**University of Calgary**

**PRISM Repository** **https://prism.ucalgary.ca**

The Vault Open Theses and Dissertations

2013-01-30

# Parallel Black Oil Solvers on GPU

YU, SONG

YU, SONG. (2013). Parallel Black Oil Solvers on GPU (Master's thesis, University of Calgary, Calgary, Canada). Retrieved from https://prism.ucalgary.ca. doi:10.11575/PRISM/27253 http://hdl.handle.net/11023/530

*Downloaded from PRISM Repository, University of Calgary*

UNIVERSITY OF CALGARY

# Parallel Black Oil Solvers on GPU

by

Song Yu

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF CHEMICAL AND PETROLEUM ENGINEERING

CALGARY, ALBERTA
JANUARY, 2013

© Song Yu 2013

# Abstract

Reservoir simulation for a full field heterogeneous model with millions of grid blocks demands significant computational time; therefore, improving the computational efficiency becomes crucial in designing a reservoir simulator. Graphics Processing Unit (GPU), a new high-profile parallel processor with hundreds of microprocessors, stands out in parallel simulation because of its efficient power utilization and high computational efficiency. Also, its cost is relatively low, making large-scale parallel reservoir simulation possible for most of the desktop users.

In this thesis, a GPU-based parallel linear solver package, named Parallel Accelerated Simulation Solvers (PASS), is developed to speed up the reservoir simulation using the GPU. For this new solver package, several different linear solvers and preconditioners have been implemented based on GPU. For solvers, it has GMRES, BICGSTAB and ORTHOMIN, which are commonly used in reservoir simulators. For preconditioners, a group of ILU preconditioners are developed on GPU, including ILU(0), ILUT, block ILU(0) and block ILUT.

In the numerical experiments performed, the SPE10 problem, a 3D heterogeneous model with over one million grid blocks, is selected to test the speedup of the GPU solver. On the state-of-the-art CPU and GPU platforms, the new GPU implemen-

tation is able to achieve a speedup of over 8 times in solving linear systems arising from this SPE10 problem compared with the single CPU based sequential solver. Moreover, the GPU solver is successfully coupled with an in-house black-oil reservoir simulator to test the performance of the whole parallel simulation process, with a speedup of around 6 times. The excellent speedup and accurate results demonstrate that the GPU-based parallel linear solver has a great potential in the parallel reservoir simulation.

# Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Zhangxin Chen for his encouragement, support and guidance throughout this work. I deeply appreciate his reading of my drafts and the valuable input to this thesis.

I also thank Dr. Hui Liu, the post doctor fellow in our group, for leading this project and guiding me both on mathematics and programming.

I also want to give my thanks to my colleagues and friends, who help me within and outside my academic life.

Special thanks are due to my parents, for their love and support.

# Table of Contents

**Abstract** ii

**Acknowledgements** iv

**Table of Contents** v

**List of Tables** vii

**List of Figures** viii

**1 OVERVIEW** 1
- 1.1 Reservoir Simulation . . . 1
- 1.2 Linear Solver . . . 2
- 1.3 GPU Computing . . . 3
- 1.4 GPU Based Reservoir simulation . . . 7
- 1.5 Literature Review . . . 8
- 1.6 Structure of Thesis . . . 8

**2 BLACK OIL MODEL** 10
- 2.1 Introduction . . . 10
- 2.2 Basic Differential Equations . . . 10
- 2.3 Fluid Properties . . . 15
  - 2.3.1 Density . . . 15
  - 2.3.2 Viscosity . . . 16
- 2.4 Rock/Fluid Properties . . . 17
- 2.5 Phase States . . . 18
- 2.6 Numerical Solution of the Black Oil Model . . . 19
  - 2.6.1 Solution Techniques . . . 20

**3 GPU COMPUTING** 30
- 3.1 Introduction to CUDA . . . 30

3.1.1 CUDA Program Structure . . . 30
3.1.2 Device Memories and Data Transfer . . . 32
3.1.3 Kernel Functions and Threading . . . 32
3.2 CUDA Memories . . . 33
3.2.1 Global and Constant Memory . . . 33
3.2.2 Registers and Shared Memory . . . 33
3.3 Memory Coalescing . . . 34

**4 GPU-BASED PARALLEL SOLVER 36**
4.1 PASS Overview . . . 36
4.2 Sparse Matrix Storage Format . . . 38
4.2.1 COO . . . 38
4.2.2 CSR . . . 39
4.2.3 ELL . . . 39
4.2.4 HYB . . . 40
4.2.5 HEC . . . 40
4.3 Sparse Matrix-Vector Multiplication . . . 41
4.3.1 SPMV test . . . 42
4.4 Solvers . . . 43
4.4.1 GMRES . . . 43
4.4.2 BICGSTAB . . . 46
4.4.3 ORTHOMIN . . . 47
4.4.4 Solver Comparison . . . 47
4.5 Preconditoning . . . 48
4.5.1 ILU . . . 50
4.5.2 Triangular solver . . . 53
4.6 ILU test . . . 57

**5 GPU-BASED RESERVOIR SIMULATION 59**
5.1 SPE10 Problem . . . 59
5.2 GPU-based Simulation . . . 60
5.3 Simulation Test . . . 61

**6 CONCLUSIONS AND FUTURE WORK 65**
6.1 Conclusions . . . 65
6.2 Future Work . . . 66

**Bibliography 68**

**A SPE10 datafile 72**

# List of Tables


4.1 Matrices used for testing . . . . . 43
4.2 Speedup of SPMV, Double Precision . . . . . 44
4.3 SPE10 matrix description . . . . . 48
4.4 solver parameter . . . . . 48
4.5 Performance comparison of solvers for SPE10-1 . . . . . 48
4.6 Performance comparison of solvers for SPE10-2 . . . . . 49
4.7 Performance of SPE10 . . . . . 57
4.8 Performance of SPE10 . . . . . 58

5.1 Experimental platform . . . . . 64
5.2 Sovler parameters . . . . . 64
5.3 Performance of PASS for SPE10 problem . . . . . 64

# List of Figures


1.1 GPUs are scaling with a rate much greater than CPUs. The Figure shows how some of the modern CPUs compare with modern GPUs in producing flops[11]. . . . . 5
1.2 Architecture of a CUDA-capable GPU. . . . . 6
1.3 Suitable components to migrate to GPU[30]. . . . . 7

2.1 Relative permeability curves in a three-phase system. . . . . 17

3.1 CUDA Proramming model [11]. . . . . 31
3.2 The Architecture of GPU [11]. . . . . 32
3.3 CUDA Memory Hierarchy[11]. . . . . 35

4.1 Parallel accelerated simulation solver(PASS). . . . . 37
4.2 COO format. . . . . 38
4.3 CSR format. . . . . 39
4.4 ELL format. . . . . 40
4.5 HEC format. . . . . 41
4.6 Distribution of non-zero elements in ILU(0) factorization. . . . . 51
4.7 Block ILU factorization[]. . . . . 54

5.1 SPE10 porosity distribution. . . . . 60
5.2 Logarithm of horizontal and vertical permeability seen from below. . 61
5.3 Workflow of GPU-based simulation. . . . . 62
5.4 Workflow of GPU solver. . . . . 63

# Chapter 1

# OVERVIEW

## 1.1 Reservoir Simulation

In the oil and gas industry, reservoir simulation is a major tool to model the reservoir physics and make predictions and optimizations. An oil and gas recovery project often takes a huge investment with hundreds of millions of dollars. The high risk of field operations must be assessed and minimized beforehand. Reservoir simulation presents a relatively accurate modeling of reservoir physics with taking into account the complex mechanisms of a recovery process. The major goal of reservoir simulation is to predict future performance of the reservoir and find ways to optimize the recovery of oil or gas under various operating conditions.

The numerical simulation and modeling stages [7] include: establishment of physical models, development of mathematical models, discretization of these models and design of computer algorithms.

From mathematical point of view, reservoir simulation is the art and science tool of using numerical methods to solve the fluid flow and heat transfer equations in porous media. To improve the accuracy and efficiency of reservoir modeling and simulation, researchers in reservoir simulation mainly focus on the following areas: gridding, fluid modeling, numerical approximations, linear solvers, and reservoir and

geological modelling, etc.

Generally, reservoir simulators can be classified by the fluid types. Black oil simulators are conventional simulators without taking into account the compositional changes between fluids; Compositional oil simualtors are used to model multicompoent flow and mass transfer between different phases; Thermal oil simulators are developed for thermal recovery with a nonisothermal condition. In this thesis, we mainly focus on the black oil reservoir simulator which will be introduced in Chapter 2.

## 1.2 Linear Solver

In the reservoir simulation, to solve the fluid flow and heat transfer equations, numerical methods like finite difference, finite element and finite volume methods are often used to discretize these mathematical equations. A lot of matrix systems arising from those methods are sparse, unsymmetric and ill-conditioned, which are time-consuming and difficult to solve. To deal with these kinds of matrix systems, Krylov subspace solvers [26] like GMRES are the best choice, and for large-scale reservoir simulation with above one million grid blocks, a reservoir simulator always takes 80% or 90% of the total simulation time on those linear solvers; particularly, for highly heterogeneous models, it is difficult for solvers to converge. Therefore, to develop a fast and stable linear solver is a very important part for the simulator development. In the Krylove subspace solvers, there are a lot of different algorithms, including GMRES, BiCGSTAB, ORTHORMIN and CGS. These methods differ fundamentally in their capabilities, and performs differently in terms of computation, storage cost and convergence ability. Furthermore, these solvers often fail to converge without

preconditioning. The performance of iterative solvers depends highly on the quality of preconditioners. In the petroleum industry, even in all the scientific computation fields, most of the effort is put on developing better preconditioners.

Besides the effort of improving the stability and convergence of solvers, the parallelization of solvers is also an essential technique to speed up the reservoir simulation. The common way of the development of parallel solvers is to develop a highly parallel BLAS operations, sparse matrix-vector product (SPMV), and fast triangular solvers using MPI or OpenMP on multiple processors. Based on those operations, a fast liner solver can be built. In this thesis, we focus on building those parallel solvers using CUDA on GPU to speed up reservoir simulation.

## 1.3 GPU Computing

Microprocessor-based single Central Processors such as Intel Pentium and AMD families have driven rapid development of computer application in performance. They brought giga floating processors per second to desktops and hundreds of giga floating processors per second to cluster servers. For computer users and developers, they are fascinated with every improvement of hardware, which can accelerate their software program. However, the speed has slowed down since 2003 due to energy consumption and heat dissipation issues.

Under such circumstance, most of processor vendors have switched to multiple processing units referred to as processor cores designed on one chip. so customers of computers expect that computer applications can run faster on the new powerful multiple processors. The simple expectation seems reasonable. However, it is not actually the real case. for previously most of programs or software are sequential,

which can only be executed on one processor, and as a result, performance will not improve even with multiple processor units. Therefore, to develop parallel programs which can run on different cores has been the major topic for software engineers and especially in the high performance computing region. These parallel programs are supposed to solve large-scale problems and execute on multiple processors. Now, with the increasing development of this new microprocessor, lots of parallel software are expected to be developed.

Since 2003, the semiconductor industry has formed two major directions of designing microprocessors: multi-core processors and many-core processors. A multi-core processor is a single component with two or more independent actual processors, beginning with dual-core processors and currently Intel Core i7 microprocessor with four processor cores. The multi-core processor is out-of-order, multiple instruction processors and only designed to maximize the execution speed of sequential programs. In contrast, the many-core processor was designed to integrate a large number of much smaller cores, which are the same in-order single instruction processor. A current exemplar is NVIDIA GeForce GTX 280 graphics processing unit (GPU) with 240 small cores. In terms of the floating -point performance, GPU has led the trend and continued to improve relentlessly; instead, the improvement of multi-core CPU has slowed significantly, which is illustrated clearly in Figure 1.1. The great performance gap between many-core GPUs and general-purpose CPUs is determined by the physical architecture and memory bandwidth. From the point of architecture, CPUs use sophisticated control logic to allow instructions from a single thread of execution to execute in parallel or even out of the sequential order while maintaining the appearance of sequential execution. More specifically, large cache memories are provided to

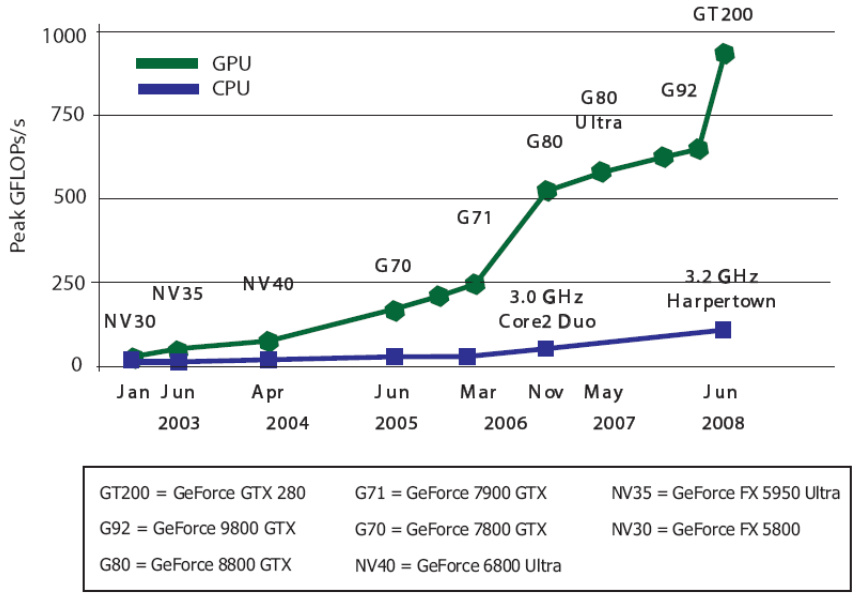


Figure 1.1: GPUs are scaling with a rate much greater than CPUs. The Figure shows how some of the modern CPUs compare with modern GPUs in producing flops[11].

reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation speed. In conclusion, the design of CPU is optimized for sequential code performance. For memory bandwidth, GPUs have been operating at approximately 10 times the bandwidth of contemporaneously available CPU chips. Many recent GPUs are able to move data at over 100 gigabytes per second (GB/s) in and out of its main dynamic random access memory; however, the bandwidth of CPU cannot grow beyond 50 GB/s for about 3 years.

GPU is developed by the video game industry which exerts tremendous economic pressure for the ability to perform a massive number of floating -point calculations. So, for this advantage, GPU is designed as numeric computing engines, and it will not perform well on some tasks on which CPUs are designed to perform well, therefore,

CPUs and GPUs are supposed to cooperate to perform the parallel program, in which CPU executes the sequential part and GPU works on numerically intensive parts. This is why the CUDA (Compute Unified Device Architecture) programming model, introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of an application.

A modern GPU architecture is shown in Figure 1.2. A GPU card consists of several SM multiprocessors, each of which can generate 1024 co-resident threads. Also, in each SM, there is a number of streaming processors(sp) which share the same control logic and instruction cache. NVIDIA Tesla c2050 based on NVIDIAs next generation CUDA architecture named Fermi can be equipped with 14 SM and each SM consists of 32 with the total of 448 streaming processors. In each SM, Integer and single precision floating point operations are performed by 16 sp, and the other 16 for double precision. Each GPU currently comes up with 4 gigabytes of DRAM, named global memory on GPU. This memory is a very high bandwidth, off-chip memory.

