested framework on multiple GPUs, we also implement the same nested RAS framework designed for a CPU, which will run in a single thread to serve as a reference for calculating the speedup of the nested RAS algorithm on multiple GPUs. Four experiments are presented. In these

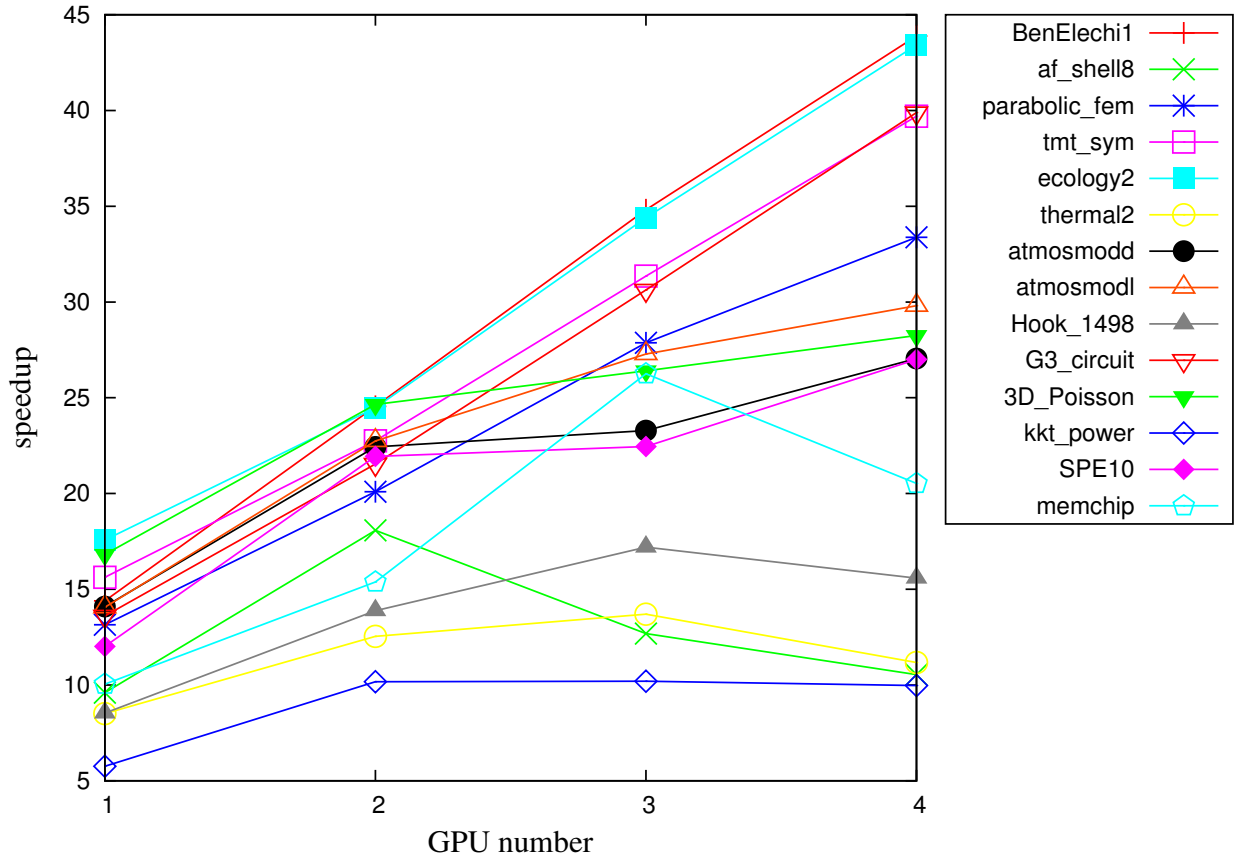


Figure 5.2: SPMV algorithm speedup vs. GPU number. for HYB format

experiments, different parameter combinations are configured to test the speedup which shows the nested RAS parallel performance.

Experiment 1 : The matrix used in this experiment comes from discretization of a three-dimensional Poisson equation. The grid is $120 \times 120 \times 120$ and its dimension is 1,728,000. It has 12,009,600 nonzero elements.

From Table 5.6, there are three data sections. They are responsible for variations of the outer RAS, the inner RAS and the overlap (the outer overlap and the inner overlap) parameters. In the first section, the yellow parts denote an increasing of the outer RAS when the other parameters are fixed. The speedup is 11.71 when the outer RAS is one (only one GPU is employed). With setting a higher outer RAS, the nested RAS algorithm displays an improvement of parallelism gradually. The speedup is improved to 18.90, 20.95 and 26.45 when the outer RAS is 2, 3 and 4,

Outer RAS	Inner RAS	Outer overlap	Inner overlap	ILU(k) level k	CPU time (second)	GPU time (second)	Speedup	Iteration
1	16	1	1	0	19.2223	1.6412	11.71	5
2	16	1	1	0	18.8530	0.9976	18.90	5
3	16	1	1	0	16.7350	0.7980	20.95	5
4	16	1	1	0	18.7378	0.7087	26.45	5
4	32	1	1	0	18.1899	0.6975	26.05	5
4	128	1	1	0	21.7188	0.7913	27.53	6
4	512	1	1	0	23.5084	0.8248	28.55	6
4	2048	1	1	0	24.2982	0.8589	28.29	6
4	16	2	1	0	17.4788	0.7663	22.80	5
4	16	1	2	0	17.4902	0.7387	23.67	5
4	16	2	2	0	19.3838	0.8166	23.74	5

Table 5.6: Nested RAS performance for 3D Poisson equation

respectively (when two GPUs, three GPUs and four GPUs are employed, respectively). As we fix the outer RAS at four and increase the inner RAS as denoted by blue parts, the speedup can be increased to 26.05, 27.53, 28.55 and 28.29 gradually when the inner RAS is set as 32, 128, 512 and 2048 separately in data section two. As analyzed in the last chapter, high RAS values cause more nonzero elements discarded outside of the square blocks. This leads to low calculation accuracy, which needs more iterations to balance out. Therefore, the iteration improves in five or six with the outer RAS and inner RAS increasing. Because the overlaps are fixed at one in the first two data sections, the loss of calculation accuracy can only be compensated by improving the iteration times. It can also be compensated through setting a larger outer overlap or inner overlap parameter. In the last data section, we test the performance of the nested RAS according to different overlap parameters as the red parts indicate. When the outer overlap and the inner overlap are set to 2 and 1, the speedup reduces to 22.80 and the iteration number reduces to 5. When the outer overlap and

the inner overlap are set to 1 and 2, the speedup is 23.67 and the iteration number is 5. When the outer overlap and the inner overlap are set to 2 and 2, the speedup is 23.74 and the iteration number is 5. Because a higher overlap imports more elements into the square blocks to be calculated, the degree of parallelism about the square blocks is decreased. This explains a decrease in speedup. However, more elements can improve the accuracy of calculation, which contributes to a decrease in the iteration number.

The experiment gives a reasonable effect of outer RAS, inner RAS and overlap parameters. It shows that the nested RAS framework is effective and over 20 times speedup can be achieved when running on our four GPUs workstation against on a traditional CPU.

Experiment 2 : The matrix used in this experiment is the matrix *atmosmodd* which is taken from the University of Florida sparse matrix collection [54]. *atmosmodd* is derived from a computational fluid dynamics problem. Its dimension is 1,270,432 and has 8,814,880 nonzero elements.

Outer RAS	Inner RAS	Outer overlap	Inner overlap	ILU(k) level k	CPU time (second)	GPU time (second)	Speedup	Iteration
1	16	1	1	0	9.9472	1.0148	9.80	4
2	16	1	1	0	10.0678	0.6039	16.67	4
3	16	1	1	0	9.3382	0.5102	18.30	4
4	16	1	1	0	12.6885	0.5094	24.91	5
4	32	1	1	0	12.6788	0.5007	25.32	5
4	128	1	1	0	13.8076	0.5144	26.87	5
4	512	1	1	0	13.9041	0.5017	27.72	5
4	2048	1	1	0	15.2217	0.5426	28.15	5
4	16	2	1	0	12.0949	0.5727	21.16	5
4	16	1	2	0	13.4669	0.5753	23.44	5
4	16	2	2	0	13.5958	0.6027	22.64	5

Table 5.7: Nested RAS performance for *atmosmodd*

The results are displayed in Table 5.7. In the first data section, the yellow parts show the effect of increasing the outer RAS as other parameters are fixed. The speedup reaches 9.80, 16.67, 18.30 and 24.91 when the outer RAS is set to 1, 2, 3 and 4, respectively. In the second data section, if the outer RAS is fixed at 4 and the inner RAS is increased as the blue parts indicate, the speedup can reach 25.32, 26.87, 27.72 and 28.15 when the inner RAS is set to 32, 128, 512 and 2048, respectively. The iteration improves from 4 to 5 as the outer RAS and inner RAS increase. The improvement of speedup with the outer RAS and the inner RAS increasing demonstrates the parallel performance is improved on the nested RAS framework. Just like the results of experiment one, the iteration increases from 4 to 5 to compensate the loss of calculation accuracy. The third data section shows the influence of overlap parameters on the nested RAS framework, which is presented by the red parts. Higher overlap parameters reduce the speedup. When the outer overlap is 2 and the inner overlap is 1, the speedup reduces to 21.16. When the outer overlap is 1 and the inner overlap is 2, the speedup reduces to 23.44. When the outer overlap is 2 and the inner overlap is 2, the speedup is 22.64. Because the speedup is influenced not only by the factors of these parameters but also by the factor of matrix pattern, the data of speedup only shows a general tendency. The iteration does not reduce back to 4 from 5 with high overlap, which is also influenced by the matrix pattern. However, the nested RAS can still keep the speedup over 20 when outer overlap or inner overlap is set to 2. This experiment also demonstrates that the nested RAS framework can provide 20 times speedup or higher as four GPUs are employed.

