

DESIGN AND IMPLEMENTATION OF PARALLEL COMPUTING MODELS FOR  
SOLAR RADIATION SIMULATION

A Thesis

by

DA LIANG

Submitted to the Office of Graduate and Professional Studies of  
Texas A&M University  
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee, Jyh-Charn (Steve) Liu  
Committee Members, Vivek Sarin  
Michael P. Bishop

Head of Department, Dilma Da Silva

December 2015

Major Subject: Computer Engineering

Copyright 2015 Da Liang

## ABSTRACT

In order to simulate geographical phenomenon, many complex and high precision models have been developed by scientists. But at most time common hardware and implementation of those computation models are not capable of processing large amounts of data, and the time performance might be unacceptable. Nowadays, the growth in the speed of modern graphics processing units is incredible, and the flops/dollar ratio provided by GPU is also growing very fast, which makes large scale GPU clusters gain popularity in the scientific computing community. However, GPU programming and clusters' software deployment and development are associated with a number of challenges.

In this thesis, the geo-science model developed by I. D. Dobрева and M. P. Bishop proposed in A Spatial Temporal, Topographic and Spectral GIS based Solar Radiation Model (SRM) was analyzed. I built a heterogeneous cluster and developed its software framework which could provide powerful computation service for complex geographic models. Time performance and computation accuracy has been analyzed. Issues and challenges such as GPU programming, job balancing and scheduling are addressed.

The SRM application running on this framework can process data fast enough and be able to give researchers rendering images as feedback in a short time, which improved the performance by hundreds of times when compared to the current performance in our available hardware, and the speedup can easily be scaled by adding

new machines.

## DEDICATION

To my parents who support me for my dream, and my girlfriend Haiqing Wang who walks with me in the present and in the future.

## ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Jyh-Charn Liu, and my committee members, Dr. Vivek Sarin and Dr. Michael P. Bishop, for their guidance and support throughout this research. And I would like to appreciate Iliyana Dobрева for her contribution to her previous project and her help to my research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my mother and father for their encouragement and to my girlfriend for her patience and love, the most important in my life.

## NOMENCLATURE

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphic Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
CUDA	Compute Unified Device Architecture
SM	Streaming Multi-Processors
SP	Streaming Processor
MPI	Message Passing Interface
NFS	Network File System
SRM	Spatial-Temporal, Topographic and Spectral GIS based Solar Radiation Model
SLB	Static Load Balancing
DLB	Dynamic Load Balancing

## TABLE OF CONTENTS

	Page
ABSTRACT .....	ii
DEDICATION .....	iv
ACKNOWLEDGEMENTS .....	v
NOMENCLATURE .....	vi
TABLE OF CONTENTS .....	vii
LIST OF FIGURES .....	ix
LIST OF TABLES .....	xii
1. INTRODUCTION .....	1
2. INTRODUCTION TO THE SOLAR RADIATION MODEL .....	4
2.1 SRM Work Flow .....	4
2.2 System Static Analysis .....	8
2.3. System Dynamic Analysis .....	15
3. FINE GRANULARITY PARALLELIZATION .....	20
3.1 GPGPU Programming and CUDA Model .....	20
3.2 Parallelization Model .....	21
3.3 Performance .....	32
4. COARSE GRANULARITY PARALLELIZATION .....	42
4.1 Tools: MPI and Open MP .....	43
4.2 Coarse Granularity Parallelization Model .....	44
4.3 Load Balancing and Network Overhead .....	47
4.4 Performance .....	53
5. IMPLEMENTATION DETAILS .....	61
5.1 CUDA-C. MPI, Open MP Compiling and Library Loading .....	61