## 1.4 GPU Based Reservoir simulation

The concept of GPU-based reservoir simulation in this thesis means to apply the GPU to reservoir simulation to speed up the whole simulation process. Since a GPU is a special parallel processor, it is only suitable for highly parallel floating point calculations, which means it cannot replace CPU in reservoir simulation. So, to apply the GPU to a reservoir simulator, we need exploit the component which is the most time-consuming and also suitable for the GPU calculations. In Figure 1.3, we list three suitable components to migrate to the GPU. Among these three components,

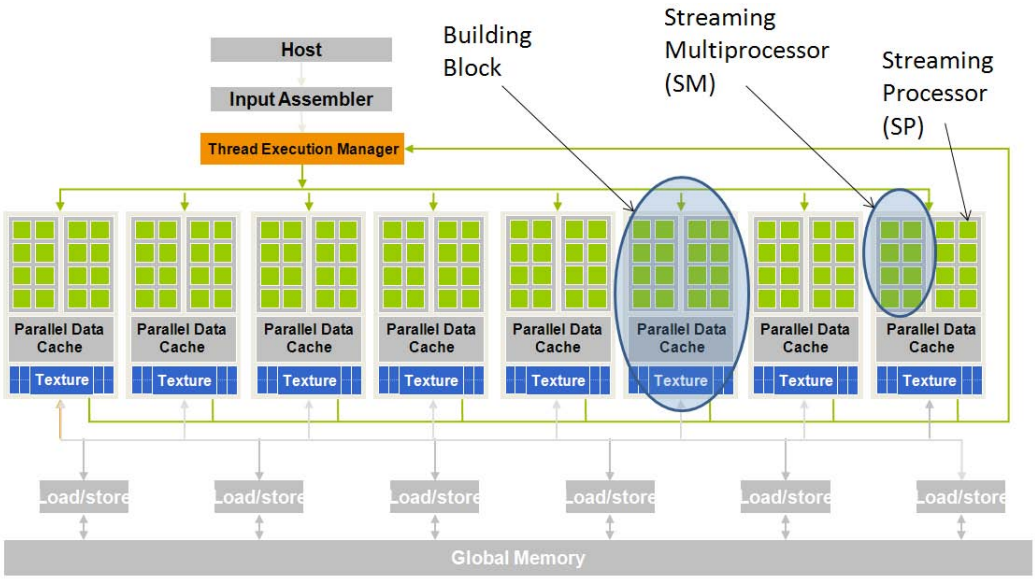


Figure 1.2: Architecture of a CUDA-capable GPU.

the linear solver is the most potential component which can be applied to the GPU and speed up our reservoir simulation. So, now, to develop a GPU-based parallel linear solver is the objective for applying the GPU to the simulation to achieve the speed-up.

## 1.5 Literature Review

Since the advent of CUDA, the massive parallelism of GPU starts to offer tremendous performance in scientific computing applications [16, 24, 1, 18]. Many researchers begin to study the performance of GPU in linear solvers. Bell and Garland [5] implemented the sparse matrix-vector multiplication on the CUDA platform for GPU with different matrix storage formats and achieved a performance of 10GELOP/s in double precision and a bandwidth of 90 GBytes/s or 63.5% of peak on Geforce

| Component | % run time & % code base | Type of algorithm |
|---|---|---|
| Property calculations | 15-30% run time<br>15+15% code base<br>Ratio ~ 1 | Largely independent cell calculations for fluid properties. Also includes well model. Many branches depending on model |
| Matrix Assembly | 15 – 20% run time<br>1% code base<br>Ratio ~ 15 | Independent assembly of governing equations for each cell |
| Linear Solver | 50 – 70% run time<br>2% code base<br>Ratio ~ 30 | Most potential but also the most sequential part of the code |

Figure 1.3: Suitable components to migrate to GPU[30].

GTX 280. Klie et al [17] designed a GPU-based linear solver using the Generalized Minimum Residual (GMRES) algorithm on Tesla C2050, and then this GPU-based solver was coupled with parallel reservoir simulator, which achieved about a speedup of three in reservoir simulation. Li and Saad [20] also implemented a GPU-based GMRES linear solver and especially, several appropriate preconditioning methods for parallel sparse linear solvers on GPU and achieved a speedup of around four. Appleyard [3] described a massively parallel incomplete factorization on GPU which is used as a preconditioner in conjunction with the GMRES algorithm to speed up large scale reservoir simulation. In their work, GPU was capable of producing a factor of 10 speed-up in the linear solver compared with the CPU based serial solver, dealing with structured grids.

## 1.6 Structure of Thesis

The structure of this thesis is described as follows:

**Chapter One**: Overview of the reservoir simulation, linear solvers and GPU computing.

**Chapter Two**: Black oil simulation model's essentials. It includes the physical model of black oil simulation and numerical techniques for the black oil model.

**Chapter Three**: An introduction of GPU computing and CUDA programming model.

**Chapter Four**: An investigation of the new GPU-based linear solver package. It discusses the architecture of the new solver package and introduces the implementation details of the components of the package including the matrix storage format, the BLAS, SPMV operations, a new triangular solvers, the Krylove solvers and preconditioners. Furthermore, the numerical experiment of the solvers is presented in this chapter to show the speedup of the GPU package in dealing with the large sparse linear systems.

**Chapter Five**: Numerical experiments of performance of the solver package in the whole simulation process. The SPE10 simulation problem is described in this chapter. The new solver package is coupled with an in-house black oil simulator to test the speedup of the whole simulation.

**Chapter Six**: Conclusions and future work.

# Chapter 2

# BLACK OIL MODEL

## 2.1 Introduction

The black oil model is a basic and common model used in reservoir simulation. It is usually used to model primary and secondary recovery, since it is an isothermal model without energy equations. In the black oil model, there are three phases: water (aqueous phase), oil (oleic phase) and gas (gas phase). The hydrocarbon system is able to be classified by two components, a non-volatile component(black oil) and a volatile component(gas) soluble in the oil phase. In detail, water only exists in the aqueous phase; the dead oil or black oil component only exists in the oleic phase, and the gas component can exist both in the gas phase and oleic phase as solution gas.

## 2.2 Basic Differential Equations

The governing equations in the black oil model are the mass balance equations for different components: water, oil and gas.

General mass balance equation:

$$Accumulation = Flux_{in} - Flux_{out} + (-)source(sink), \tag{2.1}$$

which means the mass accumulation in a grid block equals to the mass flux flowing into the block minus flux out of the block and plus (minus) the flow being injected (produced).

Mass balance is simply based on mass conservation of each component, which needs to sum up the component in different phases. For the black oil model, the mass balance equations can be expressed as follow:

for the water component:

$$\frac{\partial(\phi\rho_w S_w)}{\partial t} = -\nabla\cdot(\rho_w \mathbf{u}_w) + q_W, \tag{2.2}$$

for the oil component:

$$\frac{\partial(\phi\rho_{O_o} S_o)}{\partial t} = -\nabla\cdot(\rho_{O_o} \mathbf{u}_o) + q_O, \tag{2.3}$$

for the gas component:

$$\frac{\partial}{\partial t}(\phi\rho_{G_o} S_o + \phi\rho_g S_g) = -\nabla\cdot(\rho_{G_o} \mathbf{u}_o + \rho_g \mathbf{u}_g) + q_G, \tag{2.4}$$

Here for the gas component, it can exist in both the oil phase and gas phase, and as a result, two components exist in the oleic phase. So in the mass balance equation of the gas component, we need sum up the mass of the gas component in both phases to show the component mass conservation. Here, $\rho_{O_o}$ and $\rho_{G_o}$ indicate the partial densities of the oil and gas components in the oleic phase.

In the flux term of the mass balance equation, the volumetric flow rate $\mathbf{u}$ can be obtained by Darcy's law for each phase.

water phase:

$$\mathbf{u}_w = -\frac{1}{\mu_w}\mathbf{k}_w(\nabla p_\alpha - \rho_\alpha \wp \nabla z), \tag{2.5}$$

oil phase:

$$\mathbf{u}_o = -\frac{1}{\mu_o}\mathbf{k}_o(\nabla p_o - \rho_o \wp \nabla z), \tag{2.6}$$

gas phase:

$$\mathbf{u}_g = -\frac{1}{\mu_g}\mathbf{k}_g(\nabla p_g - \rho_g \wp \nabla z), \tag{2.7}$$

There are some extra equations to solve this model. The addition of saturation of all the phases is equal to one:

$$S_w + S_o + S_g = 1. \tag{2.8}$$

In addition, the phase pressures are related by the capillary pressure:

$$p_{cow} = p_o - p_w, \qquad p_{cgo} = p_g - p_o. \tag{2.9}$$

The relationship of $p_w$ and $p_g$ can be defined by $(p_{cow} + p_{cgo})$.

To define this model over standard volumes instead of mass, we bring in the concept of the dissolved gas oil ratio, $R_{so}$:

$$R_{so}(p, T) = V_{Gs}/V_{Os}, \tag{2.10}$$

From the concept of the oil formation volume factor $B_o$:

$$B_o = V_o/V_{Os}, \tag{2.11}$$

we can derive the oil phase density as:

$$\rho_o = \frac{R_{so}\rho_{Gs} + \rho_{Os}}{B_o} \tag{2.12}$$

and from the gas formation volume factor $B_g$ and water formation volume factor $B_w$, the phase density of water and gas can be defined as:

$$\rho_g = \frac{\rho_{Gs}}{B_g} \tag{2.13}$$

$$\rho_w = \frac{\rho_w}{B_w} \tag{2.14}$$

For the sink and source terms, they can be defined by:

$$q_W = \frac{q_{Ws}\rho_{Ws}}{B_w} \tag{2.15}$$

$$q_O = \frac{q_{Os}\rho_{Os}}{B_o} \tag{2.16}$$

$$q_G = \frac{q_{Gs}\rho_{Gs}}{B_g} + \frac{q_{Os}R_{so}\rho_{Gs}}{B_o} \tag{2.17}$$

Now, all the mass properties can be expressed by standard volumes. Moreover, to bring in the definition of transmissibility, we define:

$$\mathbf{T}_w = \frac{k_{rw}}{\mu_w B_w}\mathbf{k}, \tag{2.18}$$

$$\mathbf{T}_o = \frac{k_{ro}}{\mu_o B_o}\mathbf{k}, \tag{2.19}$$

$$\mathbf{T}_g = \frac{k_{rg}}{\mu_g B_g}\mathbf{k}, \tag{2.20}$$

Finally, we define the fluid gravities:

$$\gamma_w = \rho_w \wp, \tag{2.21}$$

$$\gamma_o = \rho_o \wp, \tag{2.22}$$

$$\gamma_g = \rho_g \wp. \tag{2.23}$$

The mass balance equations can be expressed by:

$$\frac{\partial}{\partial t}(\frac{\phi S_w}{B_w}) = \nabla \cdot (\mathbf{T}_w[\nabla p_w - \gamma_w \nabla z]) + \frac{q_{Ws}}{B_w} \tag{2.24}$$

$$\frac{\partial}{\partial t}(\frac{\phi S_o}{B_o}) = \nabla \cdot (\mathbf{T}_o[\nabla p_o - \gamma_o \nabla z]) + \frac{q_{Os}}{B_w} \tag{2.25}$$

$$\frac{\partial}{\partial t}[\phi(\frac{S_g}{B_g}+\frac{R_{so}S_o}{B_o})] = \nabla\cdot(\mathbf{T}_g[\nabla p_g-\gamma_g\nabla z]+R_{so}\mathbf{T}_o[\nabla p_o-\gamma_o\nabla z])+\frac{q_{Gs}}{B_g}+R_{so}\frac{q_{Os}}{B_w}. \tag{2.26}$$

The volumetric flow rates at wells are:

$$q_{Ws} = \sum_{\nu=1}^{N_w}\sum_{m=1}^{M_{w\nu}} WI_m^{(\nu)}\frac{k_{rw}}{\mu_w}[p_{bh}^{(\nu)} - p_w - \gamma_w(z_{bh}^{(\nu)} - z)]\delta(x - x_m^{\nu}), \tag{2.27}$$

$$q_{Ws} = \sum_{\nu=1}^{N_w}\sum_{m=1}^{M_{w\nu}} WI_m^{(\nu)}\frac{k_{ro}}{\mu_o}[p_{bh}^{(\nu)} - p_o - \gamma_o(z_{bh}^{(\nu)} - z)]\delta(x - x_m^{\nu}), \tag{2.28}$$

$$q_{Ws} = \sum_{\nu=1}^{N_w}\sum_{m=1}^{M_{w\nu}} WI_m^{(\nu)}\frac{k_{rg}}{\mu_g}[p_{bh}^{(\nu)} - p_g - \gamma_g(z_{bh}^{(\nu)} - z)]\delta(x - x_m^{\nu}), \tag{2.29}$$

where the well index is

$$WI_m^{(\nu)} = \frac{2\pi\overline{k}\Delta h}{ln(r_e/r_w)}\ |_m^{(v)}, \tag{2.30}$$

## 2.3 Fluid Properties

In this section, we are going to discuss the density and viscosity for the water, oil and gas phases and components in the black oil model.

### 2.3.1 Density

**water phase density**:

The water density is expressed by water compressiblity, $c_w$.

$$\rho_w = \frac{\rho_{Ws}}{B_{wi}}(1 + c_w(p - p^0)), \tag{2.31}$$

where $\frac{\rho_{Ws}}{B_{wi}}$ is the water density in reservoir conditions at a reference pressure, $p^0$. Since the water phase solely has a pure water component, the density of the water component is the same with the phase density.

**oil phase density**:

There are both the oil component and solution gas in the oil phase, so the oil phase density is composed of the oil component density $\rho_{Oo}$ and solution gas component density, $\rho_{Go}$:

$$\rho_o = \rho_{Oo} + \rho_{Go}. \tag{2.32}$$

The oil component density $\rho_{Oo}$ is defined by:

$$\rho_{Oo} = \frac{\rho_{Os}}{B_o}. \tag{2.33}$$

The formation volume factor of oil, $B_o$ is :

$$B_o = B_{ob}(p_b)(1 - c_o(p - p_b)), \tag{2.34}$$

where $B_{ob}$ is the formation volume factor of oil at the bubble point and $c_o$ is the oil compresssibility.

The solution gas density $\rho_{Go}$ is obtained from:

$$\rho_{Go} = \frac{R_{so}\rho_{Gs}}{B_o}, \tag{2.35}$$

**gas phase density**:

In the gas phase, it has a purely gas (pseudo) component. Then the phase density is the same with the gas component density as free gas:

$$\rho_g = \frac{\rho_{Gs}}{B_g}, \tag{2.36}$$

where

$$\rho_{Gs} = Y_G\rho_{air}, B_g = \frac{ZT}{p}\frac{p_s}{T_s}, \tag{2.37}$$

where $Y_G$ is the raw gas density, $z$ is the gas deviation factor and $p_s$ and $T_s$ are the reservoir pressure and temperature at standard conditions, respectively.

### 2.3.2 Viscosity

Usually, temperature has a great effect on hydrocarbon viscosity, but with the isothermal assumption in the black oil model, the viscosity is solely a function of pressure. The water viscosity is taken to be constant.

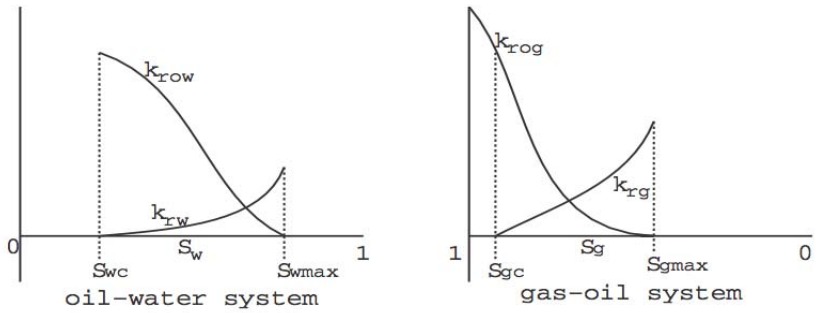


Figure 2.1: Relative permeability curves in a three-phase system.

**oil phase viscosity**:

$$\mu_o = \mu_{ob}(p_b)(1 + c_\mu(p - p_b)), \tag{2.38}$$

where $\mu_{ob}$ is the oil viscosity at $p_b$ and $c_\mu$ is the oil viscosity compressibility.

**gas phase viscosity**:

$$\mu_g = \mu_g(p). \tag{2.39}$$

## 2.4 Rock/Fluid Properties

The oil pressure will be one of the primary variables to be used:

$$p = p_o. \tag{2.40}$$

The capillary pressures $p_{cw}$ and $p_{cg}$ are assumed to be functions of the saturations only [19]:

$$p_{cow} = p_{cow}(S_w), \quad p_{cgo} = p_{cgo}(S_g). \tag{2.41}$$

Fig 2.1 illustrates relative permeability curves for a water, oil and gas system in an

isotropic porous medium. The relative permeabilities for water and gas are assumed to be of the form

$$\begin{aligned} k_{rw} &= k_{rw}(S_w), \quad k_{row} = k_{row}(S_w), \\ k_{rg} &= k_{rg}(S_g), \quad k_{rog} = k_{rog}(S_g). \end{aligned} \tag{2.42}$$

As an example, Stone's model II for the oil relative permeability is used

$$\begin{aligned} k_{ro}(S_w, S_g) = k_{rc}\Big\{ \Big[\tfrac{k_{row}(S_w)}{k_{rc}} + k_{rw}(S_w)\Big] \Big[\tfrac{k_{rog}(S_g)}{k_{rc}} + k_{rg}(S_g)\Big] \\ - k_{rw}(S_w) - k_{rg}(S_g)\Big\}, \end{aligned} \tag{2.43}$$

where $k_{rc} = k_{row}(S_{wc})$ and $S_{wc}$ is the critical saturation. Finally, the porosity $\phi$ is assumed to have the form

$$\phi = \phi^o(1 + c_R(p - p^o)), \tag{2.44}$$

where $\phi^o$ is the porosity at a reference pressure $p^o$ and $c_R$ is the rock compressibility.