Experiment 3 : The matrix used in this experiment is the matrix *atmosmodl* which is taken from the University of Florida sparse matrix collection [54]. *atmosmodl* is derived from a computational fluid dynamics problem. Its dimension is 1,489,752 and has 10,319,760 nonzero elements.

Outer RAS	Inner RAS	Outer overlap	Inner overlap	ILU(k) level k	CPU time (second)	GPU time (second)	Speedup	Iteration
1	16	1	1	0	6.2765	0.5929	10.59	2
2	16	1	1	0	6.3254	0.3576	17.69	2
3	16	1	1	0	5.6723	0.3058	18.56	2
4	16	1	1	0	5.9346	0.2715	21.85	2
4	32	1	1	0	6.2587	0.2749	22.77	2
4	128	1	1	0	5.9306	0.2592	22.86	2
4	512	1	1	0	6.8067	0.2608	26.13	2
4	2048	1	1	0	7.0531	0.2701	26.14	2
4	16	2	1	0	5.8703	0.2830	20.73	2
4	16	1	2	0	6.2210	0.2860	21.77	2
4	16	2	2	0	6.0569	0.2928	20.69	2

Table 5.8: Nested RAS performance for *atmosmodl*

The nested RAS results of the matrix *atmosmodl* are shown in Table 5.8. This matrix can be solved quickly on the nested RAS framework and only two iteration numbers are needed. For different parameter configurations, the iteration remains at 2 because the influence of these parameters is not serious enough to change the iteration numbers. In the first data section, the yellow parts give changes of speedup and iteration with different outer RAS parameters. The speedup can be accelerated to 10.59, 17.69, 18.56 and 21.85 when the outer RAS is set as 1, 2, 3 and 4, respectively. This shows that higher parallel performance is obtained on more GPUs. In the second data section, the blue parts demonstrate the increase of speedup as the inner RAS increases. When the inner RAS is configured as 32, 128, 512 and 2048, the speedup reaches 22.77, 22.86, 26.13 and 26.14, respectively. The parallel performance is further improved through improving the inner RAS. Higher overlap parameters are set in the third data section and the results are displayed in the red parts. The speedup reduces to 20.73 when the outer overlap is 2 and the inner overlap is 1.

The speedup reduces to 21.77 when the outer overlap is 1 and the inner overlap is 2. The speedup is 20.69 when the outer overlap is 2 and the inner overlap is 2. The speedups in the third data section are apparently less than those in the second data section. That is because of lower degree of parallelism caused by higher overlaps. Based on the analysis above, this experiment also shows that 20 times speedup or higher can be provided by the nested RAS framework on the four GPUs workstation.

Experiment 4 : $SPE10$ is a classical reservoir model in porus media and has been widely applied for benchmark tests by the oil and gas industry [2]. The problem is highly heterogeneous and is difficult to solve. The grid size for SPE10 is $60 \times 220 \times 85$. The number of unknowns is 2,188,851 and the number of nonzero elements is 29,915,573. This experiment uses one of the SPE 10 matrices generated at a time step.

Outer RAS	Inner RAS	Outer overlap	Inner overlap	ILU(k) level k	CPU time (second)	GPU time (second)	Speedup	Iteration
1	16	1	1	0	121.8090	13.5696	8.98	24
2	16	1	1	0	117.3070	8.0304	14.61	24
3	16	1	1	0	109.2337	5.6871	19.21	22
4	16	1	1	0	124.4517	5.0351	24.71	24
4	32	1	1	0	140.9913	5.4257	25.98	26
4	128	1	1	0	187.4940	7.7556	24.18	36
4	512	1	1	0	173.4530	6.4400	26.93	29
4	2048	1	1	0	234.5983	8.9732	26.15	37
4	16	2	1	0	214.3207	8.9957	23.83	42
4	16	1	2	0	147.2643	5.6850	25.90	26
4	16	2	2	0	173.0123	8.0894	21.39	33

Table 5.9: Nested RAS performance for $SPE10$

$SPE10$ is hard to solve and much time is needed as illustrated in Table 5.9. The iterations of

all runs are equal to or greater than 22. Though it is complicated and costs more time to reach convergence compared to the previous three matrices, its results have a similar data tendency like them. The yellow parts of the first data section show the increase of speedup as the outer RAS goes up. When the outer RAS is configured as 1, 2, 3 and 4, respectively, the corresponding speedup is 8.98, 14.61, 19.21 and 24.71. The blue parts of the second data section display the change of speedup with the inner RAS. If the inner RAS is set as 32, 128, 512, and 2048, the speedup will be 25.98, 24.18, 26.93 and 26.15. We find that the speedup does not strictly increase as the inner RAS goes up. That is because the speedup is also affected by the pattern of the $SPE10$. However, the speedups in the blue parts are greater than those in the yellow parts as a whole. The iteration from the first data section to the second data section also displays a general growing tendency. The iteration number is 24, 24, 22, and 24 when the outer RAS is set as 1, 2, 3 and 4, respectively. When the outer RAS is fixed at 4 and the inner RAS is set as 32, 128, 512 and 2048, the iteration goes up to 26, 36, 29 and 37, respectively. Therefore, the general data tendency is reasonable and the nested RAS still contributes to the improvement of parallel performance. In the last data section, the speedup goes down as higher overlap parameters are configured compared to the second data section. When the outer RAS is 2 and the inner RAS is 1, the speedup is 23.83. When the outer RAS is 1 and the inner RAS is 2, the speedup is 25.90. When the outer RAS is 2 and the inner RAS is 2, the speedup is 21.39. The iteration values in the third data section also present a general decreasing tendency with respect to the values in the second data section because extra elements are introduced by high overlaps. Overall, the speedup in this experiment is still kept over 20 times when four GPUs are employed.

5.3 GMRES with ILU(k)

In this section, we focus on testing the performance of the algorithm of GMRES(20) with the preconditioner ILU(k) on multiple GPUs. We use the same testing matrices and the nested RAS parameters as in Section 5.2. For any combination of the nested RAS parameters, the parameter

k is configured as 0, 1, 2 and 3, respectively. The influence on the speedup and the iteration of different parameter k can be tested and analyzed. We also implement the same ILU(k) algorithm for a CPU, which plays as a contrastive algorithm used with the algorithm for GPUs to determine the speedup. The experiment results for matrices $3D\_Poisson$, $atmosmodd$, $atmosmodl$ and $SPE10$ are listed in Tables 5.10, 5.11, 5.12 and 5.13, respectively. In each table, the data are divided into many data sections. Each data section is responsible for a concrete RAS parameter combination. When k is zero, namely, the preconditioner is ILU(0), the gray color is set for the cells. The cells in other colors are used for higher k levels which is greater than zero.

A higher k introduces more fill-in elements and the calculation accuracy for solving preconditioner systems is optimized. However, an apparent negative effect also takes effect at the same time, which is that more elements decrease the degree of parallelism for a precondtioner matrix. In addition, the algorithm performance is also related to the pattern of a matrix. Therefore, the running result of an algorithm with a certain k needs comprehensive consideration under all these impact factors.

Experiment 1: The matrix used in this experiment is from a 3D Poisson equation. Its dimension is 1,728,000 and it has 12,009,600 nonzero elements. The average number of nonzero elements in each row is seven. The results are displayed in Table 5.10.