5.2 System Configuration .....	63
6. CONCLUSION .....	66
REFERENCES .....	67
APPENDIX .....	69

## LIST OF FIGURES

FIGURE	Page
1	Irradiance Computing for Each Pixel. Computing Block 1 per Pixel with Parallel Logic.....6
2	Modifier for Every Pixel. Computation Block 2 per Pixel, with Serial Logic ...7
3	Shadow Justification. Computation Block 3 per Pixel, with Serial Logic .....7
4	SRM Result.....8
5	UML for Main Function of SRM ..... 11
6	Software Architecture of SRM ..... 12
7	Data Matrix of DEM..... 12
8	Data Matrix for Sky-view Factor..... 13
9	Data Matrix for Slope ..... 13
10	Data Matrix for Aspect ..... 14
11	Data Matrix for Geoid ..... 14
12	Time Cost for Different Computing Phase ..... 18
13	Comparison of Time Cost for I/O Bound, Memory Bound, and Computation Amount ..... 18
14	UML Graph for Terrain::RunPixelBased4 () .....22
15	Pixel Level Fine Granularity Parallelization .....23
16	Work Flow of Global_Irradiance() .....24
17	Usage of Buffer in Global_Irradiance().....26
18	Eliminating Buffer in Global_Irradiance() .....27

19	Modified Work Flow of Global_Irradiance() .....	28
20	SRM Fine Granularity Parallelization Model in CUDA Model .....	30
21	Data Flow and SRM Architecture with GPU .....	31
22	Speeds Up of SRM with Fine Granularity Parallelization (Slave-Titan) .....	35
23	Speeds Up of SRM with Fine Granularity Parallelization (Slave-GTX) .....	35
24	Speeds Up of SRM with Fine Granularity Parallelization (Slave-Tesla) .....	36
25	Speedup for Phase 2, with Different Area Size .....	38
26	Ratio of GPU-CPU Communication Cost / GPU Computation Cost.....	39
27	Data Matrix Produced by Serial SRM.....	40
28	Data Matrix Produced by GPU.....	40
29	Absolute Error Matrix.....	41
30	Relative Error Matrix.....	41
31	Simple Solution for SRM Phase 4.....	42
32	Open MP Shared Memory Model.....	44
33	Heterogeneous Cluster System Model.....	45
34	SRM Work Flow with GPU Cluster .....	46
35	Loading Balancing Protocol .....	48
36	Algorithm for Master Schedule Process .....	49
37	Algorithm for Slave Computation Thread.....	50
38	Algorithm for Slave Data Product Transferring .....	50
39	Shared Memory Data Pool in Slave.....	51
40	Slave Memory Management.....	52

41	Cost for Compute and File System in Each Node .....	55
42	Ideal Relative System Speedup Observed by Individual Machines in Clusters .....	58
43	Practical Relative System Speedup Observed by Individual Machines in Clusters .....	58
44	System Relative Speedup for Different Task Numbers Observed by Individual Machines in Clusters .....	59
45	Overall Cluster Speed up .....	60
46	A Makefile for Cluster Source Code .....	62
47	Hosts File .....	63
48	SSH Settings .....	64
49	NFS Settings .....	65
50	Host_file .....	65
51	SRM Application Running Script.....	65

## LIST OF TABLES

TABLE	Page
1	Time Cost for System I/O and Memory Allocation ..... 16
2	Amount of <i>malloc</i> Operations for Different Input Area Size ..... 16
3	Time Cost for Different Phases in SRM (in second)..... 17
4	Performance of GPU Implementation of Phase 1, 2 and 4 (in sec, Slave-Titan)..... 32
5	Improvement Speed Up in Each Module (in times, Slave-Titan) ..... 33
6	Performance of GPU Implementation of Phase 1, 2 and 4 (in sec, Slave-Tesla) ..... 33
7	Improvement Speed Up in Each Module (in times, Slave-Tesla) ..... 34
8	Performance of GPU Implementation of Phase 1, 2 and 4 (in sec, Slave-GTX)..... 34
9	Improvement Speed Up in Each Module (in times, Slave-GTX)..... 34
10	Average Compute Cost for Each Task in Each Node (in sec)..... 54
11	Average Data Transferring Cost for Each Task in Each Node (in sec)..... 54
12	Average Scheduling Cost for Each Task in Each Node (in sec) ..... 54
13	Average Data Product Queue Size in Each Node (Normal Network)..... 56
14	Average Data Product Queue size in Each Node (Simulated Congested Network) ..... 56
15	Cluster Performance for Different Task Size (20 Tasks, in sec) ..... 57
16	Number of Tasks Processed by Each Node..... 57

## 1. INTRODUCTION

The SRM <sup>[1]</sup> developed by Dobрева and Michael P. Bishop provided a high precision simulation of the solar radiation energy on earth surface. However, as many other scientific models, implementation on common hardware are not capable of processing large amount of data, and the time performance might be unacceptable.