## 2.5 Phase States

The secondary recovery, such as water flooding and gas injection, is a good application for the black oil model. Since temperature is constant in the black oil model, the bubble pressure is a key factor to the phase changes in the simulation process. When the oil pressure is above the bubble pressure, all the gas dissolves into the oil phase, only two phases (water and oil) are present and there is no free gas in the model, which is named the undersaturated state. During the production process with the decrease of pressure, when the pressure is below the bubble point pressure, which is the saturated state, the solution gas in the oil phase will come out of oil to the gas

phase, which presents three phases (water, oil and gas) in the model.

In the undersaturated state, $S_g = 0$ and $p_b < p$. The density and viscosity in the oil phase depend both $p$ and $p_b$:

$$\begin{aligned}
&\rho_{Oo}(p, p_b) = \tfrac{\rho_{Os}}{B_{ob}(p_b)}(1 + c_o(p - p_b)),\\
&\rho_{Go}(p, p_b) = \tfrac{R_{so}(p_b)\rho_{Gs}}{B_{ob}(p_b)}(1 + c_o(p - p_b)), \quad \rho_g(p) = \tfrac{\rho_{Gs}}{B_g(p)},\\
&\mu_o(p, p_b) = \mu_{ob}(p_b)(1 + c_\mu(p - p_b)), \quad \mu_g = \mu_g(p).
\end{aligned} \tag{2.45}$$

In the saturated state, $S_g \neq 0$ and $p_b = p$; the densities and viscosities depend only on pressure $p$:

$$\begin{aligned}
&\rho_{Oo}(p) = \tfrac{\rho_{Os}}{B_{ob}(p)}, \quad \rho_{Go}(p) = \tfrac{R_{so}(p)\rho_{Gs}}{B_{ob}(p)}, \quad \rho_g(p) = \tfrac{\rho_{Gs}}{B_g(p)},\\
&\mu_o = \mu_o(p), \qquad \mu_g = \mu_g(p).
\end{aligned} \tag{2.46}$$

## 2.6 Numerical Solution of the Black Oil Model

We consider the case where the permeability tensor $\mathbf{k} = \mathrm{diag}(k_{11}, k_{22}, k_{33})$ is diagonal. Consequently, the transmissibility tensor $\mathbf{T}_\alpha$ is also diagonal, and we write $\mathbf{T}_\alpha = \mathrm{diag}(T_{\alpha 1}, T_{\alpha 2}, T_{\alpha 3})$, $\alpha = w, o, g$. Therefore, in terms of each coordinate direction, equations (2.24 - 2.26) are

$$\begin{aligned}
&\tfrac{\partial}{\partial t}\left(\tfrac{\phi S_w}{B_w}\right) = \tfrac{\partial}{\partial x_1}\left(T_{w1}\left(\tfrac{\partial p_w}{\partial x_1} - \gamma_w \tfrac{\partial z}{\partial x_1}\right)\right)\\
&+\tfrac{\partial}{\partial x_2}\left(T_{w2}\left(\tfrac{\partial p_w}{\partial x_2} - \gamma_w \tfrac{\partial z}{\partial x_2}\right)\right) + \tfrac{\partial}{\partial x_3}\left(T_{w3}\left(\tfrac{\partial p_w}{\partial x_3} - \gamma_w \tfrac{\partial z}{\partial x_3}\right)\right) + \tilde{q}_{Ws},
\end{aligned} \tag{2.47}$$

$$\begin{aligned}
&\frac{\partial}{\partial t}\left(\frac{\phi S_o}{B_o}\right) = \frac{\partial}{\partial x_1}\left(T_{o1}\left(\frac{\partial p_o}{\partial x_1} - \gamma_o \frac{\partial z}{\partial x_1}\right)\right) \\
&+\frac{\partial}{\partial x_2}\left(T_{o2}\left(\frac{\partial p_o}{\partial x_2} - \gamma_o \frac{\partial z}{\partial x_2}\right)\right) + \frac{\partial}{\partial x_3}\left(T_{o3}\left(\frac{\partial p_o}{\partial x_3} - \gamma_o \frac{\partial z}{\partial x_3}\right)\right) + \tilde{q}_{Os},
\end{aligned} \tag{2.48}$$

and

$$\begin{aligned}
&\frac{\partial}{\partial t}\left[\phi\left(\frac{S_g}{B_g} + \frac{R_{so}S_o}{B_o}\right)\right] = \frac{\partial}{\partial x_1}\left(T_{g1}\left(\frac{\partial p_g}{\partial x_1} - \gamma_g \frac{\partial z}{\partial x_1}\right)\right) \\
&+\frac{\partial}{\partial x_2}\left(T_{g2}\left(\frac{\partial p_g}{\partial x_2} - \gamma_g \frac{\partial z}{\partial x_2}\right)\right) + \frac{\partial}{\partial x_3}\left(T_{g3}\left(\frac{\partial p_g}{\partial x_3} - \gamma_g \frac{\partial z}{\partial x_3}\right)\right) \\
&+\frac{\partial}{\partial x_1}\left(R_{so}T_{o1}\left(\frac{\partial p_o}{\partial x_1} - \gamma_o \frac{\partial z}{\partial x_1}\right)\right) + \frac{\partial}{\partial x_2}\left(R_{so}T_{o2}\left(\frac{\partial p_o}{\partial x_2} - \gamma_o \frac{\partial z}{\partial x_2}\right)\right) \\
&+\frac{\partial}{\partial x_3}\left(R_{so}T_{o3}\left(\frac{\partial p_o}{\partial x_3} - \gamma_o \frac{\partial z}{\partial x_3}\right)\right) + \tilde{q}_{Gs},
\end{aligned} \tag{2.49}$$

where

$$\tilde{q}_{Ws} = \frac{q_{Ws}}{B_w}, \quad \tilde{q}_{Os} = \frac{q_{Os}}{B_o}, \quad \tilde{q}_{Gs} = \frac{q_{Gs}}{B_g} + \frac{q_{Os}R_{so}}{B_o}.$$

Numerical solutions of equations (2.47)–(2.49) will be obtained in this section.

### 2.6.1 Solution Techniques

The simultaneous solution [8] (SS) technique is the most direct way to solve the black oil system. It solves the three mass balance equations simultaneously, which means to solve for all the primary variables at the same time.

#### Saturated state

In the saturated state, the primary unknowns are $(p = p_o, S_w, S_o)$. To solve for the three unknowns, we need three equations, which are the finite difference counterparts

of equations (2.47)–(2.49):

$$
\begin{aligned}
&\tfrac{1}{\Delta t}\left(V\left[\left(\tfrac{\phi S_w}{B_w}\right)^{n+1}-\left(\tfrac{\phi S_w}{B_w}\right)^{n}\right]\right)_{i,j,k}\\
&=T^{n+1}_{w1,i+1/2,j,k}\left(p^{n+1}_{w,i+1,j,k}-p^{n+1}_{w,i,j,k}\right)-T^{n+1}_{w1,i-1/2,j,k}\left(p^{n+1}_{w,i,j,k}-p^{n+1}_{w,i-1,j,k}\right)\\
&+T^{n+1}_{w2,i,j+1/2,k}\left(p^{n+1}_{w,i,j+1,k}-p^{n+1}_{w,i,j,k}\right)-T^{n+1}_{w2,i,j-1/2,k}\left(p^{n+1}_{w,i,j,k}-p^{n+1}_{w,i,j-1,k}\right)\\
&+T^{n+1}_{w3,i,j,k+1/2}\left(p^{n+1}_{w,i,j,k+1}-p^{n+1}_{w,i,j,k}\right)-T^{n+1}_{w3,i,j,k-1/2}\left(p^{n+1}_{w,i,j,k}-p^{n+1}_{w,i,j,k-1}\right)\\
&-(T_w\gamma_w)^{n+1}_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)+(T_w\gamma_w)^{n+1}_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&-(T_w\gamma_w)^{n+1}_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)+(T_w\gamma_w)^{n+1}_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&-(T_w\gamma_w)^{n+1}_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)+(T_w\gamma_w)^{n+1}_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&+\tilde{Q}^{n+1}_{Ws,i,j,k},
\end{aligned}
\tag{2.50}
$$

$$
\begin{aligned}
&\tfrac{1}{\Delta t}\left(V\left[\left(\tfrac{\phi S_o}{B_o}\right)^{n+1}-\left(\tfrac{\phi S_o}{B_o}\right)^{n}\right]\right)_{i,j,k}\\
&=T^{n+1}_{o1,i+1/2,j,k}\left(p^{n+1}_{o,i+1,j,k}-p^{n+1}_{o,i,j,k}\right)-T^{n+1}_{o1,i-1/2,j,k}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i-1,j,k}\right)\\
&+T^{n+1}_{o2,i,j+1/2,k}\left(p^{n+1}_{o,i,j+1,k}-p^{n+1}_{o,i,j,k}\right)-T^{n+1}_{o2,i,j-1/2,k}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i,j-1,k}\right)\\
&+T^{n+1}_{o3,i,j,k+1/2}\left(p^{n+1}_{o,i,j,k+1}-p^{n+1}_{o,i,j,k}\right)-T^{n+1}_{o3,i,j,k-1/2}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i,j,k-1}\right)\\
&-(T_o\gamma_o)^{n+1}_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)+(T_o\gamma_o)^{n+1}_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&-(T_o\gamma_o)^{n+1}_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)+(T_o\gamma_o)^{n+1}_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&-(T_o\gamma_o)^{n+1}_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)+(T_o\gamma_o)^{n+1}_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&+\tilde{Q}^{n+1}_{Os,i,j,k},
\end{aligned}
\tag{2.51}
$$

and

$$
\begin{aligned}
&\tfrac{1}{\Delta t}\left(V\left[\left(\phi\left(\tfrac{S_g}{B_g}+\tfrac{R_{so}S_o}{B_o}\right)\right)^{n+1}-\left(\phi\left(\tfrac{S_g}{B_g}+\tfrac{R_{so}S_o}{B_o}\right)\right)^{n}\right]\right)_{i,j,k}\\
&=T^{n+1}_{g1,i+1/2,j,k}\left(p^{n+1}_{g,i+1,j,k}-p^{n+1}_{g,i,j,k}\right)-T^{n+1}_{g1,i-1/2,j,k}\left(p^{n+1}_{g,i,j,k}-p^{n+1}_{g,i-1,j,k}\right)\\
&+T^{n+1}_{g2,i,j+1/2,k}\left(p^{n+1}_{g,i,j+1,k}-p^{n+1}_{g,i,j,k}\right)-T^{n+1}_{g2,i,j-1/2,k}\left(p^{n+1}_{g,i,j,k}-p^{n+1}_{g,i,j-1,k}\right)\\
&+T^{n+1}_{g3,i,j,k+1/2}\left(p^{n+1}_{g,i,j,k+1}-p^{n+1}_{g,i,j,k}\right)-T^{n+1}_{g3,i,j,k-1/2}\left(p^{n+1}_{g,i,j,k}-p^{n+1}_{g,i,j,k-1}\right)\\
&-(T_g\gamma_g)^{n+1}_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)+(T_g\gamma_g)^{n+1}_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&-(T_g\gamma_g)^{n+1}_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)+(T_g\gamma_g)^{n+1}_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&-(T_g\gamma_g)^{n+1}_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)+(T_g\gamma_g)^{n+1}_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&+(R_{so}T_o)^{n+1}_{1,i+1/2,j,k}\left(p^{n+1}_{o,i+1,j,k}-p^{n+1}_{o,i,j,k}\right)-(R_{so}T_o)^{n+1}_{1,i-1/2,j,k}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i-1,j,k}\right)\\
&+(R_{so}T_o)^{n+1}_{2,i,j+1/2,k}\left(p^{n+1}_{o,i,j+1,k}-p^{n+1}_{o,i,j,k}\right)-(R_{so}T_o)^{n+1}_{2,i,j-1/2,k}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i,j-1,k}\right)\\
&+(R_{so}T_o)^{n+1}_{3,i,j,k+1/2}\left(p^{n+1}_{o,i,j,k+1}-p^{n+1}_{o,i,j,k}\right)-(R_{so}T_o)^{n+1}_{3,i,j,k-1/2}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i,j,k-1}\right)\\
&-(R_{so}T_o\gamma_o)^{n+1}_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)+(R_{so}T_o\gamma_o)^{n+1}_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&-(R_{so}T_o\gamma_o)^{n+1}_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)+(R_{so}T_o\gamma_o)^{n+1}_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&-(R_{so}T_o\gamma_o)^{n+1}_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)+(R_{so}T_o\gamma_o)^{n+1}_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&+\tilde{Q}^{n+1}_{Gs,i,j,k},
\end{aligned}
\tag{2.52}
$$

where $\tilde{Q}_{\alpha s,i,j,k}=(V\tilde{q}_{\alpha s})_{i,j,k}$, $\alpha=W,O,G$.

Equations (2.50)–(2.52) are nonlinear in terms of the primary unknowns ($p^{n+1}$, $S_w^{n+1}$, $S_o^{n+1}$). We need to use the Newton-Raphson iteration to linearize the model to

get a linear system. For each grid block, there are three unknowns $(p^{n+1}, S_w^{n+1}, S_o^{n+1})_{i,j,k}$. In each Newton-Raphson iteration, they can be represented by:

$$p^{n+1,l+1} = p^{n+1,l} + \delta p^{n+1,l+1}, \quad S_w^{n+1,l+1} = S_w^{n+1,l} + \delta S_w^{n+1,l+1},$$
$$S_o^{n+1,l+1} = S_o^{n+1,l} + \delta S_o^{n+1,l+1}.$$

The residuals of equations (2.50)–(2.52) at the Newton-Raphson iteration level $l$ are

$$\begin{aligned}
R^l_{w,i,j,k} &= \tfrac{1}{\Delta t}\left(V\left[\left(\tfrac{\phi S_w}{B_w}\right)^l - \left(\tfrac{\phi S_w}{B_w}\right)^n\right]\right)_{i,j,k} \\
&-T^l_{w1,i+1/2,j,k}\left(p^l_{w,i+1,j,k} - p^l_{w,i,j,k}\right) + T^l_{w1,i-1/2,j,k}\left(p^l_{w,i,j,k} - p^l_{w,i-1,j,k}\right) \\
&-T^l_{w2,i,j+1/2,k}\left(p^l_{w,i,j+1,k} - p^l_{w,i,j,k}\right) + T^l_{w2,i,j-1/2,k}\left(p^l_{w,i,j,k} - p^l_{w,i,j-1,k}\right) \\
&-T^l_{w3,i,j,k+1/2}\left(p^l_{w,i,j,k+1} - p^l_{w,i,j,k}\right) + T^l_{w3,i,j,k-1/2}\left(p^l_{w,i,j,k} - p^l_{w,i,j,k-1}\right) \\
&+(T_w\gamma_w)^l_{1,i+1/2,j,k}\left(z_{i+1,j,k} - z_{i,j,k}\right) - (T_w\gamma_w)^l_{1,i-1/2,j,k}\left(z_{i,j,k} - z_{i-1,j,k}\right) \\
&+(T_w\gamma_w)^l_{2,i,j+1/2,k}\left(z_{i,j+1,k} - z_{i,j,k}\right) - (T_w\gamma_w)^l_{2,i,j-1/2,k}\left(z_{i,j,k} - z_{i,j-1,k}\right) \\
&+(T_w\gamma_w)^l_{3,i,j,k+1/2}\left(z_{i,j,k+1} - z_{i,j,k}\right) - (T_w\gamma_w)^l_{3,i,j,k-1/2}\left(z_{i,j,k} - z_{i,j,k-1}\right) \\
&-\tilde{Q}^l_{Ws,i,j,k},
\end{aligned} \tag{2.53}$$