Seq No.	Outer RAS	Inner RAS	Outer overlap	Inner overlap	ILU(k) level k	CPU time (second)	GPU time (second)	Speedup	Iteration
1	1	16	1	1	0	19.2223	1.6412	11.71	5
					1	11.3803	1.2312	9.24	3
					2	16.8692	2.7678	6.10	4
					3	12.9443	2.3717	5.46	3
2	2	16	1	1	0	18.8530	0.9976	18.90	5
					1	11.7629	0.8832	13.32	3

Continued on next page

					2	16.9354	2.1309	7.95	4
					3	13.6234	2.0693	6.58	3
3	3	16	1	1	0	16.7350	0.7980	20.95	5
					1	13.3249	0.9539	13.97	4
					2	16.0166	1.8743	8.55	4
					3	13.6167	1.8888	7.21	3
4	4	16	1	1	0	18.7378	0.7087	26.45	5
					1	15.9671	0.8437	18.95	4
					2	14.9939	1.5560	9.63	4
					3	13.2788	1.5195	8.74	3
5	4	32	1	1	0	18.1899	0.6975	26.05	5
					1	15.0185	0.7989	18.78	4
					2	20.7133	1.7771	11.66	5
					3	13.7563	1.4681	9.37	3
6	4	128	1	1	0	21.7188	0.7913	27.53	6
					1	16.1373	0.7122	22.62	4
					2	21.2229	1.4322	14.81	5
					3	18.0938	1.4807	12.22	4
7	4	512	1	1	0	23.5084	0.8248	28.55	6
					1	16.6320	0.7235	22.99	4
					2	22.5626	1.3106	17.22	5
					3	19.6067	1.3298	14.74	4
8	4	2048	1	1	0	24.2982	0.8589	28.29	6
					1	17.9229	0.7493	23.90	4
					2	23.5394	1.3480	17.46	5

Continued on next page

					3	21.2095	1.2877	16.48	4
9	4	16	2	1	0	17.4788	0.7663	22.80	5
					1	14.6865	0.9659	15.21	4
					2	18.7666	2.1537	8.71	5
					3	13.1767	1.7508	7.53	3
10	4	16	1	2	0	17.4902	0.7387	23.67	5
					1	11.7230	0.7200	16.27	3
					2	16.2099	1.7118	9.47	4
					3	13.2869	1.7386	7.64	3
11	4	16	2	2	0	19.3838	0.8166	23.74	5
					1	15.6976	1.0011	15.68	4
					2	20.5109	2.2900	8.95	5
					3	13.4053	1.9399	6.91	3

Table 5.10: ILU(k) performance for 3D Poisson equation

In Table 5.10, each data section corresponds to a concrete RAS parameter combination. All the data sections have a similar data tendency and can be analyzed using the same way. We take the first data section as an analysis example. The RAS parameter combination in the first section is configured as 1, 16, 1, 1 for the outer RAS, the inner RAS, the outer overlap and the inner overlap, respectively.

With an increase in k, the CPU time changes. When k is 0, the CPU time is 19.2223. When k is 1, the CPU time is 11.3803. When k is 2, the CPU time is 16.8692. When k is 3, the CPU time is 12.9443. Obviously, when k is greater than zero, the CPU time decreases compared to $k = 0$. As a high k increases the accuracy of calculations and saves the CPU time, the data results

are reasonable. However, the CPU time does not show a strict downtrend as k increases. That is because the matrix pattern plays a part. On the whole, by applying k which is greater than zero, the algorithm on a CPU is accelerated.

Next, the GPU time also changes with different k. When k is 0, the GPU time is 1.6412. When k is 1, the GPU time is 1.2312. When k is 2, the GPU time is 2.7678. When k is 3, the GPU time is 2.3717. The degree of parallelism affects the GPU time. More fill-in elements decrease the parallel performance. The negative influence becomes more apparent when a high k is configured. The acceleration effect from the increase of calculation accuracy will be balanced out or even the negative influence accounts for a major role. Therefore, the positive factor dominates at first, the GPU time of $k = 1$ is less than the GPU time of $k = 0$. Then the negative factor dominates and the GPU time of $k = 2$ is greater than the GPU time of $k = 1$. Eventually, the positive effect comes back when k is 3 and the GPU time of $k = 3$ is less than the GPU time of $k = 2$.

The speedup goes down as k goes up. When k is 0, the speedup is 11.71. When k is 1, the speedup is 9.24. When k is 2, the speedup is 6.10. When k is 3, the speedup is 5.46. Although the algorithm on multiple GPUs obtains a favorable acceleration of speedups greater than five, the parallel performance is heavily affected and the speedup goes down as k increases. That is because the extra fill-in elements limit the parallel ability. The GPU time goes up. However, this has little effect on a single thread CPU platform. On the contrary, the CPU time goes down as the calculation accuracy increases. In all together, the speedup goes down.

The purpose of ILU(k) is to improve the accuracy of calculations and to reach convergence at fewer iterations. The iteration data shows reasonable results. The iteration is 5 when k is 0. The iteration is 3 when k is 1. The iteration number is 4 when k is 2. The iteration is 3 when k is 3. Obviously, the iteration has a general downtrend. The iteration goes back to 4 when k is 2. This is because the iteration is also affected by the pattern of the matrix. The whole objective of using a high k to reduce iteration is realized. The variation trend of the iteration is corresponding to that of the CPU time. That is because they are both affected by the factors of calculation accuracy and

matrix pattern.

The data sections of other RAS parameter combinations can be analyzed like above. From another analysis angle, if the k value is fixed, the speedup and iteration can reflect the parallel ability of different RAS parameter combinations, as illustrated in Figures 5.3 and 5.4. The horizontal axes of these two figures are both the sequence number of combinations. The sequence numbers from 1 to 8 represent the outer RAS increase from 1 to 4 and then the inner RAS increases from 16 to 2048. For Figure 5.3, the speedup shows an apparent rising trend in the first eight points. The last three sequence number 9, 10 and 11 represent combinations with high overlaps. In addition, the speedup of them goes down since the degree of parallelism is affected negatively by the high overlaps. For Figure 5.4, the iteration has an upgoing trend when the sequence number goes from 1 to 8. After the sequence number is beyond 8, the iteration goes down.

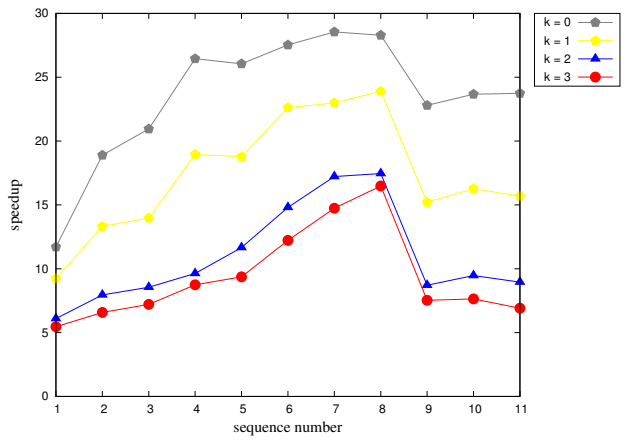


Figure 5.3: Speedup for matrix *3D_Poission*

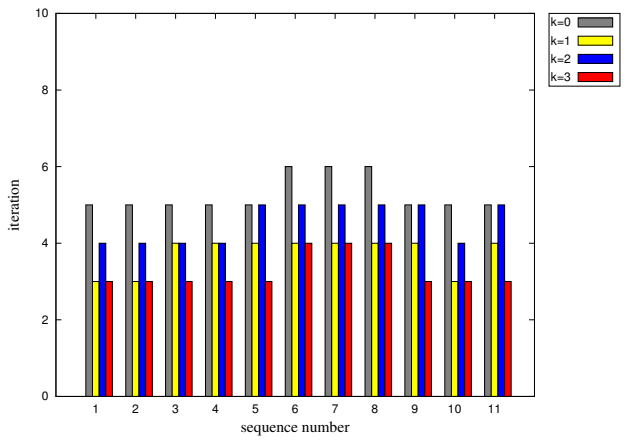


Figure 5.4: Iteration for matrix *3D_Poission*

Experiment 2: The matrix used in this experiment is *atmosmodd* which is downloaded from the University of Florida sparse matrix collection [54]. Its dimension is 1,270,432 and has 8,814,880 nonzero elements. The average number of nonzero elements in each row is seven. The results are listed in Table 5.11.

Seq No.	Outer RAS	Inner RAS	Outer overlap	Inner overlap	ILU(k) level k	CPU time (second)	GPU time (second)	Speedup	Iteration
1	1	16	1	1	0	9.9472	1.0148	9.80	4
					1	8.3525	0.9687	8.62	3
					2	10.9537	2.2354	4.90	4
					3	8.9240	2.0367	4.38	3
2	2	16	1	1	0	10.0678	0.6039	16.67	4
					1	8.6323	0.6525	13.22	3
					2	11.3346	1.5006	7.55	4
					3	9.0390	1.4414	6.27	3
3	3	16	1	1	0	9.3382	0.5102	18.30	4
					1	8.0514	0.5824	13.83	3
					2	10.7574	1.3659	7.88	4
					3	8.7288	1.3294	6.57	3
4	4	16	1	1	0	12.6885	0.5094	24.91	5
					1	7.8831	0.5157	15.27	3
					2	11.6020	1.1862	9.78	4
					3	9.5057	1.2340	7.71	3
5	4	32	1	1	0	12.6788	0.5007	25.32	5
					1	8.5043	0.4750	17.90	3
					2	11.3103	1.1416	9.92	4
					3	9.2747	1.1008	8.43	3
6	4	128	1	1	0	13.8076	0.5144	26.87	5
					1	8.8368	0.4235	20.96	3
					2	11.6731	0.8925	13.09	4

Continued on next page

					3	9.5661	0.8709	10.99	3
7	4	512	1	1	0	13.9041	0.5017	27.72	5
					1	9.6113	0.4134	23.25	3
					2	13.2917	0.7863	16.91	4
					3	11.1087	0.7578	14.68	3
8	4	2048	1	1	0	15.2217	0.5426	28.15	5
					1	13.8028	0.5471	25.23	4
					2	17.9290	0.9906	18.12	5
					3	16.1325	0.9243	17.47	4
9	4	16	2	1	0	12.0949	0.5727	21.16	5
					1	8.0350	0.5681	14.17	3
					2	11.3430	1.3017	8.72	4
					3	9.1000	1.2555	7.25	3
10	4	16	1	2	0	13.4669	0.5753	23.44	5
					1	7.7173	0.5928	13.02	3
					2	12.1547	1.3952	8.72	4
					3	9.8107	1.4445	6.79	3
11	4	16	2	2	0	13.5958	0.6027	22.64	5
					1	8.5784	0.6367	13.47	3
					2	11.7119	1.4768	7.94	4
					3	9.6959	1.4918	6.50	3

Table 5.11: ILU(k) performance for *atmosmodd*

The data result of *atmosmodd* is listed in Table 5.11. Because all the data sections have a

similar data tendency, the approach for their data analysis is identical. We take the fifth data section as an sample of analysis. The outer RAS, the inner RAS, the ouer overlap and the inner overlap are 4, 32, 1 and 1, respectively.