Microprocessors based on single a single CPU drove rapid performance increases and cost reductions in computer architecture for more than two decades. However, this drive has slowed down since 2003 due to energy consumption and heat dissipation. Since then, many microprocessor vendors have switched to models where multiple processor cores are used in each chip to increase the processing power. <sup>[8]</sup>

Nowadays, the growth in the speed of modern GPU is incredible, and the flops/dollar ratio provided by GPU is also increasing very fast. And large scale GPU clusters gain popularity in the scientific computing community. The combination of GPU and distributed system can provide powerful computation services, and improve the time performance in orders of magnitude.

To our best of knowledge, no available tools can automatically translate common C/C++ codes to codes that can be run on GPU. And the open-source distributed system frameworks like Hadoop, Spark, etc. are not suitable for the characteristics of SRM.

Graphic Processing Unit is becoming popular in many fields such as biomedical science <sup>[24]</sup>, petroleum engineering <sup>[25]</sup>, geographic science simulation <sup>[26]</sup>, mathematical applications <sup>[27]</sup>, etc. Researchers and developer has made much effort to introduce GPU

programming model to improve the performance of their applications. It is a challenge to translate serial code to paralleled code with corresponding programming model.

There have been many solutions for scientific model computation acceleration using heterogeneous distributed systems that consists of multiple machines with GPU. Center for Visual Computing in Stony Brook University proposed and developed a GPU cluster and implemented the Lattice Boltzmann Method (LBM) on that cluster, and compared the performance results with the common CPU clusters. Issues such as overlapping between nodes, time performance speedup and computation efficiency were discussed in their paper <sup>[11]</sup>.

Kindratenko and his colleagues described their experiences in deploying two GPU clusters at NCSA, presented data on performance, job scheduling, resource management and other challenges posted by GPU accelerated clusters in their paper <sup>[12]</sup>.

Load balancing is an important problem in distributed system, many of strategies and algorithms were proposed, either dynamic load balancing <sup>[14, 16, 17]</sup> (DLB), or static load balancing <sup>[15]</sup> (SLB). However, they are not very suitable for our heterogeneous cluster since our cluster topological structure are not very complex and the data transferring cost is not the most important bottleneck. But inspired by Task Queue Scheme <sup>[14]</sup>, the four phase process dynamic load balancing model <sup>[16]</sup>, I developed a set of strategies to manage the load balance in each node, to guarantee the efficiency of data transferring and to prevent memory leak when data grows too large.

In my experiments, I employed three generations of GPU cards to gain insight on their performance properties. The first is Tesla card based on Tesla C1060. The second

is a Kepler card based on GTX 680. The third one is a Maxwell based on Titan-X.

This thesis is organized as below; Section 1 gives an introduction to the background of GPU and distributed system usage in scientific model computation. Section 2 gives a basic description about SRM and provides the SRM software architecture analysis. In Section 3, fine granularity parallelization model and performance are discussed. Section 4 proposes the coarse granularity parallelization model and performance. In Section 5, I would give the system implementation details. Section 6 is the conclusion of my project.

## 2. INTRODUCTION TO THE SOLAR RADIATION MODEL

### 2.1 SRM Work Flow

The Spatial Temporal, Topographic and Spectral GIS based Solar Radiation Model <sup>[1]</sup> simulates and models the solar radiation reaching from Sun to Earth surface. The data required to the model is a Digital Elevation Model (DEM), exoatmosphere irradiance standard curves, and atmosphere constituent properties. The spatial resolution is 30 m.

The SRM accounts for variations in topography since that the atmospheric properties such as water content change as a function of elevation. Multi-scaled topographic effects are taken into consideration because the surrounding terrain might block the direct solar radiation or may obscure a fraction of the sky affecting the diffuse irradiance; a sky-view factor was developed for this phenomenon. Additionally, the solar radiation model is spectral in nature to properly account for wavelength-dependent matter-energy intersections <sup>[1]</sup>. Besides variation in topography, earth orbital parameters, solar geometry, local ellipsoidal radius, atmospheric attenuation, planet gravitation, and geoid were built to simulate and model the energy behavior in each pixel of the terrain.