$$
\begin{aligned}
&R^l_{o,i,j,k} = \tfrac{1}{\Delta t}\left(V\left[\left(\tfrac{\phi S_o}{B_o}\right)^l - \left(\tfrac{\phi S_o}{B_o}\right)^n\right]\right)_{i,j,k} \\
&-T^l_{o1,i+1/2,j,k}\left(p^l_{o,i+1,j,k} - p^l_{o,i,j,k}\right) + T^l_{o1,i-1/2,j,k}\left(p^l_{o,i,j,k} - p^l_{o,i-1,j,k}\right) \\
&-T^l_{o2,i,j+1/2,k}\left(p^l_{o,i,j+1,k} - p^l_{o,i,j,k}\right) + T^l_{o2,i,j-1/2,k}\left(p^l_{o,i,j,k} - p^l_{o,i,j-1,k}\right) \\
&-T^l_{o3,i,j,k+1/2}\left(p^l_{o,i,j,k+1} - p^l_{o,i,j,k}\right) + T^l_{o3,i,j,k-1/2}\left(p^l_{o,i,j,k} - p^l_{o,i,j,k-1}\right) \\
&+(T_o\gamma_o)^l_{1,i+1/2,j,k}\left(z_{i+1,j,k} - z_{i,j,k}\right) - (T_o\gamma_o)^l_{1,i-1/2,j,k}\left(z_{i,j,k} - z_{i-1,j,k}\right) \\
&+(T_o\gamma_o)^l_{2,i,j+1/2,k}\left(z_{i,j+1,k} - z_{i,j,k}\right) - (T_o\gamma_o)^l_{2,i,j-1/2,k}\left(z_{i,j,k} - z_{i,j-1,k}\right) \\
&+(T_o\gamma_o)^l_{3,i,j,k+1/2}\left(z_{i,j,k+1} - z_{i,j,k}\right) - (T_o\gamma_o)^l_{3,i,j,k-1/2}\left(z_{i,j,k} - z_{i,j,k-1}\right) \\
&-\tilde{Q}^l_{Os,i,j,k},
\end{aligned}
\tag{2.54}
$$

and

$$
\begin{aligned}
&R^l_{g,i,j,k} = \tfrac{1}{\Delta t}\left(V\left[\left(\phi\left(\tfrac{S_g}{B_g}+\tfrac{R_{so}S_o}{B_o}\right)\right)^l-\left(\phi\left(\tfrac{S_g}{B_g}+\tfrac{R_{so}S_o}{B_o}\right)\right)^n\right]\right)_{i,j,k}\\
&-T^l_{g1,i+1/2,j,k}\left(p^l_{g,i+1,j,k}-p^l_{g,i,j,k}\right)+T^l_{g1,i-1/2,j,k}\left(p^l_{g,i,j,k}-p^l_{g,i-1,j,k}\right)\\
&-T^l_{g2,i,j+1/2,k}\left(p^l_{g,i,j+1,k}-p^l_{g,i,j,k}\right)+T^l_{g2,i,j-1/2,k}\left(p^l_{g,i,j,k}-p^l_{g,i,j-1,k}\right)\\
&-T^l_{g3,i,j,k+1/2}\left(p^l_{g,i,j,k+1}-p^l_{g,i,j,k}\right)+T^l_{g3,i,j,k-1/2}\left(p^l_{g,i,j,k}-p^l_{g,i,j,k-1}\right)\\
&+(T_g\gamma_g)^l_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)-(T_g\gamma_g)^l_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&+(T_g\gamma_g)^l_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)-(T_g\gamma_g)^l_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&+(T_g\gamma_g)^l_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)-(T_g\gamma_g)^l_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&-(R_{so}T_o)^l_{1,i+1/2,j,k}\left(p^l_{o,i+1,j,k}-p^l_{o,i,j,k}\right)+(R_{so}T_o)^l_{1,i-1/2,j,k}\left(p^l_{o,i,j,k}-p^l_{o,i-1,j,k}\right)\\
&-(R_{so}T_o)^l_{2,i,j+1/2,k}\left(p^l_{o,i,j+1,k}-p^l_{o,i,j,k}\right)+(R_{so}T_o)^l_{2,i,j-1/2,k}\left(p^l_{o,i,j,k}-p^l_{o,i,j-1,k}\right)\\
&-(R_{so}T_o)^l_{3,i,j,k+1/2}\left(p^l_{o,i,j,k+1}-p^l_{o,i,j,k}\right)+(R_{so}T_o)^l_{3,i,j,k-1/2}\left(p^l_{o,i,j,k}-p^l_{o,i,j,k-1}\right)\\
&+(R_{so}T_o\gamma_o)^l_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)-(R_{so}T_o\gamma_o)^l_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&+(R_{so}T_o\gamma_o)^l_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)-(R_{so}T_o\gamma_o)^l_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&+(R_{so}T_o\gamma_o)^l_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)-(R_{so}T_o\gamma_o)^l_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&-\tilde{Q}^l_{Gs,i,j,k}.
\end{aligned}
\tag{2.55}
$$

The unknown and residual vectors are defined as:

$$
\mathbf{y}=(p,S_w,S_o)^T,\quad \mathbf{R}^l_{i,j,k}=\left(R^l_{w,i,j,k},R^l_{o,i,j,k},R^l_{g,i,j,k}\right)^T,
$$

Using the Newton-Raphson method, equations (2.50)–(2.52) can lead to a linear system of equations in terms of $\delta \mathbf{y}^{l+1}$:

$$\begin{aligned}
&\frac{\partial \mathbf{R}^l_{i,j,k}}{\partial \mathbf{y}_{i,j,k-1}} \delta \mathbf{y}^{l+1}_{i,j,k-1} + \frac{\partial \mathbf{R}^l_{i,j,k}}{\partial \mathbf{y}_{i,j-1,k}} \delta \mathbf{y}^{l+1}_{i,j-1,k} + \frac{\partial \mathbf{R}^l_{i,j,k}}{\partial \mathbf{y}_{i-1,j,k}} \delta \mathbf{y}^{l+1}_{i-1,j,k} \\
&\quad + \frac{\partial \mathbf{R}^l_{i,j,k}}{\partial \mathbf{y}_{i,j,k}} \delta \mathbf{y}^{l+1}_{i,j,k} + \frac{\partial \mathbf{R}^l_{i,j,k}}{\partial \mathbf{y}_{i+1,j,k}} \delta \mathbf{y}^{l+1}_{i+1,j,k} + \frac{\partial \mathbf{R}^l_{i,j,k}}{\partial \mathbf{y}_{i,j+1,k}} \delta \mathbf{y}^{l+1}_{i,j+1,k} \\
&\quad \frac{\partial \mathbf{R}^l_{i,j,k}}{\partial \mathbf{y}_{i,j,k+1}} \delta \mathbf{y}^{l+1}_{i,j,k+1} = -\mathbf{R}^l_{i,j,k},
\end{aligned} \tag{2.56}$$

After that, we use a linear solver to solve the linear system to get the increment $\delta \mathbf{y}^{l+1}$ and then the solution is updated until a convergence criterion is achieved:

$$\mathbf{y}^{l+1} = \mathbf{y}^l + \delta \mathbf{y}^{l+1}.$$

**Undersaturated state**

In the undersaturated state, the primary unknowns are $(p = p_o, S_w, p_b)$. Similar equations can be obtained as in the saturated state:

$$
\begin{aligned}
&\tfrac{1}{\Delta t}\left(V\left[\left(\tfrac{\phi S_w}{B_w}\right)^{n+1}-\left(\tfrac{\phi S_w}{B_w}\right)^{n}\right]\right)_{i,j,k}\\
&=T^{n+1}_{w1,i+1/2,j,k}\left(p^{n+1}_{w,i+1,j,k}-p^{n+1}_{w,i,j,k}\right)-T^{n+1}_{w1,i-1/2,j,k}\left(p^{n+1}_{w,i,j,k}-p^{n+1}_{w,i-1,j,k}\right)\\
&+T^{n+1}_{w2,i,j+1/2,k}\left(p^{n+1}_{w,i,j+1,k}-p^{n+1}_{w,i,j,k}\right)-T^{n+1}_{w2,i,j-1/2,k}\left(p^{n+1}_{w,i,j,k}-p^{n+1}_{w,i,j-1,k}\right)\\
&+T^{n+1}_{w3,i,j,k+1/2}\left(p^{n+1}_{w,i,j,k+1}-p^{n+1}_{w,i,j,k}\right)-T^{n+1}_{w3,i,j,k-1/2}\left(p^{n+1}_{w,i,j,k}-p^{n+1}_{w,i,j,k-1}\right)\\
&-(T_w\gamma_w)^{n+1}_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)+(T_w\gamma_w)^{n+1}_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&-(T_w\gamma_w)^{n+1}_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)+(T_w\gamma_w)^{n+1}_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&-(T_w\gamma_w)^{n+1}_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)+(T_w\gamma_w)^{n+1}_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&+\tilde{Q}^{n+1}_{Ws,i,j,k},
\end{aligned}
\tag{2.57}
$$

$$
\begin{aligned}
&\tfrac{1}{\Delta t}\left(V\left[\left(\tfrac{\phi S_o}{B_o}\right)^{n+1}-\left(\tfrac{\phi S_o}{B_o}\right)^{n}\right]\right)_{i,j,k}\\
&=T^{n+1}_{o1,i+1/2,j,k}\left(p^{n+1}_{o,i+1,j,k}-p^{n+1}_{o,i,j,k}\right)-T^{n+1}_{o1,i-1/2,j,k}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i-1,j,k}\right)\\
&+T^{n+1}_{o2,i,j+1/2,k}\left(p^{n+1}_{o,i,j+1,k}-p^{n+1}_{o,i,j,k}\right)-T^{n+1}_{o2,i,j-1/2,k}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i,j-1,k}\right)\\
&+T^{n+1}_{o3,i,j,k+1/2}\left(p^{n+1}_{o,i,j,k+1}-p^{n+1}_{o,i,j,k}\right)-T^{n+1}_{o3,i,j,k-1/2}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i,j,k-1}\right)\\
&-(T_o\gamma_o)^{n+1}_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)+(T_o\gamma_o)^{n+1}_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&-(T_o\gamma_o)^{n+1}_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)+(T_o\gamma_o)^{n+1}_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&-(T_o\gamma_o)^{n+1}_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)+(T_o\gamma_o)^{n+1}_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&+\tilde{Q}^{n+1}_{Os,i,j,k},
\end{aligned}
\tag{2.58}
$$

and

$$
\begin{aligned}
&\tfrac{1}{\Delta t}\left(V\left[\left(\tfrac{\phi R_{so}S_o}{B_o}\right)^{n+1}-\left(\tfrac{\phi R_{so}S_o}{B_o}\right)^{n}\right]\right)_{i,j,k}\\
&=(R_{so}T_o)^{n+1}_{1,i+1/2,j,k}\left(p^{n+1}_{o,i+1,j,k}-p^{n+1}_{o,i,j,k}\right)-(R_{so}T_o)^{n+1}_{1,i-1/2,j,k}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i-1,j,k}\right)\\
&+(R_{so}T_o)^{n+1}_{2,i,j+1/2,k}\left(p^{n+1}_{o,i,j+1,k}-p^{n+1}_{o,i,j,k}\right)-(R_{so}T_o)^{n+1}_{2,i,j-1/2,k}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i,j-1,k}\right)\\
&+(R_{so}T_o)^{n+1}_{3,i,j,k+1/2}\left(p^{n+1}_{o,i,j,k+1}-p^{n+1}_{o,i,j,k}\right)-(R_{so}T_o)^{n+1}_{3,i,j,k-1/2}\left(p^{n+1}_{o,i,j,k}-p^{n+1}_{o,i,j,k-1}\right)\\
&-(R_{so}T_o\gamma_o)^{n+1}_{1,i+1/2,j,k}\left(z_{i+1,j,k}-z_{i,j,k}\right)+(R_{so}T_o\gamma_o)^{n+1}_{1,i-1/2,j,k}\left(z_{i,j,k}-z_{i-1,j,k}\right)\\
&-(R_{so}T_o\gamma_o)^{n+1}_{2,i,j+1/2,k}\left(z_{i,j+1,k}-z_{i,j,k}\right)+(R_{so}T_o\gamma_o)^{n+1}_{2,i,j-1/2,k}\left(z_{i,j,k}-z_{i,j-1,k}\right)\\
&-(R_{so}T_o\gamma_o)^{n+1}_{3,i,j,k+1/2}\left(z_{i,j,k+1}-z_{i,j,k}\right)+(R_{so}T_o\gamma_o)^{n+1}_{3,i,j,k-1/2}\left(z_{i,j,k}-z_{i,j,k-1}\right)\\
&+\left(\tfrac{Vq_{Os}R_{so}}{B_o}\right)^{n+1}_{i,j,k}.
\end{aligned}
\tag{2.59}
$$

Equations (2.57)–(2.59) are also nonlinear in terms of the primary unknowns $(p^{n+1}, S_w^{n+1}, p_b^{n+1})$. We need to use the Newton-Raphson iteration to linearize the model to get a linear system and solve it iteratively.

$$p^{n+1,l+1} = p^{n+1,l} + \delta p^{n+1,l+1}, \quad S_w^{n+1,l+1} = S_w^{n+1,l} + \delta S_w^{n+1,l+1},$$
$$p_b^{n+1,l+1} = p_b^{n+1,l} + \delta p_b^{n+1,l+1}.$$

The resulting linear system can be expressed in the increments $\delta p^{n+1,l+1}$, $\delta S_w^{n+1,l+1}$, and $\delta p_b^{n+1,l+1}$ as in the saturated case.

Now we get the linear system here which is usually called a Jacobi matrix. In the next chapter, we will talk about the main topic of this thesis, which is how to solve the Jacobi matrix efficiently.

# Chapter 3

# GPU COMPUTING

## 3.1 Introduction to CUDA

Compute Unified Device Architecture (CUDA), a parallel programming framework by NVIDIA, is a new hardware and software architecture for issuing and managing computations on GPU as a data-parallel computing device. For a CUDA programmer, the computing system includes two part: a host and a device. The host is simply the CPU, and the device is the GPU which has a large number of small scalar processors.

### 3.1.1 CUDA Program Structure

A CUDA application consists of two parts: host program and device program [11], which is shown in Figure 3.1. This host program is completely CPU sequential code running on CPU. The device program is executed on a GPU parallel device and often named the kernel function. A kernel function is a completely SPMD (Single Program Multiple Data) computational model, generating a large number of threads and executing the same kernel function on many scalar cores on the GPU card. The bunch of threads is organized into a grid of thread blocks. The threads in the same thread block are able to cooperate with each other by barrier synchronization and a share memory which is private for each block.

Figure 3.2 shows the architecture of a modern NVIDIA GPU card. A GPU card

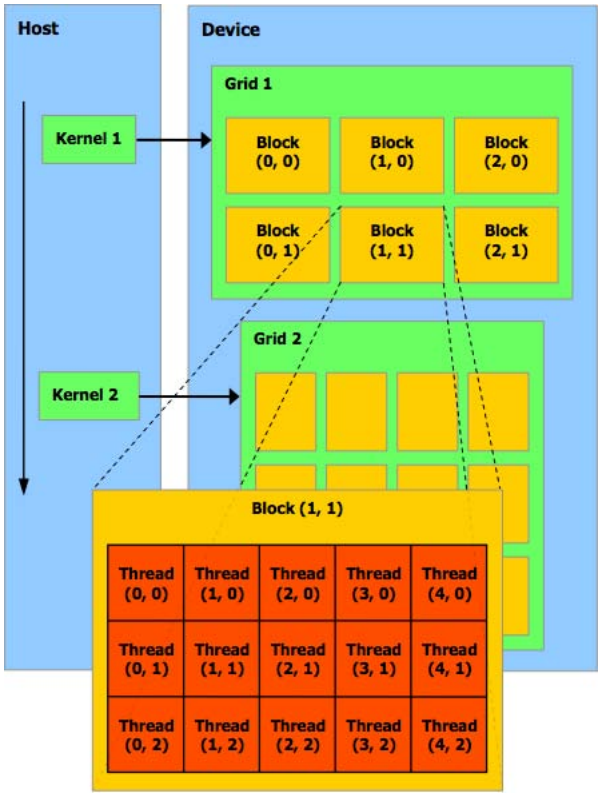


Figure 3.1: CUDA Proramming model [11].

consists of several SM multiprocessors, each of which can generate 1024 co-resident threads. Also, in each SM, there is a number of streaming processors(sp) which share the same control logic and instruction cache. NVIDIA Tesla C2050 based on NVIDIA's next generation CUDA architecture named Fermi can be equipped with 14 SM and each SM consists of 32 with the total of 448 streaming processors. In each SM, Integer and single precision floating point operations are performed by 16 sp, and the other 16 for double precision.