If k is assigned with a high value, more fill-in elements will lead to high calculation accuracy. The CPU running time is saved. When k is 0, the CPU time is 12.6788. When k is 1, the CPU time is 8.5043. When k is 2, the CPU time is 11.3103. When k is 3, the CPU time is 9.2747. All the CPU time for k above zero is less than the CPU time at $k = 0$. The running time at $k = 2$ is even greater than that at $k = 3$. This is caused by the matrix pattern.

The GPU time shows the effect of different k on the multiple GPU platform. When k is 0, the GPU time is 0.5007. When k is 1, the GPU time is 0.4750. When k is 2, the GPU time is 1.1416. When k is 3, the GPU time is 1.1008. There are two opposite factors which affect the GPU time. Since a high k imports more fill-in elements, a positive effect is an increase in calculation accuracy but a negative effect is a decrease in the degree of parallelism. Therefore, the overall effect is determined by which one dominates. At k equal to 1, the positive factor dominates and the GPU time decreases. However, the negative factor prevails at k equal to 2 or 3. Thus the GPU time goes up for these two cases.

All the speedups are over eight and favorable. When k is 0, the speedup is 25.32. When k is 1, the speedup is 17.90. When k is 2, the speedup is 9.92. When k is 3, the speedup is 8.43. Obviously, a high k does not help improve the speedup. That is because the parallel performance of the algorithm decreases with an increase in k on multiple GPUs. But the algorithm on a CPU with a single thread has no relation to the parallelization. The CPU time even decreases due to the improvement of calculation accuracy. Thus the speedup goes down.

The iteration number can be reduced by using a high k. When k is 0, the iteration is 5. When k is 1, the iteration is 3. When k is 2, the iteration is 4. When k is 3, the iteration is 3. Because the matrix pattern takes effect, the iteration does not decrease continuously as k goes up. The variation trend of iteration is just like that of the CPU time.

We show only one sample explanation above. Other data sections can be analyzed similarly. Next, the parallel performance for different RAS parameter combinations will be studied at a fixed k. Figure 5.5 shows the speedup vs. the sequence number and Figure 5.6 shows the iteration vs. the sequence number. The parallel ability of the nested RAS framework enhances gradually as the sequence number goes up from 1 to 8. The last three sequence numbers which are 9, 10 and 11 represent the combinations of high overlap parameters. First, Figure 5.5 validates the reasonable rising tendency of speedup for the first 8 points of each curve. The last three points on each curve show that a high overlap reduces the speedup. Figure 5.6 displays the iteration changes. The iteration reaches the maximum when the sequence number is 8. As we know, the sequence number 8 represents the outer RAS at 4 and the inner RAS at 2048, which has the highest parallel configuration in the experiment. Therefore, the effect of the iteration is appropriate. The iteration numbers go down when a high overlap is used, which is shown as the sequence number is 9, 10 or 11. One thing to note is the iteration remains unchanged for $k = 0$ even if the overlap is high. That is because the impact of the high overlap is not sufficient to decrease the iteration when k is fixed at 0.

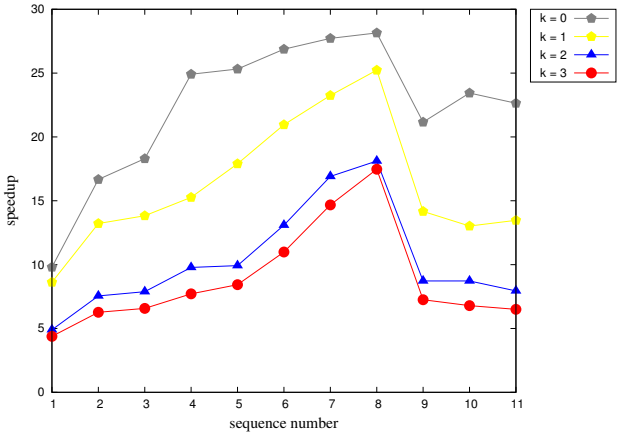


Figure 5.5: Speedup for matrix *atmosmodd*

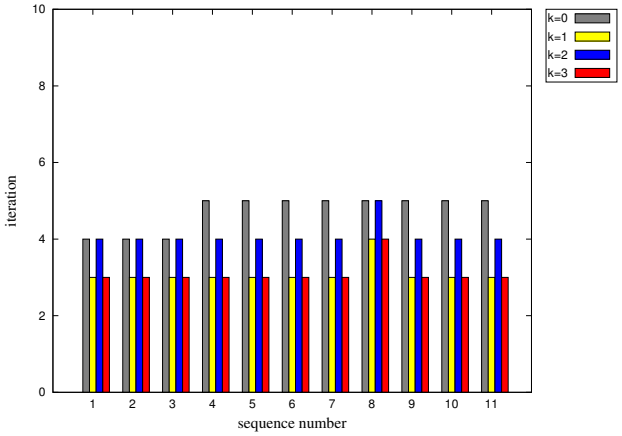


Figure 5.6: Iteration for matrix *atmosmodd*

Experiment 3: The matrix used in this experiment is the matrix *atmosmodl* which is downloaded from the University of Florida sparse matrix collection [54]. Its dimension is 1,489,752

and it has 10,319,760 nonzero elements. The average number of nonzero elements in each row is seven. The results are listed in Table 5.12.

Seq No.	Outer RAS	Inner RAS	Outer overlap	Inner overlap	ILU(k) level k	CPU time (second)	GPU time (second)	Speedup	Iteration
1	1	16	1	1	0	6.2765	0.5929	10.59	2
					1	3.3045	0.3654	9.04	1
					2	6.6461	1.1165	5.95	2
					3	3.4993	0.6921	5.06	1
2	2	16	1	1	0	6.3254	0.3576	17.69	2
					1	6.5085	0.5067	12.85	2
					2	6.9288	0.8684	7.98	2
					3	3.6075	0.5709	6.32	1
3	3	16	1	1	0	5.6723	0.3058	18.56	2
					1	6.5227	0.4306	15.15	2
					2	6.3221	0.7311	8.65	2
					3	3.5934	0.4951	7.26	1
4	4	16	1	1	0	5.9346	0.2715	21.86	2
					1	6.5157	0.4005	16.27	2
					2	6.7846	0.6862	9.89	2
					3	3.8291	0.4722	8.11	1
5	4	32	1	1	0	6.2587	0.2749	22.77	2
					1	6.7763	0.3742	18.11	2
					2	6.8376	0.5997	11.40	2
					3	7.5432	0.7603	9.92	2
6	4	128	1	1	0	5.9306	0.2592	22.86	2

Continued on next page

					1	7.1376	0.3509	20.34	2
					2	7.1071	0.5434	13.08	2
					3	7.7270	0.6581	11.74	2
7	4	512	1	1	0	6.8067	0.2608	26.13	2
					1	6.6383	0.3220	20.62	2
					2	7.7505	0.4710	16.46	2
					3	8.2833	0.5699	14.54	2
8	4	2048	1	1	0	7.0531	0.2701	26.14	2
					1	8.0811	0.3475	23.26	2
					2	8.0445	0.4831	16.65	2
					3	9.2654	0.5671	16.34	2
9	4	16	2	1	0	5.8703	0.2830	20.73	2
					1	5.9753	0.4149	14.40	2
					2	6.4990	0.6857	9.48	2
					3	3.6067	0.4916	7.34	1
10	4	16	1	2	0	6.2210	0.2860	21.77	2
					1	6.8137	0.4393	15.51	2
					2	6.6448	0.7268	9.14	2
					3	3.6022	0.4733	7.62	1
11	4	16	2	2	0	6.0569	0.2928	20.69	2
					1	6.5487	0.4290	15.28	2
					2	6.6413	0.7417	8.95	2
					3	3.6763	0.4951	7.42	1

Table 5.12: ILU(k) performance for *atmosmodl*

The results of experiment 3 are listed in Table 5.12. Because all the data sections have a similar data tendency and can be analyzed in the same way, we only take one of them as an example analysis. We select the data section 10 as an example, which has a RAS parameter combination of 4, 16, 1, and 2 for the outer RAS, the inner RAS, the outer overlap and the inner overlap, respectively.

The CPU time changes with different k. When k is 0, the CPU time is 6.2210. When k is 1, the CPU time is 6.8137. When k is 2, the CPU time is 6.6448. When k is 3, the CPU time is 3.6022. As we mentioned above, a high k introduces fill-in elements, which leads to the improvement of accuracy for solving a preconditioner system. The CPU time can be saved. However, the CPU time only decreases when k is 3. It increases a little bit when k is 1 or 2. This is a different data tendency compared to the experiment one and the experiment two. Because the workstation and configuration of parameters are kept unchanged, this result is caused by the pattern of the matrix.