The computation work of solar radiation model can be described as Figure 1, Figure 1 and Figure 3. Given a certain area with elevation data, first earth gravitation, geoid, sky view and other environment configuration will be initialized. For each time stamp, such as every 30 minutes between sunrise and sunset in one day, earth orbit and Greenwich Mean Time is revised and the model will compute radiation energy for each

pixel in the sample area. For each pixel in the area, the solar irradiance energy can be calculated by the integral of the irradiance with different wavelength above that point. As shown in Figure 1.

The irradiance is calculated with sunshine spectrum, ozone spectrum, aerosol spectrum, and some other modifiers such as sky-view factor, tangent slope of the terrain, ellipsoid elevation, earth gravity, zenith and azimuth. Figure 2 shows the definition of each modifier. Furthermore, as Figure 3 illustrates, the model will evaluate if the current pixel is in a shadow where sunlight cannot directly touch. The modifiers and spectra would be merged into the target irradiance-wavelength spectrum, whose integral is the desired solar energy.

Two sets of intermediate parameters are determined by DEM: sky-view factors and geoid. SRM provides the sky-view factor model; the process is similar with Figure 3. For every pixel, like pixel A, we will scan pixels along a direction within a certain distance and calculate the angle of elevation between the current pixel and pixel A. Scanning along one direction is not enough, typically, SRM will make every pixel as a center and scans every 1 or 5 degree of 360 degree. The sky-view factor is the sum of cosine of every elevation angle. The process originally took about one week for an area with 6250000 pixels if each direction of every 1 degree for each pixel was computed, which is very slow.

The Geoid parameters can be treated as a map from latitude and longitude. Some third part libraries provided this function. SRM uses GeographicLib <sup>[21]</sup> to generated Geoid data for each location in our focused area.