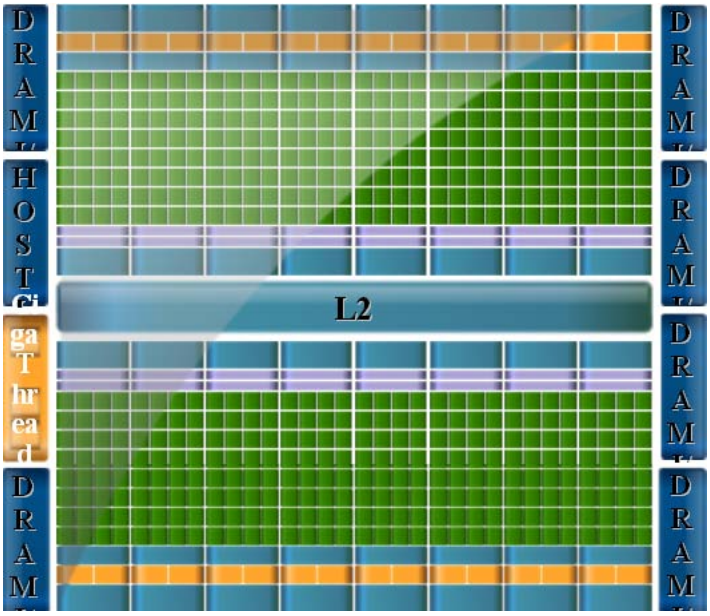


Figure 3.2: The Architecture of GPU [11].

### 3.1.2 Device Memories and Data Transfer

The memory structure of GPU is composed of off-ship memory including global memory, local memory, constant memory, and texture memory, and on-chip memory: share memory and registers. The most efficient way to access the global memory is to coalesce the simultaneous memory accesses by threads in a half-warp into a single memory transaction. Since it is on chip, the share memory is much faster than the global memory but we also need to pay attention to the problems of bank conflict.

### 3.1.3 Kernel Functions and Threading

Launching a CUDA kernel, a grid of threads will be generated in two level hierarchy. At the top level, a grid consists of a one or two dimensional thread blocks, and at the bottom level, each block consist of one, two or three dimensional threads. Each

thread block can only be executed in one multiprocessor. For the executing timing of threads, each block is divided into 32-thread units called warps. A warp executes a single instruction at a time across all its threads. Tthe SM can switch the warps, which can reduce the latency of memory accesses.

## 3.2 CUDA Memories

CUDA presents several types memory to achieve high performance computation. To highly utilize the great performance, the programmer should understand the memory hierarchy of GPU, which shows in Figure 3.3.

### 3.2.1 Global and Constant Memory

From Figure 3.3, we can find global memory and constant memory at the bottom. They can both be written and read using some API functions from a host program. For the global memory, the programmer needs to allocate memory in the global memory, and transfer the data from CPU Host to the allocated device memory. So, in some sense, the global memory is for storing the data we need process from CPU. For constant memory, it has read-only access and high-bandwidth and short-latency by the threads on device.

### 3.2.2 Registers and Shared Memory

Registers and shared memory are the on-chip memories, which can present high speed access with a highly parallel way. Registers belong to each individual threads and at the same time, each individual threads can only access its own registers. Share memory is the critical memory for GPU. Compared with registers, it belongs to each

thread block. All the threads inside the block can access share memory, which offer a communication or cooperation manner for the threads, and it is 100x faster than access of global memory; however, the amount of shared memory is quite limited. Because of its high access and communication effect for threads, how to fully utilize the share memory is quite sensitive for the speed-up for GPU.

## 3.3 Memory Coalescing

During load or store operations, the threads of a warp are free to use arbitrary addresses. If the memory we are accessing is closed, the access operation of all threads can be coalesced for great memory efficiency. In detail, the global memory of GPU is aligned by a sequence of 128-byte segments. Each time, memory is requested by 16 threads which is a half warp. The number of memory transactions is determined by the number of segments which are accessed by the 16 threads. For example, if the memory addresses being requested by the half warp are located in 2 segments, the number of transactions will be two. If the requested memory locates in the same segments, the numnber of transactions is just one, which is fully coalesced; otherwise, it is called uncoalesced. To achieve an efficiency memory accessing, we need try to optimize the memory access to fully coalesced [10].

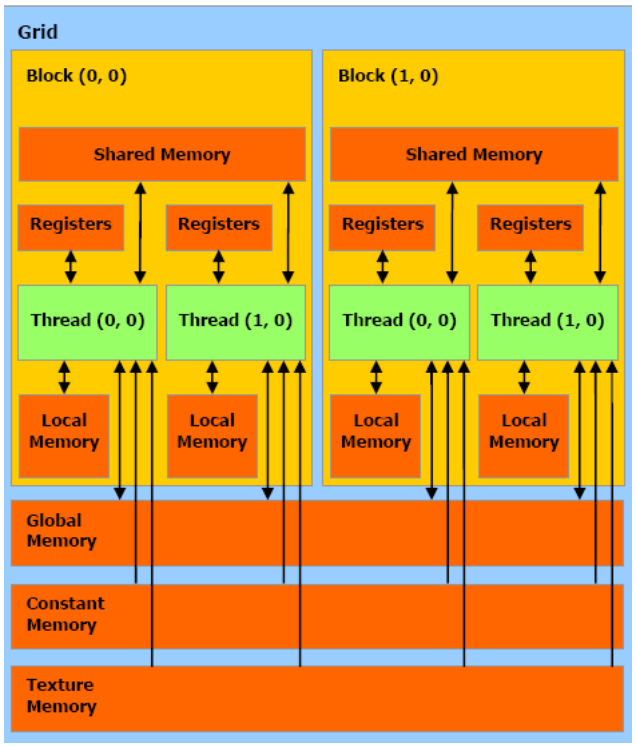


Figure 3.3: CUDA Memory Hierarchy[11].

# Chapter 4

# GPU-BASED PARALLEL SOLVER

## 4.1 PASS Overview

In Chapter 1, it has been emphasized that the linear solver is of the great importance in reservoir simulator development. To speed up the reservoir simulation, the direct way is to speed up the linear solver, which means to build a parallel linear solver package. In this thesis, a GPU-based linear solver package, named Parallel accelerated simulation solver (PASS) [31], is developed, which is a highly parallel linear solver package built on GPU architecture. In PASS, we create a new HEC matrix storage format to build our highly parallel BLAS and SPMV library. Based on those libraries, we assemble our linear solvers, including GMRES, BiCGSTAB, ORTHOMIN, ORTHODIR, GCR and CGS. Each solver has its own advantages and disadvantages on convergence, storage and computation cost. Without preconditioning, iterative solvers sometimes fail to converge. So we develop several different preconditioners, including Neumann polynomial, ILU(0), ILUT, Block ILU(0) and Block ILUT. Also, to get speed-up, we develop a parallel triangular solver for ILU preconditioners. Furthermore, we have several matrix pre-process methods like matrix scaling [2], matrix

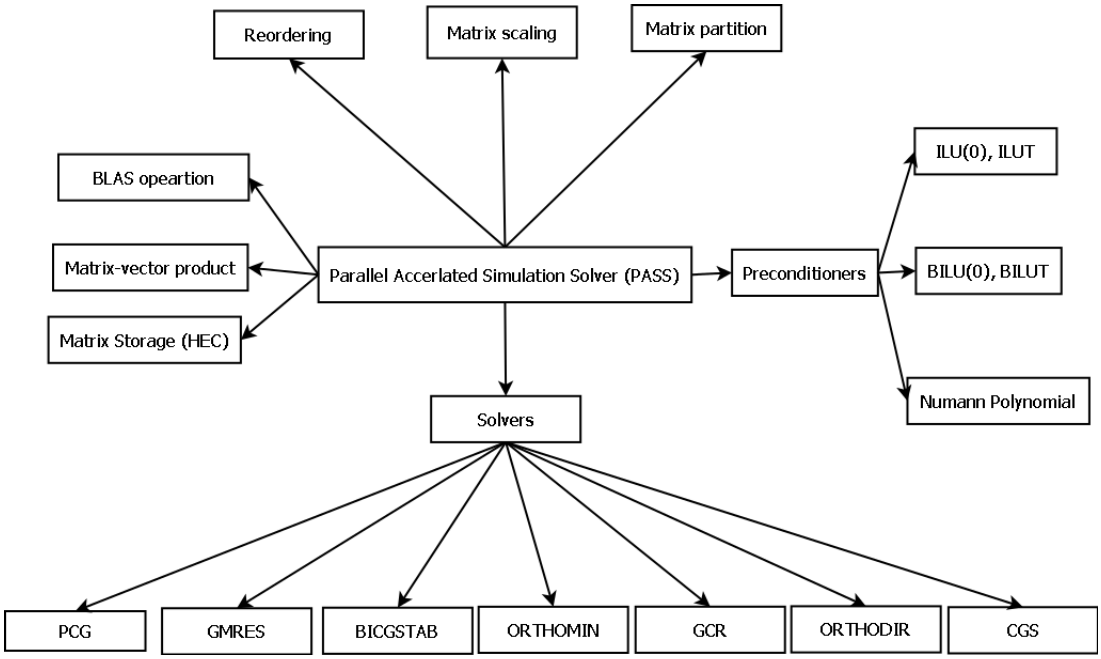


Figure 4.1: Parallel accelerated simulation solver(PASS).

partition [15] and reordering [23]. The architecture is described in Figure 4.1.

To develop a GPU-based parallel solver, we need first capture the main computational factors in the solver algorithm. From the algorithm point of view, which shows in Algorithm 2 , three computational factors can be easily captured: BLAS operations, Sparse matrix vector product, and preconditioning operation which means solving or getting an inverse of the preconditioner matrix. In other words, if those three operations can be highly parallel on GPU, we can definitely achieve a high performance GPU solver package.

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

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

Figure 4.2: COO format.

## 4.2 Sparse Matrix Storage Format

For a sparse matrix, to minimize the storage cost, only nonzero entries are usually stored, since most of entries of the sparse matrix are zeros. There are lots of different storage formats for a sparse matrix, which mainly differ from the sparsity pattern and the structure of the nonzero entries. To choose the most efficient format depends on the specific hardware architecture and the algorithms performed on it. In this section, several popular sparse matrix storage formats and their implementations for a sparse matrix vector product will be introduced.

### 4.2.1 COO

The coordinate (COO) format, described in Figure 4.2, is a simple method to store the sparse matrix. It stores three arrays which are row, col, and data, representing the row index, column index, and values of nonzero entries. This is a very general storage format and easy to understand. The storage cost is proportional to the number of nonzeros.

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

$$\begin{aligned} \mathtt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \mathtt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

Figure 4.3: CSR format.

### 4.2.2 CSR

The compressed row (CSR) format is the most popular and general method in linear solvers. Compared with the COO format, it compresses the row index to row index offsets. So, in CSR, we also have three arrays, ptr, indices, and data. Ptr stores the offset of first nonzero entry of each row. Indices stores the column index of each nonzero and data stores the nonzero values. This format is clearly described in Figure 4.3.

### 4.2.3 ELL

ELLPACK (ELL), as described in Figure 4.4, is also a common sparse matrix storage method, well-suited to vector architectures. This format actually consists of two matrices or arrays. One is the column indices and the other one stores nonzeros. For each row of the matrix, we use the same size to store. If the number of nozeros of a row is less than this size, then we use 0 to fill in.

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

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

Figure 4.4: ELL format.

### 4.2.4 HYB

The Hybrid format (HYB) is a combination of ELL and COO formats. The rows with the most frequent number of nonzeors are compressed and stored with the ELL format, and the rest is stored in the COO format.

### 4.2.5 HEC

The HEC format [22] is a new matrix storage format, invented in this work. It is a combination of ELL and CSR as shown in Figure 4.5. In HEC, the regular part of a given triangular matrix L is stored in ELL format and the irregular part in CSR part.

The main advantage of HEC is that ILU preconditioning can take advantage of this data structure to get speedup for triangular solvers. To transform the data structure of a given matrix to HEC, the row length of the ELL matrix can be calculated by Equation 4.1:

$$Find\ l\ (l \geq 0) such\ that w(i) = i * n + p_r * nz(i) is minimized, \tag{4.1}$$

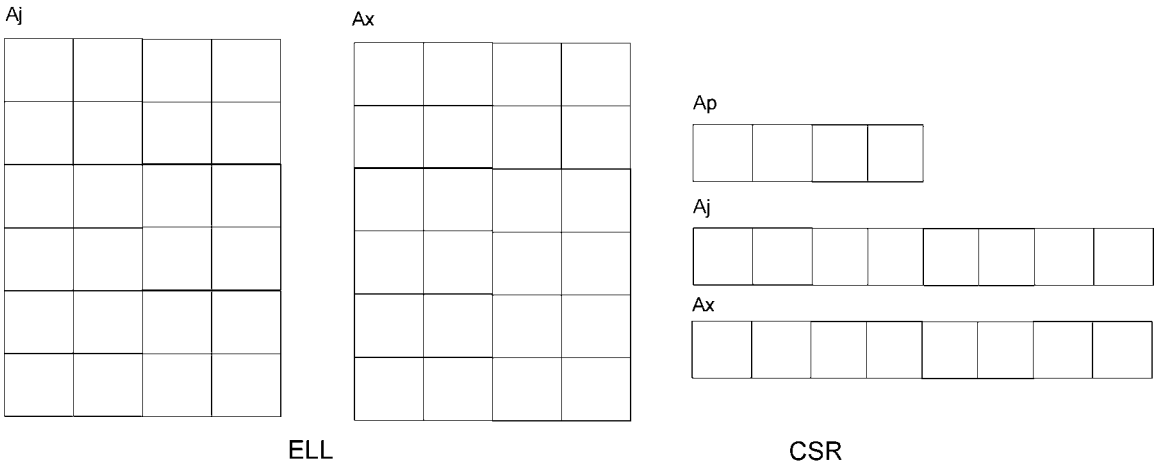


Figure 4.5: HEC format.

where $p_r$ is the relative performance of the ELL and CSR matrices and $nz(i)$ is the number of non-zeros in the CSR part when the row length of the ELL part is $i$. The minimum problem is easy to solve. Before we solve it, we should know $p_r$, which can be obtained by testing. A typical value for $p_r$ is 20 [5].

## 4.3 Sparse Matrix-Vector Multiplication

In the solver package, the SPMV for different matrix storage formats has been implemented. The implementation details can be found in [5]. Here, we only focus on the new storage format HEC.

Since the HEC matrix consists of two format matrices, ELL and CSR, the SPMV algorithm of HEC, $y = Ax$, can also be divided into two steps:

1. Calculate SPMV of the ELL matrix. Use each thread to calculate each row operation, which presents the great coalesced access because of the column-major order of ELL. The result is stored in array $y$.

2. Calculate the SPMV of the CSR matrix. Use each thread to calculate each row operation, and add the result to array $y$ to get the final result of the SPMV product.

**Algorithm 1** Sparse Matrix-Vector Multiplication, $y = Ax$

```
1: for i = 1: n do                    ▷ ELL, Use one GPU kernel to deal with this loop
2:     the i-th thread calculate the i-th row of ELL matrix;    ▷ Use one thread
3: end for
4:
5: for i = 1: n do                    ▷ CSR, Use one GPU kernel to deal with this loop
6:     the i-th thread calculate the i-th row of CSR matrix;    ▷ Use one thread
7: end for
```

For other BLAS operations in solver algorithms, like $y = \alpha Ax$, $y = \alpha Ax + \beta y$ and $z = \alpha Ax + \beta y$, the implementations are quite similar.

### 4.3.1 SPMV test

In this section, the numerical experiment for SPMV of different matrix storage formats is performed. The CPU used here is Intel Xeon X5570 and the GPUs are NVIDIA Tesla C2050. The operating system is Fedora 13 X86_64 with CUDA Toolkit 4.0 and GCC 4.4. All CPU codes are compiled with -O3 option. Testing matrices are listed in Table 4.1. Fourteen testing matrices are selected from Sparse Matrix Collection [12] and one SPE10 simulation matrix arising from an in-house black oil simulator is also tested here. The CSR(S) is a scalar algorithm for the CSR format matrix and the CSR(V) is a vector algorithm.