The GPU time is also affected heavily by k. When k is 0, the GPU time is 0.2860. When k is 1, the GPU time is 0.4393. When k is 2, the GPU time is 0.7268. When k is 3, the GPU time is 0.4733. Obviously, when k is greater than zero, all the running times go up. Although a high k can improve the accuracy of calculation and save GPU running time, fill-in elements decrease the parallelism. In this experiment, the negative effect dominates and all the GPU times increase.

The speedups are all over seven but decrease quickly as k goes up. When k is 0, the speedup is 21.77. When k is 1, the speedup is 15.51. When k is 2, the speedup is 9.14. When k is 3, the speedup is 7.62. As applying a high k, the CPU time remains almost unchanged or goes down. But the GPU time goes up. Thus the speedup appears a fast downtrend.

Next, we analyze the iteration change with the variation of k. When k is 0, the iteration is 2. When k is 1, the iteration is 2. When k is 2, the iteration is 2. When k is 3, the iteration is 1. If a high k can reduce the CPU time significantly, it may improve the accuracy of calculation sharply. This effect can also be reflected from the iteration number. We can see that the iteration goes down

to 1 when k is 3. But for other situations, the iteration remains 2. That is because the calculation accuracy cannot be optimized for this kind of matrix pattern.

The other data sections can be analyzed similarly. Next, we explain the data from another aspect. Figure 5.7 gives the curves of the speedup vs. the sequence number, and Figure 5.8 gives the histogram of the iteration vs. the sequence number. The sequence number from 1 to 8 represents the RAS combinations with the parallel ability increasing gradually. Each curve in Figure 5.7 shows a general up trend in the first eight points. Figure 5.8 also shows the iteration reaches the maximum for each k value when the sequence number is eight. The sequence numbers of 9, 10 and 11 represent three RAS parameter combinations of high overlap. High overlap imports more elements. The accuracy of calculation is compensated but the degree of parallelism is reduced. Therefore, there is a low speedup for these three points in Figure 5.7. There is a small iteration number for $k = 3$ in Figure 5.8.

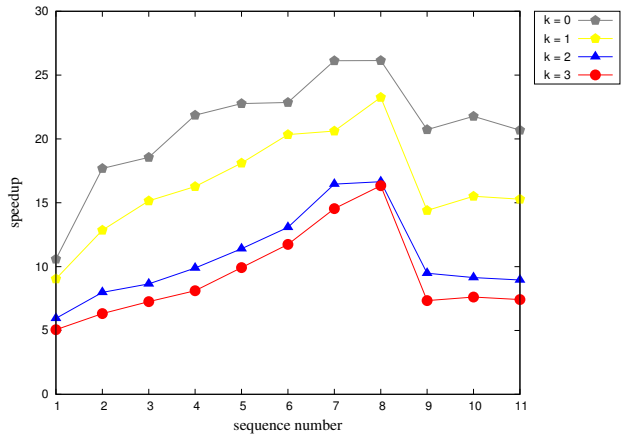


Figure 5.7: Speedup for matrix *atmosmodl*

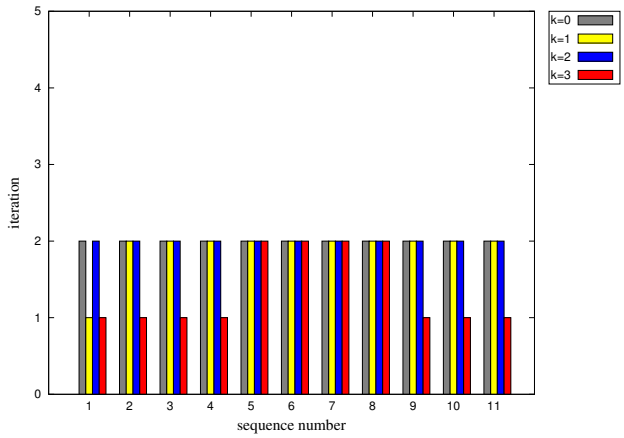


Figure 5.8: Iteration for matrix *atmosmodl*

Experiment 4: The matrix used in this experiment is the matrix $SPE10$. Its result is listed in Table 5.13. It comes from a classical reservoir model which is widely used for benchmark tests in the petroleum industry. Its grid size is $60 \times 220 \times 85$. The dimension is 2,188,851 and the number of nonzero elements is 29,915,573. The average number of nonzero elements in each row is fourteen.

Seq No.	Outer RAS	Inner RAS	Outer overlap	Inner overlap	ILU(k) level k	CPU time (second)	GPU time (second)	Speedup	Iteration
1	1	16	1	1	0	121.8090	13.5696	8.98	24
					1	78.8841	13.3447	5.91	13
					2	146.9233	38.2944	3.84	24
					3	out of memory			
2	2	16	1	1	0	117.3070	8.0304	14.61	24
					1	89.5844	9.8897	9.06	14
					2	128.7977	26.8328	4.80	21
					3	114.2540	40.2732	2.84	13
3	3	16	1	1	0	109.2337	5.6871	19.21	22
					1	77.2547	6.1570	12.55	12
					2	110.7097	15.3333	7.22	18
					3	107.9553	28.4604	3.79	12
4	4	16	1	1	0	124.4517	5.0351	24.71	24
					1	90.3743	6.7871	13.32	14
					2	149.8153	19.7128	7.60	22
					3	129.6670	33.6608	3.85	14
5	4	32	1	1	0	140.9913	5.4257	25.98	26
					1	118.9020	7.6273	15.59	18
					2	166.5340	17.9623	9.27	26
					3	168.7813	31.8236	5.30	18
6	4	128	1	1	0	187.4940	7.7556	24.18	36
					1	160.1940	9.8493	16.26	23
					2	206.2483	18.3740	11.23	31

Continued on next page

					3	166.3463	29.6867	5.60	18
7	4	512	1	1	0	173.4530	6.4400	26.93	29
					1	149.1907	9.1286	16.34	20
					2	169.7443	14.2519	11.91	23
					3	205.1323	26.0689	7.87	20
8	4	2048	1	1	0	234.5983	8.9732	26.15	37
					1	288.2267	18.0664	15.95	34
					2	353.4443	29.4679	11.99	44
					3	435.6997	55.5115	7.85	40
9	4	16	2	1	0	214.3207	8.9957	23.83	42
					1	103.7013	8.5702	12.10	16
					2	162.8040	21.4573	7.59	25
					3	132.8710	31.7873	4.18	15
10	4	16	1	2	0	147.2643	5.6850	25.90	26
					1	96.1632	7.9527	12.09	14
					2	151.6970	22.0360	6.88	22
					3	126.8273	35.7694	3.55	13
11	4	16	2	2	0	173.0123	8.0894	21.39	33
					1	132.8593	12.2156	10.88	19
					2	193.9310	30.1315	6.44	28
					3	128.3077	38.6992	3.32	13

Table 5.13: ILU(k) performance for *SPE*10

Table 5.13 gives the running results for *SPE*10. Each data section is configured by a fixed RAS

parameter combination. For the first data section where the outer RAS is one, namely, the number of GPU is one, the running for $k = 3$ indicates memory overflow. Because the matrix is huge, only two or more GPUs can solve it when k is configured at 3. The data results of other data sections are complete. They all have similar data tendency and can be analyzed in the same way. We take the second data section for instance. The second data section has a RAS parameter combination of 2, 16, 1 and 1 for the outer RAS, the inner RAS, the outer overlap and the inner overlap, respectively.

Obviously, the CPU time has a great change with different k values. When k is 0, the CPU time is 117.3070. When k is 1, the CPU time is 89.5844. When k is 2, the CPU time is 128.7977. When k is 3, the CPU time is 114.2540. The calculation accuracy of solving a preconditioner system can be improved by including extra fill-in elements. A high k provides such a role. Thus the CPU time decreases with k increasing. This is reasonable for $k = 1$ and $k = 3$. But the pattern matrix also takes a part in the calculation which may counteract the action of k, such as the running time of $k = 2$ which is even longer than that of $k = 0$.

The GPU time goes up as a high k is configured. When k is 0, the GPU time is 8.0304. When k is 1, the GPU time is 9.8897. When k is 2, the GPU time is 26.8328. When k is 3, the GPU time is 40.2732. The up trend caused by the degree of parallelism is heavily affected by the fill-in elements. The fill-in elements of a high k make a preconditioner matrix pattern much complex and the relation among rows much tight. Although the improvement of calculation accuracy can shorten calculation time, the influence of it is too weak. Thus the sharp fall of the degree of parallelism makes the GPU time go up sharply, too.

The speedup for different k is listed as follows. When k is 0, the speedup is 14.61. When k is 1, the speedup is 9.06. When k is 2, the speedup is 4.80. When k is 3, the speedup is 2.84. As a higher k is applied, the decrease of degree of parallelism makes the GPU time become larger but the accuracy of calculation makes the CPU time less. Therefore, the speedup goes down when k goes up.

The iteration number has the same variation tendency as the CPU time because they both are

affected by the same reason of calculation accuracy and the matrix pattern. When k is 0, the iteration is 24. When k is 1, the iteration is 14. When k is 2, the iteration is 21. When k is 3, the iteration is 13. On the whole, the iteration number with a high k is smaller than the iteration number at $k = 0$. The purpose of using a high k to reduce the iteration time is achieved. But the iteration fluctuates as k increases, which is caused by the concrete matrix pattern.