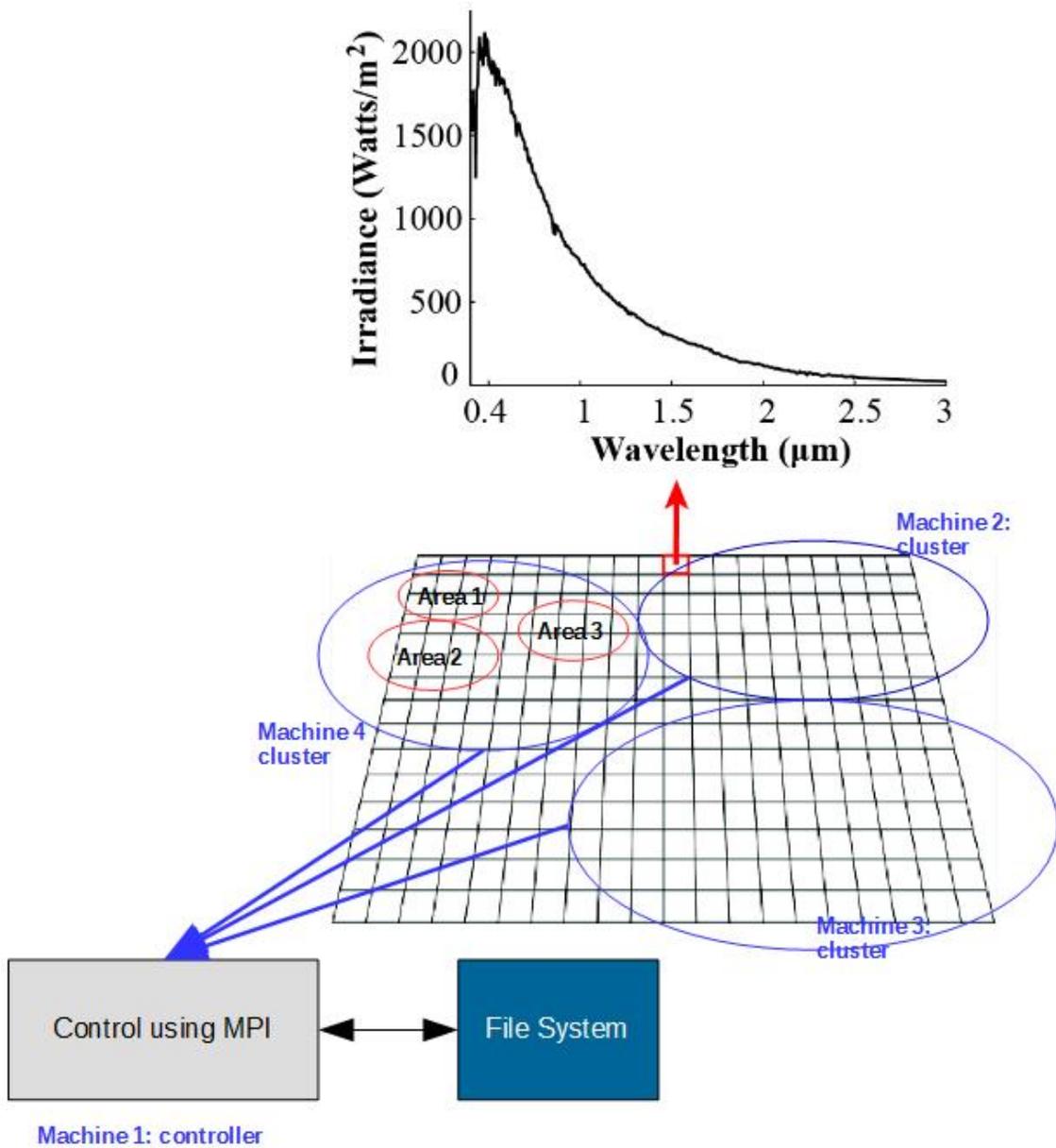


Figure 1: Irradiance Computing for Each Pixel. Computing Block 1 per Pixel with Parallel Logic

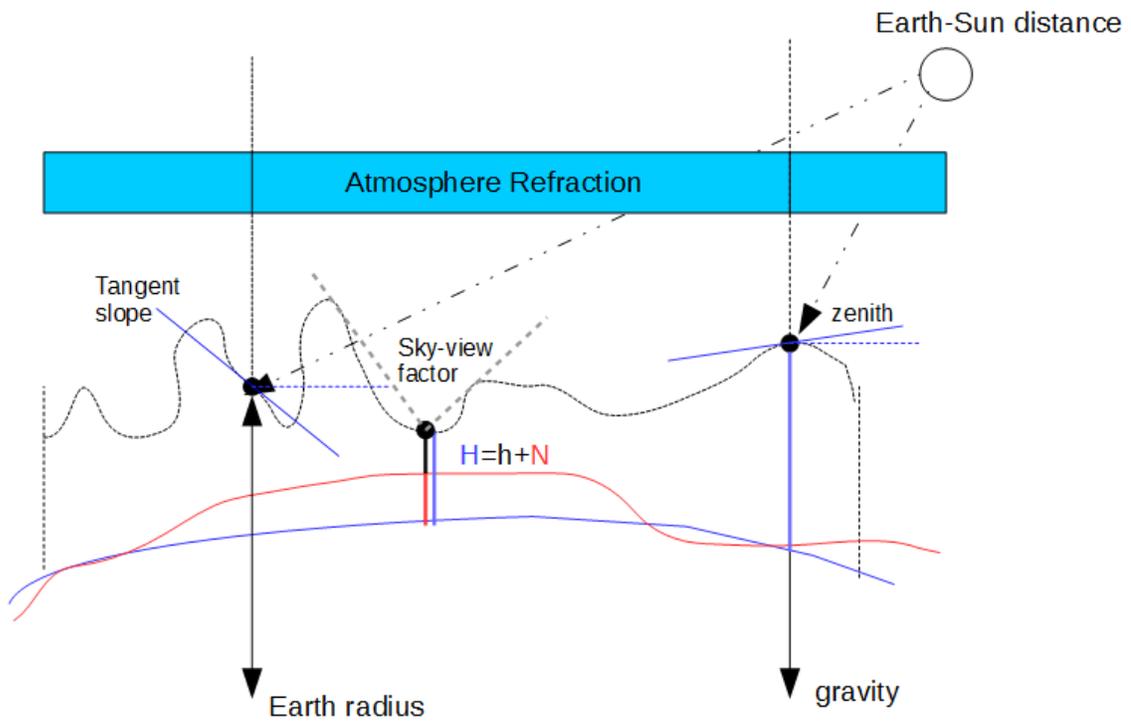


Figure 2: Modifier for Every Pixel. Computation Block 2 per Pixel, with Serial Logic