From Table 4.2, HYB and HEC achieve better average performance than other formats to deal with general sparse matrices. The HEC format has better speedup than HYB format in most cases, and also HEC has more advantages to increase the

| Matrix | N | Non-zero | NNZ/N |
|---|---|---|---|
| human_gene1 | 22283 | 12345963 | 554.1 |
| msc23052 | 23052 | 588933 | 25.5 |
| TSOPF_FS_b300_c2 | 56814 | 4391071 | 77.3 |
| cfd2 | 123440 | 1605669 | 13.0 |
| ESOC | 327062 | 6019939 | 18.4 |
| cage13 | 445315 | 7479343 | 16.8 |
| af_shell8 | 504855 | 9046865 | 17.9 |
| parabolic_fem | 525825 | 2100225 | 4.0 |
| Emilia_923 | 923136 | 20964171 | 22.7 |
| atmosmodd | 1270432 | 8814880 | 6.9 |
| Serena | 1391349 | 32961525 | 23.7 |
| Hook_1498.mtx | 1498023 | 30436237 | 5.9 |
| SPE10 | 2188851 | 29915573 | 13.7 |
| cage15 | 5154859 | 99199551 | 19.2 |

Table 4.1: Matrices used for testing

speedup of solvers which will be discussed in the preconditioner section.

## 4.4 Solvers

In this section, three main linear solvers will be introduced and the performance comparison of GPU implementations of these solvers will be presented in the numerical experiment.

### 4.4.1 GMRES

A linear system arising from the discretization of the governing equations in reservoir simulation has special properties, which is sparse, nonsymmetric and indefinite. The natural banded structure is usually spoiled by wells that perforate into many grid blocks or by irregular grid block structure. Dealing with these kinds of matrix, the Generalized Minimum Residual (GMRES) solver is widely used in the reservoir

| Matrix | CSR(S) | CSR(V) | ELL | HYB | HEC |
|---|---|---|---|---|---|
| human_gene1 | 0.76 | 4.85 | 0.84 | 2.68 | 1.13 |
| msc23052 | 0.71 | 3.63 | 1.55 | 2.29 | 2.50 |
| TSOPF_FS_b300_c2 | 0.79 | 8.62 | 5.42 | 5.41 | 8.96 |
| cfd2 | 1.41 | 2.26 | 8.06 | 6.86 | 11.61 |
| ESOC | 1.06 | 2.62 | 11.56 | 11.56 | 11.61 |
| cage13 | 1.25 | 2.36 | 7.56 | 9.42 | 11.04 |
| af_shell8 | 1.17 | 3.58 | 8.66 | 9.63 | 11.12 |
| parabolic_fem | 4.36 | 1.12 | 9.97 | 7.56 | 10.00 |
| Emilia_923 | 0.97 | 3.69 | 6.64 | 8.36 | 8.65 |
| atmosmodd | 2.94 | 1.38 | 14.54 | 14.50 | 14.57 |
| Serena | 0.98 | 3.58 | 1.79 | 7.26 | 7.29 |
| Hook_1498.mtx | 0.94 | 3.48 | 5.39 | 7.89 | 8.01 |
| SPE10 | 1.13 | 1.74 | 1.24 | 11.15 | 10.27 |
| cage15.mtx | 1.15 | 2.16 | 7.09 | 10.72 | 10.86 |

Table 4.2: Speedup of SPMV, Double Precision

simulation.

GMRES was invented by Saad and Schultiz [27] in 1986 to solve a large sparse non-symmetric system. GMRES generates a sequence of orthogonal vectors, and all previously computed vectors in the orthogonal sequence have to be retained. For this reason, "restarted" versions of this method are used. For restart GMRES (m), the Arnoldi process is applied to explicitly generate a basis of dimension m for the Krylov subspace $K_m(A, v) = span\{v, Av, A^2v, ..., A^{m-1}v\}$ for a given initial residual $v$. Given the Krylov basis, an intermediate solution $x_m$ is constructed as $x^{(i)} = x^0 + y_1v^{(1)} + \cdots + y_iv^{(i)}$, where the coefficients $y_k$ have been chosen to minimize the residual norm $\|b - Ax^{(i)}\|$, which is a least squares problem solved by Givens rotations introduced by Gentleman [13].

For GMRES, all the previously computed vectors need to be reserved, which makes the amount of work and storage required per iteration rise linearly with the iteration

count. The restarted GMRES (m) is often used to overcome this storage cost. The choice of $m$ is crucial to the efficiency of GMRES (m). If $m$ is too small, GMRES (m) tends to converge slowly and even fails to converge; on the contrary, with a large $m$, more computation work, time and storage are required. There is no specific rule for choosing the value of $m$ and it is more dependent on a problem itself [4].

In practice, GMRES converges after a small number of iterations when it is used in conjunction with a good preconditioner, which will be explained in the next section. The GMRES algorithm with preconditioning is listed in Algorithm 2.

**Algorithm 2** The Preconditioned GMRES(m) Method

1: $x^{(0)}$ is an initial guess
2: **for** $j = 1, 2, ...$ **do**
3: $r$ from $Mr = b - Ax^0$
4: $v_1 = r/\|r\|_2$
5: $s := \|r\|_2 e_1$
6: **for** $i = 1, 2, ..., m$ **do**
7: $w$ from $Mw = Av^{(i)}$
8: **for** $k = 1, ..., i$ **do**
9: $h_{k,i} = (w, v^{(k)})$
10: $w = w - h_{k,i}v^{(k)}$
11: **end for**
12: $h_{i+1,i} = \|w\|_2$
13: $v^{i+1} = w/h_{i+1,i}$
14: apply $J_1, ..., J_{i-1}$ on $(h_{1,i}, ..., h_{i+1,i})$
15: construct $J_i$ acting on $ith$ and $(i+1)$st component
16: of $h_{.,i}$ such that $(i+1)$st component of $j_i h_{.,i}$ is 0
17: $s := J_i s$
18: if $s(i+1)$ is small enough then (UPDATE(x,i) and quit)
19: **end for**
20: UPDATE(x, m)
21: **end for**

### 4.4.2 BICGSTAB

The BiConjugate Gradient Stabilized method (BiCGSTAB) was invented by Van der Vorst [28] to solve nonsymmetric linear systems. BiCGSTAB can be interpreted as the product of BiCG and repeatedly applied GMRES. At least locally, a residual vector is minimized, which leads to a considerably smoother convergence behaviour.

BiCGSTAB needs to calculate two matrix-vector products and four inner products per iteration. The algorithm of BiCGSTAB is described in Algorithm 3.

**Algorithm 3** The Preconditioned BiConjugate Gradient Stabilized Method

Compute $r^0 = b - Ax^0$ for some initial guess $x^0$
Choose $\overline{r}$ (for example,$\overline{r} = r^0$)
**for** $i = 1, 2, ...$ **do**
    $\rho_{i-1} = (\overline{r})^T r^{(i-1)}$
    **if then**$\rho_{(i-1)} = 0$ method fails
    **end if**
    **if then**$i = 1$
        $p^{(i)} = r^{(i-1)}$
    **else**
        $\beta_{i-1} = (\rho_{i-1}/\rho_{i-1})(\alpha_{i-1}/\omega_{i-1})$
        $p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1}\nu^{(i-1)})$
    **end if**
    Solve $Mp = p^{(i)}$
    $\nu^{(i)} = Ap$
    $\alpha_i = \rho_{(i-1)}/r^T\nu^{(i)}$
    $s = r^{(i-1)} - \alpha_i\nu^{(i)}$
    check norm of s; if small enough: set $x^{(i)} = x^{(i-1)} + \alpha_i p$ and stop
    solve $Ms = s$
    $t = As$
    $\omega_i = t^T s/t^T t$
    $x^{(i)} = x^{(i-1)} + \alpha_i p + \omega_i s$
    $r^{(i)} = s - \omega_i t$
    check convergence; continue if necessary
    for continuation it is necessary that $\omega_i \neq 0$
**end for**

### 4.4.3 ORTHOMIN

ORTHOMIN [29] was proposed for solving sparse matrices arising from reservoir simulation. This method utilizes orthogonalizations and minimizations to obtain a fast convergence rate. The convergence rate of ORTHOMIN is insensitive to iteration parameters and the transmissibility ratios. The algorithm is described in Algorithm 4.

**Algorithm 4** ORTHOMIN(k) Method

1: Compute $r_0 = b - Ax_0$. Set $p_0 = r_0$
2: **for** $j = 1, 2, ...$ until convergence **do**
3: $\quad \alpha_j = \frac{(r_j, Ap_j)}{(Ap_j, Ap_j)}$
4: $\quad x_{j+1} = x_j + \alpha_j p_j$
5: $\quad r_{j+1} = r_j - \alpha_j - \alpha_j Ap_j$
6: $\quad$ Compute $\beta_{ij} = -\frac{(Ar_{j+1}, Ap_i}{(Ap_i, Ap_i)}$, for $i = j - k + 1, ..., j$
7: $\quad p_{j+1} = r_{j+1} + \sum_{i=j-k+1}^{j} \beta_{ij} p_i$
8: **end for**

### 4.4.4 Solver Comparison

In this numerical experiment, the comparison of solvers with different preconditioners has been performed. The test environment is the same as the spmv test above. Testing matrices are two sparse matrices arising from different time steps of the SPE10 simulation problem. In Tables 4.5 and 4.6, they show the runtime on CPU and GPU as well as the speedup. From the comparison, we can found that the speedup for solvers is from 7.2 to 12.9. For the runtime, BiCGSSTAB achieves the best performance both on CPU and GPU.

| Matrix | N | NNZ | NNZ/Row |
|---|---|---|---|
| SPE10-1 | 2,188,851 | 29,915,573 | 13.7 |
| SPE10-2 | 2,188,851 | 29,915,573 | 13.7 |

Table 4.3: SPE10 matrix description

| Matrix | Relative tolerance | Restart m | Block size |
|---|---|---|---|
| SPE10-1 | 1E-3 | 40 | 8 |
| SPE10-2 | 1E-3 | 60 | 8 |

Table 4.4: solver parameter

## 4.5 Preconditoning

The convergence of an iterative linear solver depends highly on the condition number of the matrix needed to be solved. The higher condition number of a matrix, the more difficult to be solved for the matrix. Preconditioners can decrease the condition number and speed up the convergence. It transforms a linear system into the system with the same solution but a less condition number, for instance, from $Ax = b$ to $M^{-1}Ax = M^{-1}b$. In most cases of reservoir simulation, without preconditioning, an iterative solver usually fails to get converged, which leads to simulation failure.

Generally speaking, there are two criteria [14] to choose a good preconditioner.

1. $M^{-1}A$ is supposed to have a less condition number than the original matrix

| Solver | Preconditioner | CPU (s) | GPU (s) | Speedup |
|---|---|---|---|---|
| GMRES | ILU(0) | 263.8 | 27.9 | 9.5 |
| GMRES | BILU(0) | 307.8 | 27.5 | 11.2 |
| BiCGSTAB | ILU(0) | 84.3 | 11.7 | 7.2 |
| BiCGSTAB | BILU(0) | 85.6 | 9 | 9.5 |
| ORTHOMIN | ILU(0) | 284.8 | 30.1 | 9.5 |
| ORTHOMIN | BILU(0) | 283 | 27.6 | 10.3 |

Table 4.5: Performance comparison of solvers for SPE10-1

| Solver | Preconditioner | CPU (s) | GPU (s) | Speedup |
|---|---|---|---|---|
| GMRES | ILU(0) | 1125.1 | 123.1 | 9.1 |
| GMRES | BILU(0) | 1614 | 128.3 | 12.5 |
| BiCGSTAB | ILU(0) | 233.3 | 22.2 | 10.5 |
| BiCGSTAB | BILU(0) | 173.5 | 22 | 7.9 |
| ORTHOMIN | ILU(0) | 1216 | 94.3 | 12.9 |
| ORTHOMIN | BILU(0) | 787.7 | 84 | 9.4 |

Table 4.6: Performance comparison of solvers for SPE10-2

A, which means M should approximate A as closely as possible: $M \approx A$. Under this condition, the best choice may be $M = A$; however, it violates the second criteria.

2. $M^{-1}$ should be easy to compute, which means Mx = b can be easy to solve. Hence according to the characteristics of a precondtioning matrix M, we need have an efficient solver to deal with it.

Let us take left preconditoning as an example. For a sparse matrix system,

$$Ax = b, \tag{4.2}$$

it would be easier to compute:

$$M^{-1}Ax = M^{-1}b, \tag{4.3}$$

if $M^{-1}A$ has a better condition number. However, in the Krylov solvers, we need to calculate the matrix-vector product, which transforms the problem to:

$$w \leftarrow M^{-1}Av. \tag{4.4}$$

Here it is not efficient to calculate the inverse of M and also the matrix-matrix product.

Thus we separate the problem to the following steps:

$$v^* \leftarrow Av. \tag{4.5}$$

Then equation (4.4) transforms to:

$$w \leftarrow M^{-1}v^*, \tag{4.6}$$

which is identical to:

$$Mw = v^*. \tag{4.7}$$

In the following sections, we will discuss several precondtioners in the PASS solver package.

### 4.5.1 ILU

LU factorization is a well-known method to represent matrix A with L lower and U upper triangular matrices. However, for a large sparse $N$ by $N$ matrix, LU factorization requires high computational costs, and strict LU factorization for a sparse matrix will lead to a non-sparse matrix, which will both increase computational and storage costs. So LU factorization is not a appropriate method to deal with a large sparse matrix, especially in reservoir simulation. However, incomplete LU (ILU) is commonly used in solving large sparse matrix systems as a preconditioner coupled to iterative solvers.

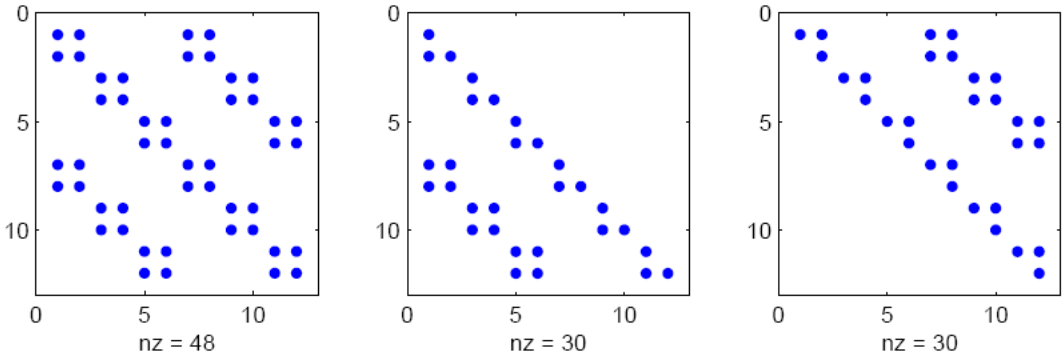


Figure 4.6: Distribution of non-zero elements in ILU(0) factorization.

**Pointwise ILU**

Incomplete LU (ILU) [26], as the name indicates, it is not a strict LU factorization. Some non-zero entries are ignored to reduce the computation and storage cost. Based on the level of fill-in, there are many different ILUs.

ILU(0) [26] is the simplest one in the ILU family. No fill-in will be kept in the LU process, so in the ILU(0) process, the lower and upper triangular matrices only keep nonzeros whose positions have non-zero elements in the original matrix. So the ILU(0) algorithm can be performed using the storage pattern of the original matrix, without requirement of significant extra memory. Figure 4.6 shows the ILU(0) patterns.

ILU(0) is simple and fast, and commonly used in a lot of reservoir simulators. But for some poorly condition problems, the approximation to the original matrix is not enough to help the iterative solvers to get converged. In order to enhance the accuracy, some ILUs with fill-in are adopted, for example, ILU with numerical threshold (ILUT) and ILU with fill-in level k (ILU(k)). The more fill-in, the more time the factorization takes. It is a trade-off between the accuracy and speed.

ILU(0) and ILUT are developed in the PASS solver package. The factorization

algorithms are showed as follows:

**Algorithm 5** ILU(0) factorization

1: **for** $i = 2, ..., n$ **do**
2: **for** $k = 1, ..., i-1$ and for $(i,k) \in NZ(A)$ **do**
3: Compute $a_{ik} = a_{ik}/a_{kk}$
4: **for** $j = k+1, ..., n$ and for $(i,j) \in NZ(A)$ **do**
5: Compute $a_{ij} = a_{ij} - a_{ik}a_{kj}$
6: **end for**
7: **end for**
8: **end for**

**Algorithm 6** ILUT factorization

1: **for** $i = 1, ..., n$ **do**
2: $w := a_{i*}$
3: **for** $k = 1, ..., i-1$ and when $w_k \neq 0$ **do**
4: $w_k = w_k/a_{kk}$
5: Apply a dropping rule to $w_k$
6: **if** $w_k \neq 0$ **then**
7: $w := w - w_k * u_{k*}$
8: **end if**
9: **end for**
10: Apply a dropping rule to row $w$
11: $l_{ij} := w_j$ for $j = 1, ..., i-1$
12: $u_{ij} := w_j$ for $j = i, ..., n$
13: $w := 0$
14: **end for**

In the factorization ILUT$(p, \tau)$ the following rule is used:

1. On line 5, an element $w_k$ is dropped (i.e., replaced by zero) if it is less than the relative tolerance $\tau_i$ obtained $\tau$ by multiplying by the original norm of the $i$-th row (e.g., the 2-norm);

2. On line 10, a dropping rule of a different type is applied. First, drop again any element in the row with a magnitude that is below the relative tolerance $\tau_i$. Then, keep only the $p$ largest elements in the $L$ part of the row and the largest elements in the $U$ part of the row in addition to the diagonal element, which is always kept.