All the other data sections can be analyzed similarly. Next, we give another aspect of the data analysis. The curves of speedup vs. the sequence number for a fixed k are illustrated in Figure 5.9. The curves of iteration vs. the sequence number for a fixed k are illustrated in Figure 5.10. The parallel ability of the nested RAS framework goes up continuously as the sequence number goes from 1 to 8. Thus the speedup also goes up which can be seen in Figure 5.9. When the sequence number is 9, 10 or 11, the outer overlap or the inner overlap is configured highly. Moreover, extra elements are imported and the ability of parallelism decreases. The speedup of these last three points on each curve is low. Better parallelism leads to less accuracy which needs more iteration to compensate. As illustrated in Figure 5.10, the iteration goes up till the combination number reaches eight. After that, the iteration goes down due to the high overlap.

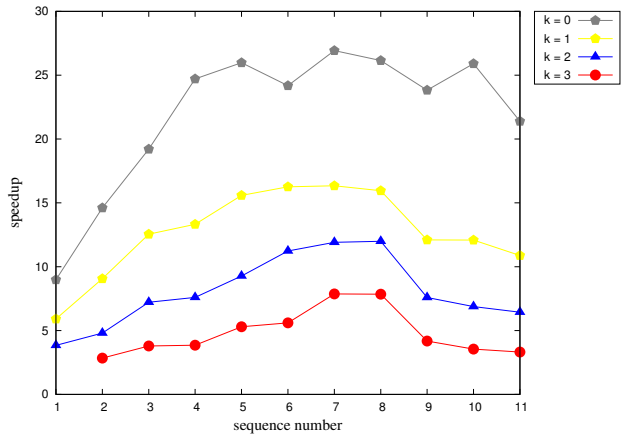


Figure 5.9: Speedup for matrix $SPE10$

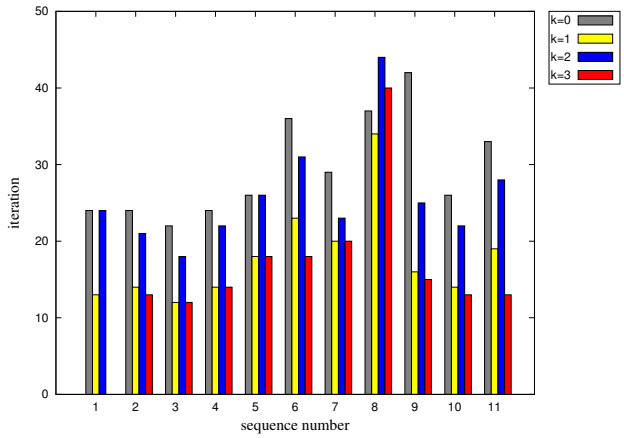


Figure 5.10: Iteration for matrix $SPE10$

Chapter 6

CONCLUSIONS

In this thesis, we have designed and developed a GMRES algorithm and an ILU(k) preconditioner for multiple-GPU architecture in a single node. The major parts of implementation are SPMV algorithm, nested RAS framework and ILU(k) preconditioner. We have designed a series of experiments to test the algorithm performance. Some conclusions are concluded below.

1. Our SPMV algorithm includes domain decomposition, compact-reorder method, communication mechanism and HEC format. It can be sped up to 20 to 40 times faster on our four GPUs workstation compared to a traditional CPU. The SPMV algorithm has favorable scalability with the number of GPUs.

2. We have designed a nested RAS framework to use the parallel capacity of multiple-GPU architecture. The outer RAS is designed for the parallelization at multiple-GPU level. The inner RAS is designed for parallelization at GPU-thread level. According to the numerical experiments, the GMRES(20) with ILU(0) achieves 20 to 30 times speedup on our four GPUs workstation compared to the same nested RAS algorithm in a single thread on a CPU. Our nested RAS framework is effective and provides great parallel performance on multiple-GPU architecture. The GMRES algorithm and ILU(0) have a promising application prospect.

3. We have implemented a decoupled ILU(k) including a symbolic phase and a factorization phase. A parallel triangular solver is employed on each GPU to solve triangular matrices. The running results show the multi-aspect characteristics of ILU(k). High k imports fill-in elements, which leads to high calculation accuracy and low parallel performance. The CPU time and iteration go down and the GPU time goes up as k increases. Thereby the speedup goes down as k increases. In a word, higher k leads to better convergence and worse speedup. These experimental results provide some enlightenments for further ILU(k) study.

In scientific computing, block-wise matrices are frequently utilized. For example, a reservoir is divided into many grid blocks in three dimensions in a saturated case of the black oil model. Each grid block has three unknowns which are the pressure of the oil phase, the saturation of the water phase and the saturation of the gas phase. If these three unknowns in each block are numbered consecutively, the discretization linear system of the mass equilibrium equations from the Newton method is a block-wise matrix where each block is a 3×3 submatrix. For other reservoir models, such as a compositional model or a thermal model, each grid block has more unknowns. Because the block size is determined by the number of unknowns in a block, it is a variable and depends on the underlying model. A preconditioner algorithm must be compatible to block matrices with a variable block size. Because a block-wise ILU(k) has a low condition number and may have more stable performance than a point-wise matrix has, to study an algorithm of solving large-scale block-wise matrices with variable block size on a multiple-GPU architecture is attractive future work.

REFERENCE

[1] Z. Chen, *Reservoir Simulation: Mathematical Techniques in Oil Recovery*, CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 77, SIAM, Philadelphia, 2007.

[2] Z. Chen, G. Huan, and Y. Ma, *Computational Methods for Multiphase Flows in Porous Media*, in the Computational Science and Engineering Series, Vol. 2, SIAM, Philadelphia, 2006.

[3] S.Yang, Z.Chen, W. Wu, Y. Zhang, and X. Zhang, *Addressing Microseismic Uncertainty from Geological Aspects to Improve Accuracy of Estimating Stimulated Reservoir Volumes*, SPE-174286-MS, Research Institute of the Henan Oilfield, Sinopec,and Haoyun Deng, China University of Geosciences, Beijing, 2015.

[4] J. Xu, Z. Chen, Y. Yu, J. Cao, *Numerical Thermal Simulation and Optimization of Hybrid CSS/SAGD Process in Long Lake with Lean Zones*, SPE-170149-MS, Society of Petroleum Engineers, SPE Heavy Oil Conference-Canada, Calgary, Alberta, Canada, 10-12 June 2014.

[5] Q. Song, Z. Chen, S. M. Farouq Ali, *Steam Injection Schemes for Bitumen Recovery from the Grosmont Carbonate Deposits*, SPE-174463-MS, Society of Petroleum Engineers, SPE Canada Heavy Oil Technical Conference, Calgary, Alberta, Canada, 09-11 June 2015.

[6] M. Lin, S. Chen, W. Ding, Z. Chen, J. Xu, *Effect of Fracture Geometry on Well Production in Hydraulic-Fractured Tight Oil Reservoirs*, SPE-167761-PA, Society of Petroleum Engineers, Journal of Canadian Petroleum Technology, Volume 54, Issue 03, May 2015.

[7] Y. Saad, *Iterative methods for sparse linear systems (2nd edition)*, SIAM, 2003.

[8] H. Liu, K. Wang, Z. Chen, and K. E. Jordan, *Efficient Multi-stage Preconditioners for Highly Heterogeneous Reservoir Simulations on Parallel Distributed Systems*, SPE-173208-MS, SPE Reservoir Simulation Symposium Held in Houston, Texas, USA, 23-25 February 2015.

[9] H. Liu, B. Yang, Z. Chen, *Accelerating the GMRES Solver with Block ILU(k) Preconditioner on GPUs in Reservoir Simulation*, Journal of Geology & Geosciences, 4(199), 2015.

[10] H. Liu, K. Wang, Z. Chen, J. Luo, S. Wu, B. Wang, *Development of Parallel Reservoir Simulators on Distributed-memory Supercomputers*, SPE-175573-MS, SPE Reservoir Characterisation and Simulation Conference and Exhibition, Abu Dhabi, UAE, 14-16 September, 2015.

[11] H. Liu, K. Wang, Z. Chen, K. E. Jordan, J. Luo, *A Parallel Platform for Reservoir Simulators on Distributed-memory Supercomputers*, SPE-176045-MS, SPE/IATMI Asia Pacific Oil & Gas Conference and Exhibition, Nusa Dua, Indonesia, 20-22 October, 2015.

[12] D. W. Peaceman, *Interpretation of Well-block Pressures in Numerical Reservoir Simulation*, SPE 6893, The 52nd Annual Fall Technical Conference and Exhibition, Denver, CO. 1977A.

[13] K. Aziz and A. Settari, *Petroleum Reservoir Simulation*, Applied Science Publishers Ltd., London, U.K., 1979.

[14] Ertekin et al., *Basic Applied Reservoir Simulation*, 2001.

[15] M. Botchev, *A.N.Krylov, a Short Biography*, 6 Dec. 2012.

[16] A. T. Grigorian, *"Krylov, Aleksei Nikolaevich." Complete Dictionary of Scientific Biography*, 2008. Encyclopedia.com. 6 Dec. 2012.