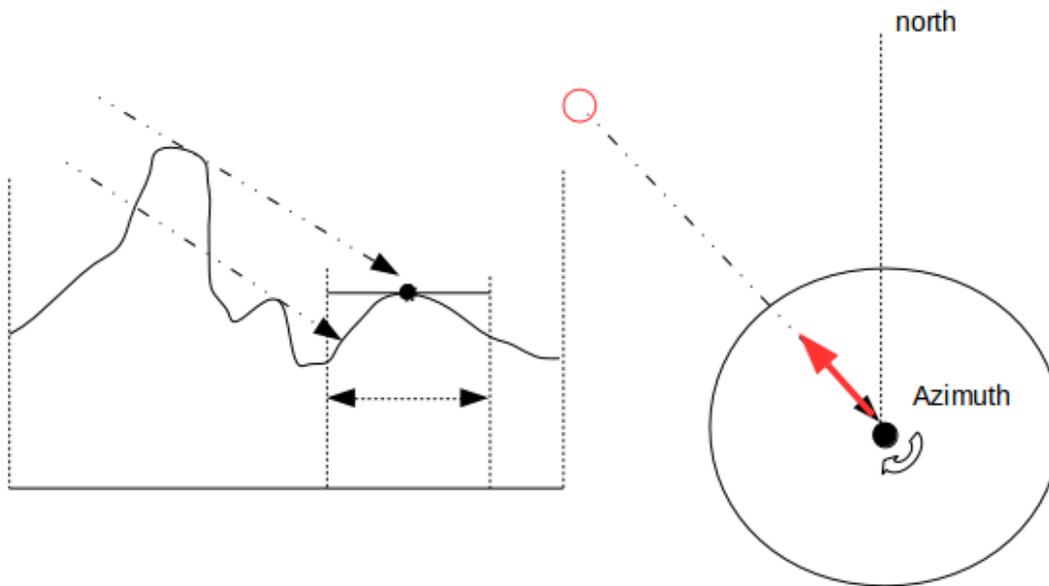


Figure 3: Shadow Justification. Computation Block 3 per Pixel, with Serial Logic



Figure 4: SRM Result

Figure 4 is a snapshot of solar energy projected to an area of Himalaya for time 43200 seconds at August 1, 2012. The x-coordinate is latitude, and y-coordinate is longitude. And value of each pixel is represented by gray scale.

## 2.2 System Static Analysis

As a prelude of the code optimization task, we perform static and dynamic analysis, which is helpful for better understanding of the model and necessary for further optimization. Some code analysis tools are used to help us clarify and analyze the code

structure. Doxygen<sup>[20]</sup> is beneficial to clarify relations between functions and classes. The commercial software Understand<sup>[19]</sup> is used for create control flow UML graph. PIN is helpful when we need to instrument the running binary code. We use the C++ standard library *<chrono>* as our timer for time performance. Besides code reviewing, the information provided by these tools is critical, and make it easier to analysis the model in system level.

As previous introduction, the SRM consists of several geographical scientific modules: Atmosphere, Terrain, Spectra, Orbit, and Planet, and they simulate different nature procedures. What we provide to SRM is a matrix of elevation data of a rectangle area we want to focus on, along with some other configurations like pixel resolution, atmosphere parameters, etc. SRM will produce the irradiance energy data for every pixel in the input focused area matrix. Besides the geographical models, there are several functionalities for data preparation and work scheduling. The dependencies between modules are complex. Figure 5 is generated by Understand<sup>[19]</sup>, and it is the visualizing of the main function of SRM application. It provides us with the basic work flow of the model. First it creates a Terrain object, which is the abstraction of the focused area. And then takes a series of functions to initialize some parameters of focused area, which include elevation, latitude, longitude, aspect and slope of the surface. Next, we can see that there are four loops for years, months, days and seconds. Note that the seconds within one day are only in the daytime that from sunrise to sunset. Here the model does the same operation for every time among the time periods we input: calculate the solar radiation energy for each pixel in the area, and write result in hard disk if necessary.