**Block ILU**

Block ILU [26] is also developed to exploit the parallelism of GPU. As Figure 4.7 shows, The coefficient matrix is divided to many sub-matrices along the main diagonal and then locally decomposed using the ILU preconditioner. If the number of blocks increases, the better parallelism will be achieved. However, Block ILU is less effective than ILU in terms of convergence, since Blcok ILU only perform LU on diagonal block sub-matrix without capturing off-diagonal parts. In many cases, it need more iterations to converge than ILU. Generally, more blocks partitioned lead to better speedup, but lower convergence. It's still a trade-off between speedup and accuracy. The appropriate choice for BILU is using METIS [15] to partition the matrix.

### 4.5.2 Triangular solver

As mentioned in the fisrt section of this chapter, solving a preconditioned matrix is one of the major computational part. To speed up the linear solver, we need the highly parallel BLAS operation and SPMV as well as a precondtioner matrix solver, which is solving $Mx = b$. Hence for the ILU precondtioner matrix M, which is a combination of lower triangular L and upper triangular U, a parallel preconditioner solver is actually a parallel triangular solver. In this section, a parallel triangular solver [21] on GPU will be introduced.

For the lower triangular solver, $Lx = b$, based on the level schedule method [20],

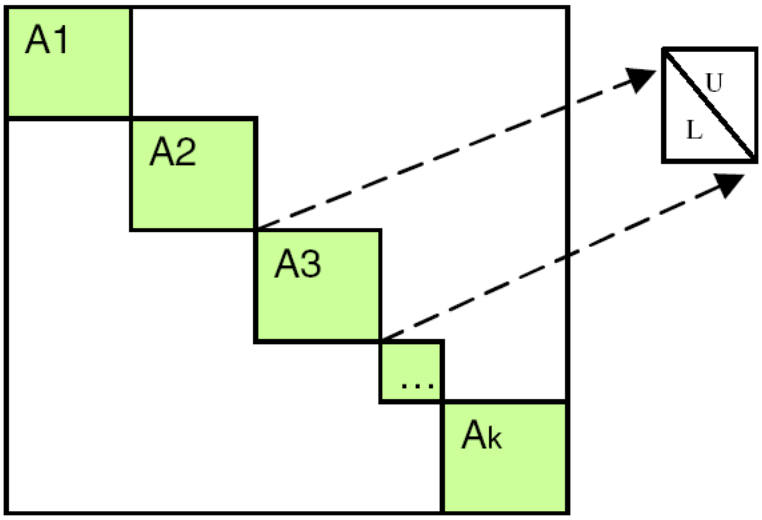


Figure 4.7: Block ILU factorization[].

the unknowns $x[i]$ are grouped to different levels, and the x[i] in the same level will be computed together on parallel. For the lower triangular problem, the level of $x(i)$ is calculated as

$$l(i) = 1 + \max_j l(j) \quad for\ all\ j\ such\ that\ L_{ij} \neq 0, i = 1, 2, \ldots, n, \tag{4.8}$$

where $L_{ij}$ is the $(i, j)$th entry of $L$, $l(i)$ is zero initially and $n$ is the number of rows.

We define $S_i = \{x(j) : l(j) = i\}$, which is the union of all unknowns that have the same level. Here we assume that each set $S_i$ is sorted in the ascending order according to the indices of the unknowns belonging to $S_i$. Define $N_i$ the number of unknowns in set $S_i$ and $nlev$ the number of levels. Now, a map $m(i)$ can be defined as follows:

$$m(i) = \sum_{j=1}^{k-1} N_j + p_k(x(i)), \quad x(i) \in S_k, \tag{4.9}$$

where $p_k(x(i))$ is the position $x(i)$ in the set $S_k$ when $x(i)$ belongs to $S_k$. With the help of the map, $m(i)$, we can reorder the triangular matrix $L$ to $L'$, where $L_{ij}$ in $L$ is transformed to $L'_{m(i)m(j)}$ in $L'$. $L'$ is still a lower triangular matrix. From this map, we find that if $x(i)$ is next to $x(j)$ in some set $S_k$, the $i$th and $j$th rows of $L$ are next to each other in $L'$ after reordering. It means that $L$ is reordered level by level, which implies that memory access in matrix $L'$ is less irregular than that in matrix $L$. Therefore, $L'$ has higher performance compared to $L$ when we solve a lower triangular problem.

The solving algorithm is divided into two steps. The first step is preprocessing and the second one is the solution step:

1. For preprocessing step, which is described in Algorithm 7, the level of each unknown is evaluated first. With those levels, using equation (4.9), a map from $L$ to $L'$ is calculated. Then the matrix $L$ is reordered. Here, $L$ can be stored by any matrix format. A general format is CSR. Finally, $L'$ is converted to the HEC format and each row of the CSR part has at least one element.

**Algorithm 7** Preprocessing lower triangular problem

1: Calculating the level of each unknown $x(i)$ using equation 4.8
2: Calculating the map $m(i)$ using equation 4.9.
3: Reordering matrix $L$ to $L'$ using map $m(i)$.
4: Converting $L'$ to the HEC format.

For the second step, it is to solve the lower triangular problem. In the algorithm, $level(i)$ is the starting row position of level $i$. First of all, the right-hand side $b$ is permutated according to the map $m(i)$ shown in the previous step. Then the unknowns in the same level is solved together, and the whole problem are solved level

**Algorithm 8** Parallel the lower triangular solver on GPU, $Lx = b$

```
1: for i = 1: n do                          ▷ Use one GPU kernel to deal with this loop
2:     b'(m(i)) = b(i);
3: end for
4: for i = 1 : nlev do                                              ▷ L'x' = b'
5:     start = level(i);
6:     end = level(i + 1) - 1;
7:     for j = start: end do                ▷ Use one GPU kernel to deal with this loop
8:         solve the jth row;
9:     end for
10: end for
11: for i = 1: n do                         ▷ Use one GPU kernel to deal with this loop
12:     x(i) = x'(m(i));
13: end for
```

by level. Each thread is responsible for one row. The final solution is achieved by a permutation.

In order to solve the upper triangular problem $Ux = b$, a transferring map is introduced:

$$t(i) = n - i,$$

where $n$ is the number of rows in $U$. With this map, the upper triangular problem is transferred to a lower triangular problem, and the lower triangular solver is called to solve the problem.

When we convert a given matrix to the HEC format, the row length of the ELL matrix is computed first, which is obtained by solving a minimum problem [6]:

$$Find\ l\ (l \geq 0)\ such\ that\ w(i) = i * n + p_r * nz(i)\ is\ minimized, \tag{4.10}$$

where $p_r$ is the relative performance of the ELL and CSR matrices and $nz(i)$ is the number of non-zeros in the CSR part when the row length of the ELL part is $i$. The minimum problem is easy to solve. Before we solve it, we should know $p_r$, which can be obtained by testing. A typical value for $p_r$ is 20 [6]. If the number of levels is too large, the row length of ELL should be reduced.

## 4.6 ILU test

In this test, we investigate the speedup of the GPU-based GMRES solver with ILU preconditioners. In particular, ILU(0), ILUT, block ILU(0) and block ILUT with a different number of blocks are studied.

| Blocks | Iterations | CPU (s) | GPU (s) | Speedup |
|---|---|---|---|---|
| 1 | 21 | 121.1 | 15 | 8.14 |
| 4 | 23 | 124.33 | 15 | 8.27 |
| 8 | 23 | 126.4 | 15.32 | 8.23 |
| 16 | 29 | 180.06 | 19.05 | 9.44 |

Table 4.7: Performance of SPE10

As shown in Table 4.7, for GMRES with ILU(0), speedup of above 8 is achieved. With an increase of the number of blocks for block ILU(0), the speedup will generally increase, because of parallelism exploied for block ILU; however, more iterations are needed to obtain convergence.

For ILUT and block ILUT, speedup of around 4 is obtained. Also, like ILU(0) and ILUT, with an increase of blocks, the speedup will increase. Compared with ILU(0), ILUT takes much less time and fewer iteration numbers, especially for CPU solver.

| Blocks | Iterations | CPU (s) | GPU (s) | Speedup |
|---|---|---|---|---|
| 1 | 5 | 34.2 | 11.7 | 2.92 |
| 4 | 7 | 44.67 | 10.35 | 4.3 |
| 8 | 7 | 45.78 | 9.58 | 4.76 |
| 16 | 10 | 63.13 | 12.43 | 5.07 |

Table 4.8: Performance of SPE10

# Chapter 5

# GPU-BASED RESERVOIR SIMULATION

In the previous section, the test problems are limited to the matrices arising from simulation. In this section, the simulation test will be presented to test the speedup of the solver package to the whole simulation process.

## 5.1 SPE10 Problem

The SPE Comparative Solution Projects present some benchmark cases to test the accuracy and efficiency of reservoir simulators. There are ten SPE problems in these projects. Among them, the SPE10 problem is the most fined-grid, highly heterogeneous model, which can be referred to as a benchmark to test the performance of a parallel reservoir simulator because of its expensive computational time. SPE10 problem is a 3D waterflood with a 1.1-million-cell geostatistical model [9], which is large enough that it is difficult to run the full model in the sequential simulators.

The SPE10 problem is described on a regular Cartesian grid. The dimensions of this model are (1200 x 2200 x 170) ft. "The top 70 ft (35 layers) represent the Tarbert formation, and the bottom 100 ft (50 layers) represent Upper Ness" [9]. The numerical grid block size is (20 10 2)ft, and hence the model has 60 x 220 x 85 blocks.

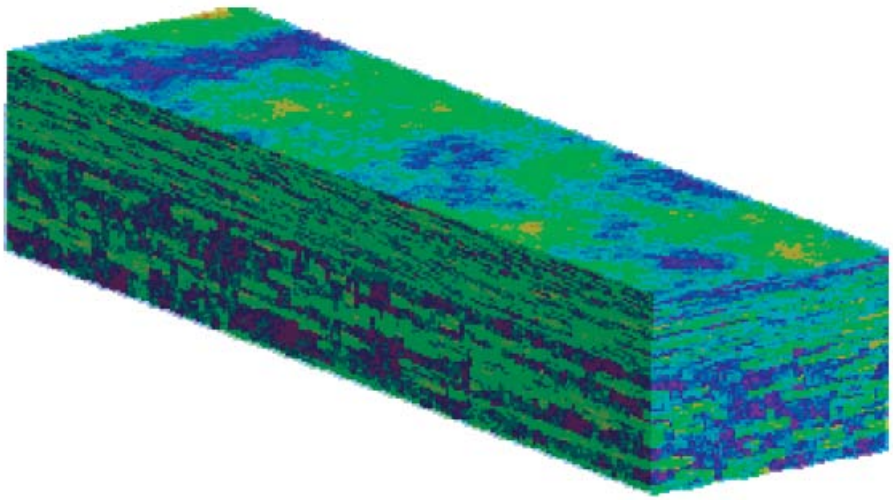

Figure 5.1: SPE10 porosity distribution.

The distribution of porosity and permeability are shown in Figure 5.1 and 5.2. The porosity and permeability distributions data can be obtained from the SPE10 project website. (http://www.spe.org/web/csp/)

For the well completion, there are four vertical producers at the corners and one vertical injectior in the center. The injector keeps a constant water injection rate of 5000 stb/day and the producers are operated at a bottom hole pressure of 4000 psi.

## 5.2 GPU-based Simulation

The concept of GPU-based parallel reservoir simulation here is simply to replace a traditional CPU solver by the new GPU-based one in the reservoir simulator. The workflow of the whole process is described in Figure 5.3 and 5.4. In each Newton iteration, the reservoir simulator generates the Jacobi matrix and transfers to the GPU, and then GPU solves the matrix on hundreds of processors in parallel and

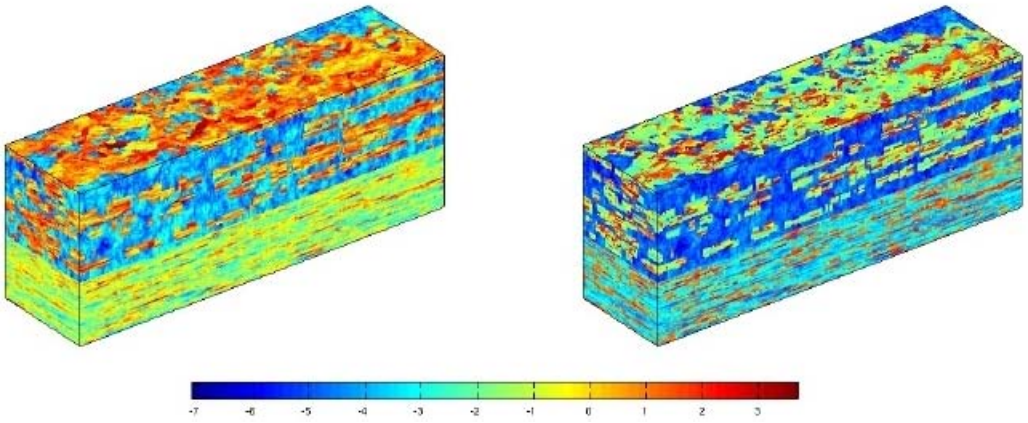

Figure 5.2: Logarithm of horizontal and vertical permeability seen from below.

transfer the solution back to the simulator in CPU.

## 5.3 Simulation Test

In this section, the speedup test of the whole simulation process is presented. As mentioned above, the PASS GPU solver package is coupled with an in-house black oil simulator to see the speedup of the SPE10 black oil simulation improved by the GPU. The experiment platform is listed in Table 5.1. The parameters of solvers are set in the Table 5.2. Three solvers, GMRES, BICGSTAB, and ORTHOMIN, are tested, coupling with ILU(0) and BILU(0) preconditioners.

For all cases, matrix scaling, reordering and METIS partition are applied to pre-process the Jacobi matrix. Among those combinations of solvers and precondtioners, GMRES with ILU(0) is commonly used in a lot of reservoir simulators, because of its general performance for different problems. In PASS, GMRES with ILU(0) can achieve a speedup of 5.2x. And the GPU only spends 17 minutes, which is the best performance compared with other combinations. For GMRES with BILU(0), it

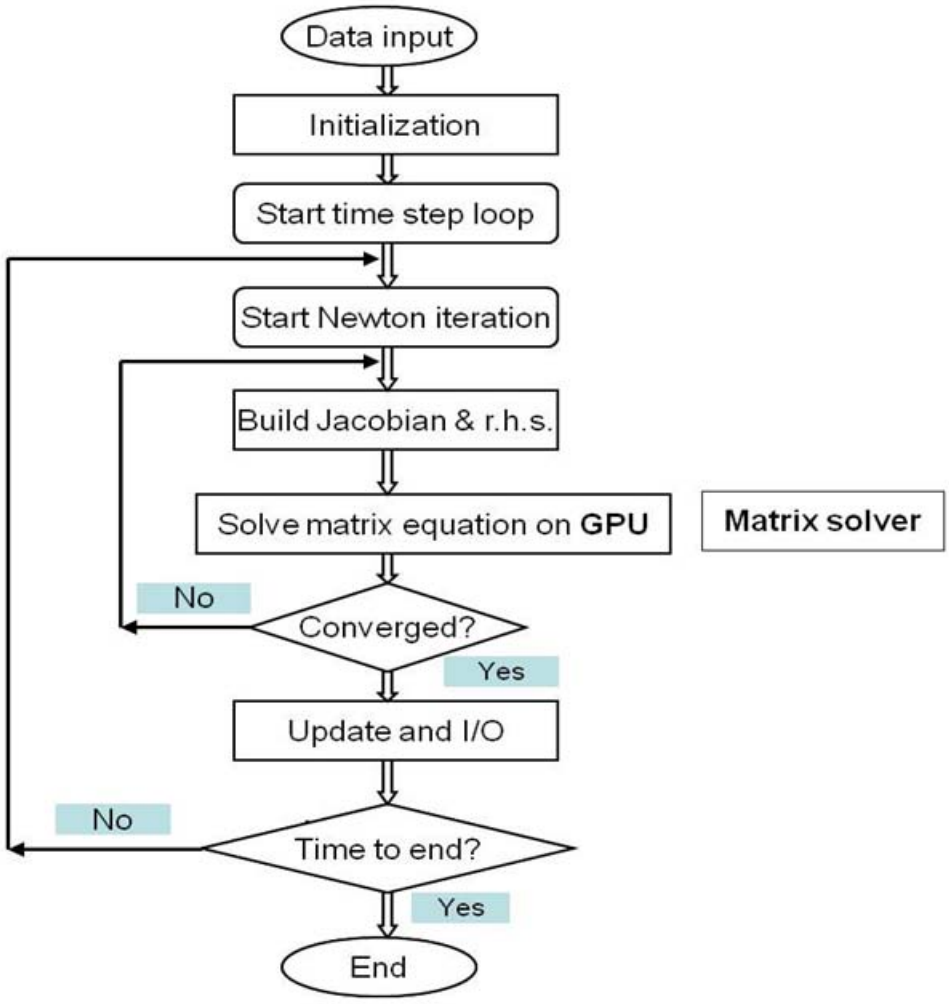


Figure 5.3: Workflow of GPU-based simulation.