[17] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Vander Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, 1994.

[18] P. K. W Vinsome, *an Iterative Method for Solving Sparse Sets of Simultaneous Linear Equations*, SPE Symposium on Numerical Simulation of Reservoir Performance, Los Angeles, California, 1976.

[19] D. B. Kirk, W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, ISBN: 978-0-12-381472-2.

[20] X. Hu, W. Liu, G. Qin, J. Xu, Y. Yan, C. Zhang, *Development of A Fast Auxiliary Subspace Pre-conditioner for Numerical Reservoir Simulators*, SPE Reservoir Characterisation and Simulation Conference and Exhibition, Abu Dhabi, UAE, SPE-148388-MS, 9-11 October 2011.

[21] H. Cao, H. A. Tchelepi, J. R. Wallis, H. E. Yardumian, *Parallel Scalable Unstructured CPR-type Linear Solver for Reservoir Simulation*, SPE Annual Technical Conference and Exhibition, 2005.

[22] J. R. Wallis, R. P. Kendall, and T. E. Little. *Constrained Residual Acceleration of Conjugate Residual Methods*, SPE Reservoir Simulation Symposium. 1985.

[23] S. Balay, W. Gropp, L. McInnes and B. Smith, *The Portable, Extensible Toolkit for Scientific Computing*, Version 2.0.13, 1996.

[24] X.-C. Cai and M. Sarkis, *A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems*, SIAM J. Sci. Comput., 21, 1999, pp. 792-797.

[25] L. Zhang, *A Parallel Algorithm for Adaptive Local Refinement of Tetrahedral Meshes Using Bisection*, Numer. Math.: Theory, Methods and Applications, 2, 2009, pp. 65-89.

[26] Z. Chen and Y. Zhang, *Development, Analysis and Numerical Tests of a Compositional Reservoir Simulator*, International Journal of Numerical Analysis and Modeling 4,2008 , pp. 86-100.

[27] Z. Chen and Y. Zhang, *Well Flow Models for Various Numerical Methods*, International Journal of Numerical Analysis and Modeling 6,2009 , pp. 375-388.

[28] R. Li and Y. Saad, *GPU-accelerated Preconditioned Iterative Linear Solvers*, Technical Report Umsi-2010-112, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2010.

[29] NVIDIA Corporation, *Nvidia CUDA Programming Guide*, (Version 3.2), 2010.

[30] NVIDIA Corporation, *CUDA C Best Practices Guide (Version 3.2)*, 2010.

[31] N. Bell and M. Garland, *Efficient Sparse Matrix-vector Multiplication on CUDA*, NVIDIA Technical Report, NVR-2008-004, NVIDIA Corporation, 2008.

[32] N. Bell and M. Garland, *Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors*, Proc. Supercomputing, November 2009, pp. 1-11.

[33] H. Liu, S. Yu, Z. Chen, B. Hsieh and L. Shao, *Sparse Matrix-vector Multiplication on NVIDIA GPU*, International Journal of Numerical Analysis & Modeling, Series B, Volume 3, 2012, No. 2, pp. 185-191.

[34] Z. Chen, H. Liu, S. Yu, B. Hsieh and L. Shao, *GPU-based Parallel Reservoir Simulators*, Proc. of 21st International Conference on Domain Decomposition Methods, France, 2012.

[35] S. Yu, H. Liu, Z. Chen, B. Hsieh, and L. Shao, *GPU-based Parallel Reservoir Simulation for Large-scale Simulation Problems*, SPE Europec/EAGE Annual Conference, Copenhagen, Denmark, 2012.

[36] G. Haase, M. Liebmann, C. C. Douglas and G. Plank, *A Parallel Algebraic Multigrid Solver on Graphics Processing Units*, High Performance Computing and Applications, 2010, pp. 38-47.

[37] H. Klie, H. Sudan, R. Li, and Y. Saad, *Exploiting Capabilities of Many Core Platforms in Reservoir Simulation*, SPE RSS Reservoir Simulation Symposium, February 2011, pp. 21-23.

[38] R. Li and Y. Saad, *GPU-accelerated Preconditioned Iterative Linear Solvers*, Technical Report Umsi-2010-112, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2010.

[39] H. Liu, S. Yu, Z. Chen, B. Hsieh and L. Shao, *Parallel Preconditioners for Reservoir Simulation on GPU*, SPE 152811-PP, SPE Latin American and Caribbean Petroleum Engineering Conference Held in Mexico City, Mexico, 16-18 April 2012.

[40] Z. Chen, H. Liu and B. Yang, *Accelerating Iterative Linear Solvers Using Multiple Graphical Processing Units*, Internaltional Journal of Computer Mathematics, Volume 92, Issue 7, 2015, pp. 1422-1438.

[41] H. Liu, Z. Chen, S. Yu, B. Hsieh and L Shao, *Development of a Restricted Additive Schwarz Preconditioner for Sparse Linear Systems on NVIDIA GPU*, International Journal of Numerical Analysis & Modeling, Series B, 5, 2014, pp. 13-20.

[42] Z. Chen, H. Liu and S. Yu, *Development of Algebraic Multigrid Solvers Using GPUs*, SPE-163661-MS, SPE Reservoir Simulation Symposium, 18-20 February, The Woodlands, Texas, USA, 2013.

[43] J. Bolz, I. Farmer, E. Grinspun and P. Schröder, *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*, Symposium A Quarterly Journal In Modern Foreign Literatures, 22(3), 2007, pp. 917-924.

[44] L. Buatois, G. Caumon and B. Lévy, *Concurrent Number Cruncher: an Efficient Sparse Linear Solver on the GPU*, High Performance Computing and Communications, 4782, 2007, pp. 358-371.

[45] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker and S. Turek, *Using GPUs to Improve Multigrid Solver Performance on a Cluster*, International Journal of Computational Science and Engineering, 4(1), 2008, pp. 36-55.

[46] J. Brannick, Y. Chen, X. Hu and L. Zikatanov, *Parallel Unsmoothed Aggregation Algebraic Multigrid Algorithms on GPUs*, Springer Processings in Mathematics and Statistics, vol. 45, 2013, pp. 81-102.

[47] H. Liu, Z. Chen and B. Yang, *Accelerating Preconditioned Iterative Linear Solvers on GPU*, International Journal of Numerical Analysis & Modeling, Series B, 5, 2014, pp. 136-146.

[48] G. Karypis and V. Kumar, *A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs*, SIAM Journal on Scientific Computing, 20(1), 1999, pp. 359-392.

[49] NVIDIA Corporation, CUSP: *Generic Parallel Algorithms for Sparse Matrix and Graph*, `http://code.google.com/p/cusp-library/`.

[50] N. Bell, S. Dalton and L. Olson, *Exposing Fine-grained Parallelism in Algebraic Multigrid Methods*, SIAM Journal on Scientific Computing, 34(4), 2012, pp. 123-152.

[51] Z. Chen, H. Liu and B. Yang, *Parallel Triangular Solvers on GPU*, Proceedings of International Workshop on Data-Intensive Scientific Discovery (DISD), Shanghai University, Shanghai, China, 1-4 August 2013.

[52] M. Naumov, *Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU*, NVIDIA Technical Report, June 2011.

[53] L. Wang, X. Hu, J. Cohen and J. Xu, *A Parallel Auxiliary Grid Algebraic Multigrid Method for Graphic Processing Unit*, SIAM Journal on Scientific Computing, 35(3), 2013, pp. 263-283.

[54] T. A. Davis, *University of Florida Sparse Matrix Collection*, NA digest, 1994, `https://www.cise.ufl.edu/research/sparse/matrices/`.

[55] Y. Saad and M. H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comput., 7:856-869, 1986. doi:10.1137/0907058.

[56] NVIDIA Official Website, http://www.nvidia.com/object/tesla-servers.html.

[57] NVIDIA Official Website, http://www.nvidia.com/object/why-choose-tesla.html.

[58] Fujitsu Official Website, http://techcommunity.ts.fujitsu.com/en/client-computing-devices-2/d/uid-5911b36b-324b-fc23-45fa-2438e4c546f3.html.

[59] Intel Official Website, http://ark.intel.com/products/82930.

[60] NVIDIA Official Website, http://www.nvidia.com/object/cuda_home_new.html.

[61] Wikipedia, http://en.wikipedia.org/wiki/CUDA.

[62] NVIDIA Developer Zone, https://developer.nvidia.com/about-cuda.

[63] J. L. Hennessy, D. A. Patterson, J. R. Larus *Computer Organization and Design : the Hardware/Software Interface (2. ed., 3rd Print. ed.).* San Francisco: Kaufmann. ISBN 1-55860-428-6, 1999.

[64] M. R. Hestenes, Ed. Stiefel *Methods of Conjugate Gradients for Solving Linear Systems*, Journal of Research of the National Bureau of Standards 49 (6), December 1952.

[65] R. Fletcher, G. Watson, Ed. Alistair *Conjugate Gradient Methods for Indefinite Systems*, Numerical Analysis. Lecture Notes in Mathematics (Springer Berlin / Heidelberg), 1976, 506: pp. 73-89.

[66] H. A. Van der Vorst *Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems*, SIAM J. Sci. and Stat. Comput. 1992, 13 (2): pp. 631-644.