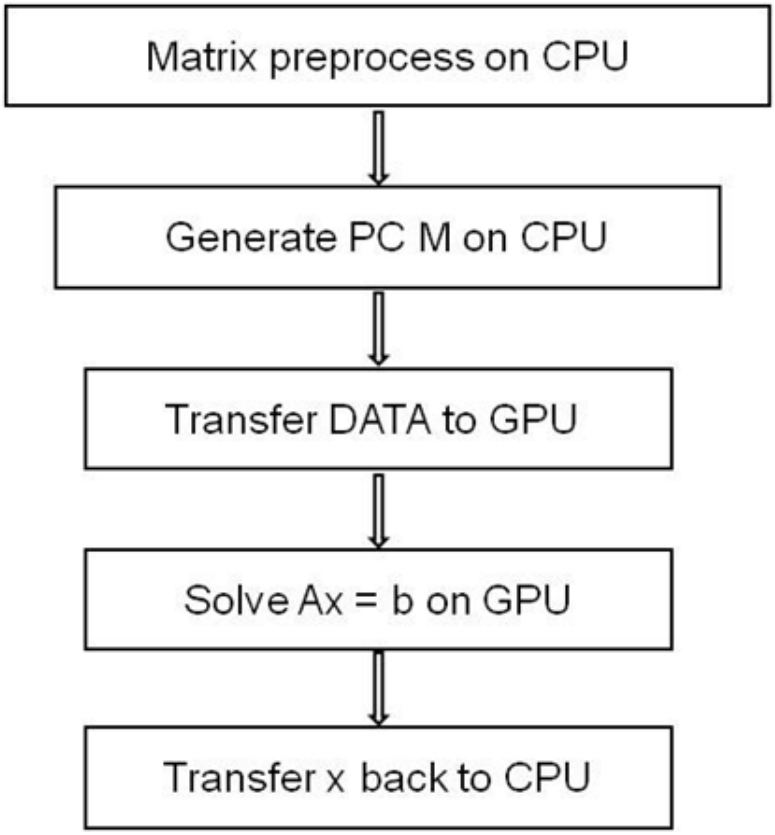


Figure 5.4: Workflow of GPU solver.

achieves the best speedup of 7.7x due to utilizing the block parallelism, but longer runtime because of the less accurate preconditioning. For BICGSTAB, CPU achieves the best runtime performance. Since the solver time of BICGSTAB for this problem is of low proportion in the whole simulation time, even high speedup of the linear solver cannot get a high speedup of the whole simulation, which lead to a speedup of nearly 2. For ORTHOMIN, the speedup of ILU(0) and BILU(0) is almost the same, which is 3.8x.

| | |
|---|---|
| CPU | Intel Xeon X5570, 8M cache, 2.93GHz, 32G memory |
| GPU | NVIDIA Tesla C2050, 3G memory |
| OS | Linux Fedora 13 x8664 kernel 2.6.34.761 |
| Compiler | GCC 4.4.5, -O3, CUDA Toolkit 3.2 |

Table 5.1: Experimental platform

| | |
|---|---|
| Relative tolerance | 1e-3 |
| Reatart m | 70 |
| METIS partition | 8 |

Table 5.2: Sovler parameters

| Solver | Preconditioner | CPU (s) | GPU (s) | Speedup |
|---|---|---|---|---|
| GMRES | ILU(0) | 1h30m16s | 17m18s | 5.2 |
| GMRES | BILU(0) | 2h37m02s | 20m18s | 7.7 |
| BICGSTAB | ILU(0) | 1h0m40s | 31m42s | 1.9 |
| BICGSTAB | BILU(0) | 1h7m22s | 34m28s | 1.95 |
| ORTHOMIN | ILU(0) | 2h25m48s | 37m58s | 3.8 |
| ORTHOMIN | BILU(0) | 2h37m23s | 41m22s | 3.8 |

Table 5.3: Performance of PASS for SPE10 problem

# Chapter 6

# CONCLUSIONS AND FUTURE WORK

## 6.1 Conclusions

1. A GPU-based linear solver package named Parallel Accelerated Simulation Solver (PASS) has been developed. This solver package has been coupled with a black oil simulator to speed up the reservoir simulation. With this solver package, reservoir simulation with millions of grid blocks can be highly sped up for desktop users.

2. Parallel Accelerated Simulation Solver (PASS) is a general linear solver package. It can solve the matrix systems arising from reservoir simulation with structured or unstructured grids.

3. In PASS, a highly parallel BLAS and SPMV library for different matrix storage formats are implemented, including CSR, ELL, HYB and HEC. Among those matrix storage formats, HEC is a new format developed in this thesis, which achieves the best speedup performance for BLAS and SPMV operations. Also, the HEC format is beneficial to the development of parallel triangular solvers on GPU.

4. For linear solvers, several different linear solvers are developed in this package, including GMRES, BICGSTAB, ORTHOMIN, CGS, ORTHODIR, GCR and PCG

with implementation on both CPU and GPU. In this thesis, we introduced the commonly linear solvers used in reservoir simulators, which are GMRES, BICGSTAB and ORTHOMIN. The speedup comparison has been performed in the numerical experiments of solving the SPE10 matrix. The speedup for this SPE matrix is from 7.9x to 12.9x.

5. Iterative solvers cannot solve a lot of problems without preconditioners. In this thesis, the implementation of ILU preconditioners including ILU(0), ILUT and their block versions has been studied. The factorization of ILU(0), ILUT and their block version have been developed on CPU, and the highly parallel triangular solvers on GPU for solving the ILU preconditioner matrix has been designed, which is critical factor for the speedup of the linear solvers.

7. The PASS solver package has been coupled with an in-house black oil simulator. In the GPU simulation test, the whole simulation test has been performed for the SPE10 problem. The speedup of different solvers and preconditoners has been presented, which clearly proved the great performance of the application of GPU in the parallel reservoir simulation.

## 6.2 Future Work

To some extent, The GPU solver package developed is sophisticated enough to be used for commercial reservoir simulator or any research purpose. However, the solver package still has several areas for further study:

1. To couple PASS to the thermal or compositional reservoir simulator to test the efficiency.

2. To develop two-stage precondtioners like Constant Pressure Residues (CPR)

on GPU.

3. To apply GPU to Jacobi assembly and fluid property calculations to get fully speedup for the whole simulation.

4. To develop an algorithm of adaptive solver selections.

# Bibliography

[1] G. Amador and A. Gomes, *Cuda-based linear solvers for stable fluids*, Information Science and Applications (ICISA), 2010 International Conference on, april 2010, pp. 1 –8.

[2] Patrick R. Amestoy, Iain S. Duff, Daniel Ruiz, and Bora Uçar, *High performance computing for computational science - vecpar 2008*, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 301–313.

[3] John R. Appleyard and Jeremy D. Appleyard, *Accelerating reservoir simulators using gpu technology*, the SPE Reservoir Simulation Symposium, Feb 2011.

[4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods, 2nd edition*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1994.

[5] Nathan Bell and Michael Garland, *Efficient sparse matrix-vector multiplication on CUDA*, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[6] Nathan Bell and Michael Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, SC '09: Proceedings of the Conference

on High Performance Computing Networking, Storage and Analysis (New York, NY, USA), ACM, 2009, pp. 1–11.

[7] Zhangxin Chen, Guanren Huan, and Yuanle Ma, *Computational methods for multiphase flows in porous media (computational science and engineering 2)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[8] Zhangxing Chen, *Reservoir simulation course notes*, University of Calgary, 2010.

[9] M.A. Christie, Heriot-Watt U., and M.J. Blunt, *Tenth spe comparative solution project: A comparison of upscaling techniques*, SPE Reservoir Simulation Symposium, Feb 2001.

[10] NVIDIA Corporation, *Cuda c best practices guide*, 2010.

[11] ———, *Cuda programming guide*, NVIDIA Corporation, 2011.

[12] Timothy A. Davis, *University of florida sparse matrix collection*, Tech. report, 1994.

[13] W.M. Gentleman, *Least squares computations by givens transformations without square roots*, Journal of Institute of Mathematics and its Applications (1973).

[14] Yuanlin Jiang, *Techniques for modeling complex reservoirs and advanced wells*, Ph.D. thesis, Stanford University, 2007.

[15] George Karypis and Vipin Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput. **20** (1998), no. 1, 359–392.

[16] Yoichiro Kawaguchi, Takahiro Harada, and Seiichi Koshizuka, *Smoothed particle hydrodynamics on gpus*, Tech. report, 2007.

[17] Hector Klie, Hari Suda, Ruipeng Li, and Yousef Saad, *Exploiting capabilities of many core platforms in reservoir simulation*, the SPE Reservoir Simulation Symposium, Feb 2011.

[18] C. Lauterbach, M. Garl, S. Sengupta, D. Luebke, and D. Manocha, *Fast bvh construction on gpus*, In Proc. Eurographics 09, 2009.

[19] M.C. Leverett and W.B. Lewis, *Steady flow of gas-oil-water mistures through unconsolidated sands*, Trans. SPE AIME **142** (1941), 107–116.

[20] R. Li and Y. Saad, *Gpu-accelerated preconditioned iterative linear solvers*, Tech. Report umsi-2010-112, 2010.

[21] Hui Liu, Song Yu, Zhangxing Chen, Ben Hsieh, and Lei Shao, *Parallel preconditioners for reservoir simulation on gpu*, SPE Latin American and Caribbean Petroleum Engineering Conference, April 2012.

[22] ———, *Sparse matrix-vector multiplication on nvidia gpu*, IJNAMB **3** (2012), no. 2, 185–191.

[23] J. U. Mallya, S. E. Zitney, S. Choudhary, and Mark A. Stadtherr, *Matrix reordering effects on a parallel frontal solver for large scale process simulation*, Computers and Chemical Engineering **23** (1998), 585–593.

[24] Matthias Müller, David Charypar, and Markus Gross, *Particle-based fluid simulation for interactive applications*, Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation (Aire-la-Ville, Switzerland, Switzerland), SCA '03, Eurographics Association, 2003, pp. 154–159.

[25] Y. Saad, *ILUT: a dual threshold incomplete ILU factorization*, Numerical Linear Algebra with Applications **1** (1994), 387–402.

[26] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[27] Youcef Saad and Martin H Schultz, *Gmres: a generalized minimal residual al-*

*gorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput. **7** (1986), no. 3, 856–869.

[28] 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. Stat. Comput. **13** (1992), no. 2, 631–644.

[29] P.K.W. Vinsome, *Orthomin, an iterative method for solving sparse sets of simultaneous linear equations*, SPE Symposium on Numerical Simulation of Reservoir Performance, Feb 1976.

[30] Mark Wakefield, *The use of gpus for reservoir simulation*, Tech. report, Schlumberger, 2011.

[31] Song Yu, Hui Liu, Zhangxing Chen, Ben Hsieh, and Lei Shao, *Gpu-based parallel reservoir simulation for large-scale simulation problems*, the EAGE Annual Conference and Exhibition incorporating SPE Europec, Jun 2012.

# Appendix A

# SPE10 datafile

```
# File created:   May 11, 2011
output_prefix :     spe10
#general parameters
description :
dimension : 3
mesh_num : 0
## phase option        5 - two phase black oil model;   6 - three phase black oil
phase : 5
## ------------------------------------------------------------------------
steady : true
#parameters for density and viscosity equations
density : 1000.0
viscosity : 0.0011404
#time parameters for MPath Mixing
bom_time_output_interval : 1
bom_time_step : 1.0
bom_time_step_number : 1
bom_time_step_unit : 0
```

```
# ------------------------------------------------------------------------
## input cell property files
Kx_file_name : pdr:rtype=sf&otype=vol&ftype=raw&loc=permx.raw
Ky_file_name : pdr:rtype=sf&otype=vol&ftype=raw&loc=permx.raw
Kz_file_name : pdr:rtype=sf&otype=vol&ftype=raw&loc=permz.raw
porosity_file_name : pdr:rtype=sf&otype=vol&ftype=raw&loc=porosity.raw
mesh_type : 0
## ------------------------------------------------------------------------
#parameters for mixing matrix solver
trans_damping_factor : 1.95
trans_method : 2
trans_output_interval : 1
trans_pres_iterations : 1000
trans_tolerance : 1.0e-5
#parameters for pressure matrix solver
damping_factor : 1.95
method : 2
output_interval : 10
precision : double
pres_iterations : 5000
tolerance : 1.0e-5
# ------------------------------------------------------------------------
## well files
well1 : well1.cwell
```

```
well2 : well2.cwell

well3 : well3.cwell

well4 : well4.cwell

well5 : well5.cwell

well_file_size : 5

## ------------------------------------------------------------------------

wp_type1 : 0

wp_type2 : 0

wp_type3 : 0

wp_type4 : 0

wp_type5 : 0

wp_unit1 : 4

wp_unit2 : 4

wp_unit3 : 4

wp_unit4 : 4

wp_unit5 : 4

wp_value1 : 1000.0

wp_value2 : -1000

wp_value3 : -1000

wp_value4 : -1000

wp_value5 : -1000

wp_vari1 : perf

wp_vari2 : perf

wp_vari3 : perf
```

```
wp_vari4 : perf

wp_vari5 : perf

#conc boundary condition

#pressure boundary condition

#boundary condition

backBC : 1

conc_backBC : 1

conc_eastBC : 1

conc_frontBC : 1

conc_northBC : 1

conc_southBC : 1

conc_westBC : 1

eastBC : 1

frontBC : 1

northBC : 1

southBC : 1

westBC : 1

# ------------------------------------------------------------------------

## PVT data

datum : 12000.0

pinit  : 6014.7

woc   : 12200.0

pcwoc : 0.0

dob   : 53.0
```

```
dgb   : 0.07
dwb   : 64.0
bwi   : 1.01
vw    : 0.3
cw    : 3.0e-06
boi   : 1.01278
voc   : 2.9972
coc   : 0.000001388
grav  : 0.0
rsi   : 0.0
## ROCK data
cr    : 1.0e-6
pbasr : 6014.7
# SWT
tsw   : 0.2200 0.3000 0.4000 0.5000 0.6000 0.8000 0.9000 1.0000
tkrw  : 0.0000 0.0700 0.1500 0.2400 0.3300 0.6500 0.8300 1.0000
tkrow : 1.0000 0.4000 0.1250 0.0649 0.0048 0.0000 0.0000 0.0000
tpcw  : 6.3000 3.6000 2.7000 2.2500 1.8000 0.9000 0.4500 0.0000
## PARAMETERS
# solver_index : 0      0-orthomin(default);  1-gmres;
solver_index : 1
#implicit_index : 0
#itnmax         : 5
#itemx           : 500
```

```
north      : 12
j_direction_index : 1
qmax_index        : 1
#output_sw_raw     : 1
#report_layer      : 1
#report_well       : 1
#report_perforation : 1
#map_k             : 999
#map_perm          : 1
#map_sw            : 1
#map_dzoil         : 1
output_np         : 999
output_sw_raw     : 1
#input_mapdata     : 1
# period
npstart    : 0
npend      : 40
dt_period  : 50
# time step
tstart     : -0.01
tmax       : 11
dswmax     : 0.05
# well
rc         : 0.1875
```

```
welltype   : 4 1 1 1 1
bhfp       : 10000 4000 4000 4000 4000
welldatum  : 12000 12000 12000 12000 12000
qmax       :  5000  1250 1250 1250 1250
```