[67] C. Paige and M. Saunders *Solution of Sparse Indefinite Systems of Linear Equations*, SIAM J. Numer. Anal. 12, 1975, pp. 617-629.

[68] C. Lanczos *An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators*, J. Res. Natl Bur. Std. 45, 1950, pp. 225-282.

[69] CUDA Toolkit Documentation, `http://docs.nvidia.com/cuda/cusparse/#hybrid-format-hyb`.

[70] GPU-Accelerated Libraries, `https://developer.nvidia.com/gpu-accelerated-libraries`.

[71] K. Stüben, *A Review of Algebraic Multigrid*, Journal of Computational and Applied Mathematics Volume 128, Issues 1-2, 2001, pp. 281-309.

[72] J. W. Ruge and K. Stüben, *Algebraic Multigrid (AMG)*, in: S.F. McCormick (Ed.), Multigrid Methods, Frontiers in Applied Mathematics, Vol. 5, SIAM, Philadelphia, 1986.

[73] A. Brandt, S.F. McCormick and J. Ruge, *Algebraic Multigrid (AMG) for Sparse Matrix Equations* D.J. Evans (Ed.), Sparsity and Its Applications, Cambridge University Press, Cambridge, 1984, pp. 257-284.

[74] C. Wagner, *Introduction to Algebraic Multigrid*, Course Notes of an Algebraic Multigrid Course at the University of Heidelberg in the Wintersemester, 1999.

[75] P. S. Vassilevski, *Lecture Notes on Multigrid Methods*, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2010.

[76] R. Falgout, A. Cleary, J. Jones, E. Chow, V. Henson, C. Baldwin, P. Brown, P. Vassilevski, and U. M. Yang, Hypre Home Page, 2011, `http://acts.nersc.gov/hypre`.

[77] A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, G. N. Miranda, and J.W. Ruge, *Robustness and Scalability of Algebraic Multigrid*, SIAM J. Sci. Comput., 21, 2000, pp. 1886-1908.

[78] V. E. Henson and U. M. Yang, *BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner*, Applied Numerical Mathematics, 41, 2000, pp. 155-177.

[79] U. M. Yang, *Parallel Algebraic Multigrid Methods - High Performance Preconditioners*, Chapter in Numerical Solution of Partial Differential Equations on Parallel Computers, A.M. Bruaset and A. Tveito, eds., Springer-Verlag, 51, 2006, pp. 209-236.

[80] R. D. Falgout, *An Introduction to Algebraic Multigrid*, Computing in Science and Engineering, Special Issue on Multigrid Computing, 8, 2006, pp. 24-33.

Appendix A

MATRIX PROPERTIES

Many matrices used in Chapter 5 are downloaded from a matrix market [54]. The properties of these matrices are listed below. The number of rows, the number of columns and nonzeros of all the matrices are recounted and validated according to our program. More details about these matrix properties can be found online [54].

Matrix	**Description**
BenElechi1	problem with many sparse right-hand sides, S. Ben Elechi
af_shell8	Olaf Schenk, Univ. Basel: AutoForm Eng. GmbH, Zurich. sheet metal forming. positive definite
parabolic_fem	Parabolic FEM, diffusion-convection reaction, constant homogenous diffusion
tmt_sym	symmetric electromagnetics problem, David Isaak, Computational_EM_Works
ecology2	circuitscape: circuit theory applied to animal/gene flow. B. McRae, UCSB
thermal2	unstructured FEM, steady state thermal problem. Dani Schmid, Univ. Oslo
atmosmodd	Atmospheric models, Andrei Bourchtein
atmosmodl	Atmospheric models, Andrei Bourchtein
Hook_1498	3D model of a steel hook with tetrahedral finite elements
G3_circuit	circuit simulation problem, Ufuk Okuyucu, AMD, Inc.
kkt_power	Optimal power flow, nonlinear optimization (KKT), Fabrice Zaoui, RTE, France
memchip	Memory chip from K. Gullapalli, Freescale Semiconductor

Table A.1: Matrix description

(a)

Matrix properties	
number of rows	245,874
number of columns	245,874
nonzeros	6,698,185
structural full rank?	yes
structural rank	245,874
# of blocks from dmperm	7
# strongly connected comp.	7
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

(b)

author	S. Ben Elechi
editor	T. Davis
date	2007
kind	2D/3D problem
2D/3D problem?	yes

Table A.2: Properties and information for *BenElechi*1

(a)

Matrix properties	
number of rows	504,855
number of columns	504,855
nonzeros	9,042,005
structural full rank?	yes
structural rank	504,855
# of blocks from dmperm	1
# strongly connected comp.	1
explicit zero entries	9,720
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

(b)

author	AutoForm Eng.
editor	O. Schenk
date	2003
kind	subsequent structural problem
2D/3D problem?	yes

Table A.3: Properties and information for af_shell8

(a)

Matrix properties	
number of rows	525,825
number of columns	525,825
nonzeros	2,100,225
structural full rank?	yes
structural rank	525,825
# of blocks from dmperm	1
# strongly connected comp.	1
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

(b)

author	P. Wissgott
editor	T. Davis
date	2007
kind	computational fluid dynamics problem
2D/3D problem?	yes

Table A.4: Properties and information for *parabolic_fem*

(a)

Matrix properties	
number of rows	726,713
number of columns	726,713
nonzeros	2,903,837
structural full rank?	yes
structural rank	726,713
# of blocks from dmperm	1
# strongly connected comp.	1
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

(b)

author	D. Isaak
editor	T. Davis
date	2008
kind	electromagnetics problem
2D/3D problem?	yes

Table A.5: Properties and information for *tmt_sym*

(a)

Matrix properties	
number of rows	999,999
number of columns	999,999
nonzeros	2,997,995
structural full rank?	yes
structural rank	999,999
# of blocks from dmperm	1
# strongly connected comp.	1
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

(b)

author	B. McRae
editor	T. Davis
date	2008
kind	2D/3D problem
2D/3D problem?	yes

Table A.6: Properties and information for *ecology*2

(a)

Matrix properties	
number of rows	1,228,045
number of columns	1,228,045
nonzeros	4,904,179
structural full rank?	yes
structural rank	1,228,045
# of blocks from dmperm	959
# strongly connected comp.	959
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

(b)

author	D. Schmid
editor	T. Davis
date	2006
kind	thermal problem
2D/3D problem?	yes

Table A.7: Properties and information for *thermal*2

(a)

Matrix properties	
number of rows	1,270,432
number of columns	1,270,432
nonzeros	8,814,880
structural full rank?	yes
structural rank	1,270,432
# of blocks from dmperm	1
# strongly connected comp.	1
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	67%
type	real
structure	unsymmetric
Cholesky candidate?	no
positive definite?	no

(b)

author	A. Bourchtein
editor	T. Davis
date	2009
kind	computational fluid dynamics problem
2D/3D problem?	yes

Table A.8: Properties and information for *atmosmodd*

(a)

Matrix properties	
number of rows	1,489,752
number of columns	1,489,752
nonzeros	10,319,760
structural full rank?	yes
structural rank	1,489,752
# of blocks from dmperm	1
# strongly connected comp.	1
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	67%
type	real
structure	unsymmetric
Cholesky candidate?	no
positive definite?	no

(b)

author	A. Bourchtein
editor	T. Davis
date	2009
kind	computational fluid dynamics problem
2D/3D problem?	yes

Table A.9: Properties and information for *atmosmodl*

(a)

Matrix properties	
number of rows	1,498,023
number of columns	1,498,023
nonzeros	30,436,237
structural full rank?	yes
structural rank	1,498,023
# of blocks from dmperm	30,001
# strongly connected comp.	30,001
explicit zero entries	1,542,994
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

(b)

author	C. Janna, M. Ferronato
editor	T. Davis
date	2011
kind	structural problem
2D/3D problem?	yes

Table A.10: Properties and information for $Hook\_1498$

(a)

Matrix properties	
number of rows	1,585,478
number of columns	1,585,478
nonzeros	4,623,152
structural full rank?	yes
structural rank	1,585,478
# of blocks from dmperm	1
# strongly connected comp.	1
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

(b)

author	U. Okuyucu
editor	T. Davis
date	2006
kind	circuit simulation problem
2D/3D problem?	no

Table A.11: Properties and information for *G3_circuit*

(a)

Matrix properties	
number of rows	2,063,494
number of columns	2,063,494
nonzeros	7,209,692
structural full rank?	yes
structural rank	2,063,494
# of blocks from dmperm	9,733
# strongly connected comp.	9,611
explicit zero entries	1,841,302
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	no
positive definite?	no

(b)

author	F. Zaoui
editor	T. Davis
date	2007
kind	optimization problem
2D/3D problem?	no

Table A.12: Properties and information for *kkt_power*

(a)

Matrix properties	
number of rows	2,707,524
number of columns	2,707,524
nonzeros	13,343,948
structural full rank?	yes
structural rank	2,707,524
# of blocks from dmperm	63
# strongly connected comp.	53
explicit zero entries	1,466,254
nonzero pattern symmetry	91%
numeric value symmetry	40%
type	real
structure	unsymmetric
Cholesky candidate?	no
positive definite?	no

(b)

author	K. Gullapalli
editor	T. Davis
date	2010
kind	circuit simulation problem
2D/3D problem?	no

Table A.13: Properties and information for *memchip*