For every time stamp of the focused area, function *Terrain::RunPixelBased4 ()* is called to calculate the radiation energy reached in the focused area. For one time and an area of size 2500\*2500 pixels, this function would take nearly one hour to complete, which is an obvious bottleneck for SRM.

Now since the work flow of SRM is clear, the software architecture can be clarified. Figure 6 shows the software architecture in system level. The SRM consists of several computation phases. The first phase is to initialize some mediate parameters with DEM data. The mediate parameters include latitude, longitude and aspect and slope for surface. During the second phase, sky view factors are initialized. In the next phase, geoid parameters are generated with latitude and longitude data. In the last phase, the solar radiation energy for the research area will be compute. There might be more than one computation tasks in Phase 4. Each task corresponds to one unique time stamp. All tasks in Phase 4 are organized in a job queue, and processed one by one in order.

Figure 7 to 11 and Figure 4 illustrate the matrix of data for each phase in SRM. The matrix represents an area of earth surface, with x-coordinate of latitude and y-coordinate of longitude. Value of each pixel is represented with gray-scale.



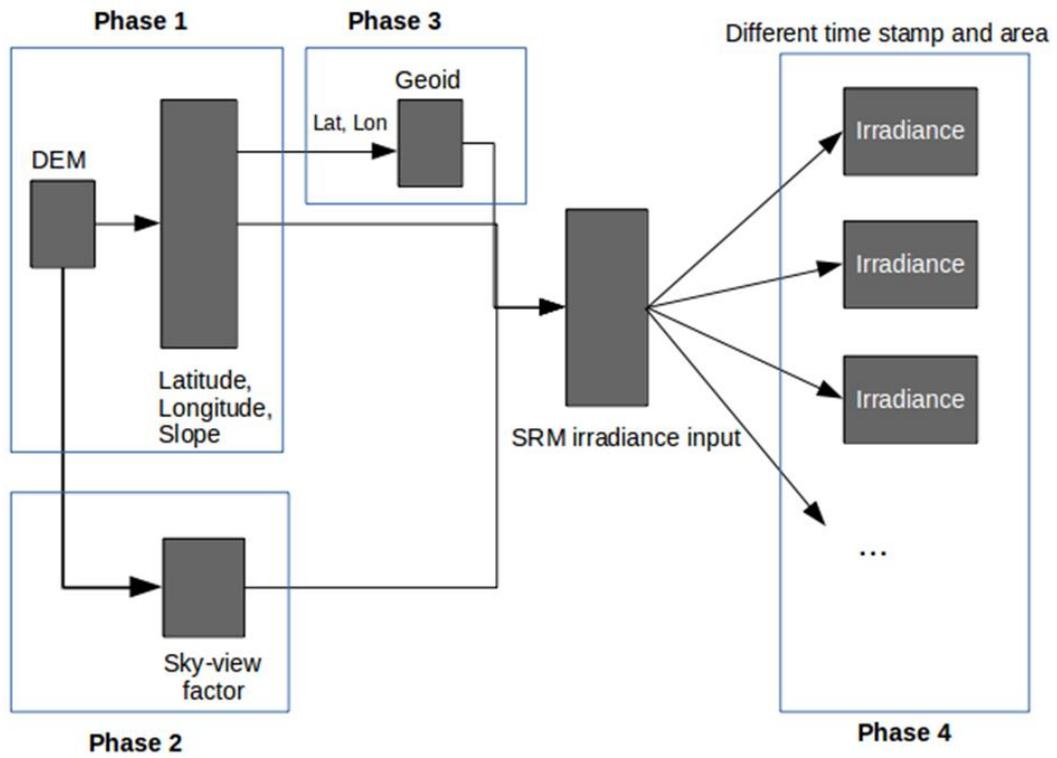


Figure 6: Software Architecture of SRM

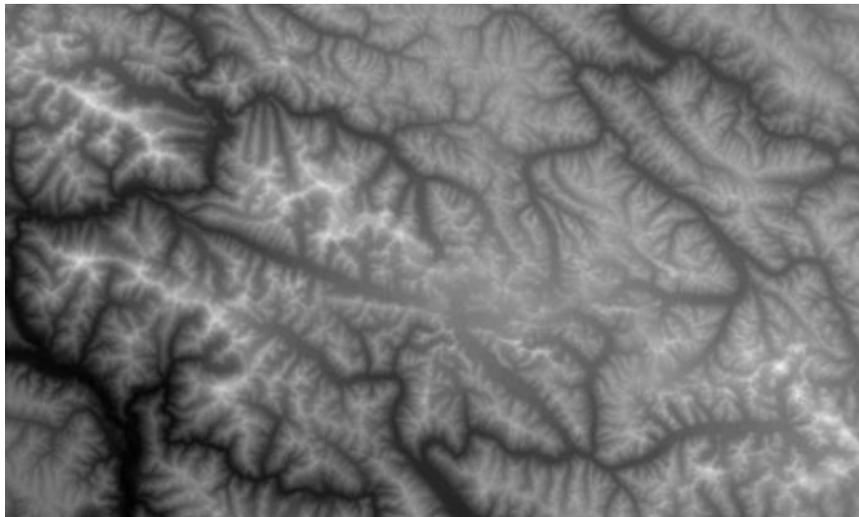


Figure 7: Data Matrix of DEM

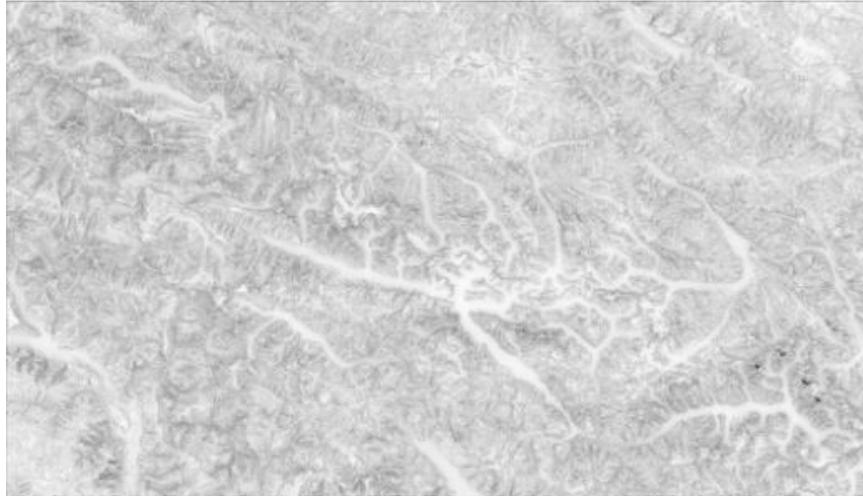


Figure 8: Data Matrix for Sky-view Factor

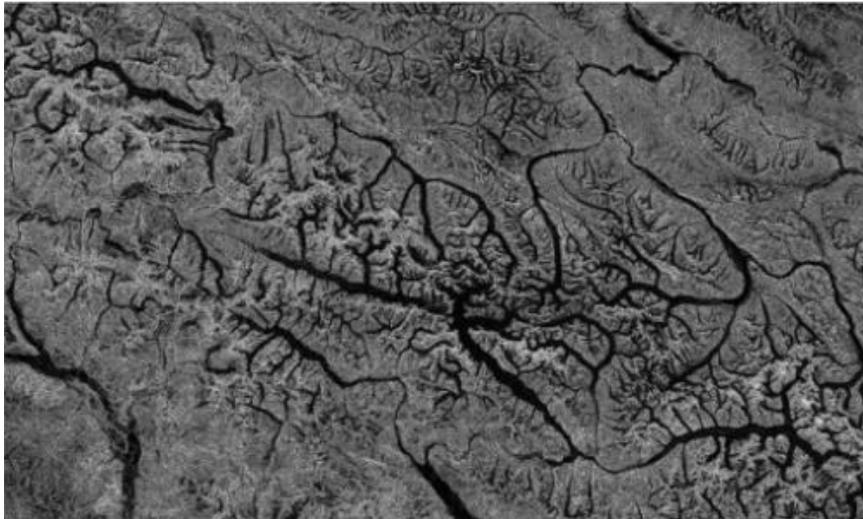


Figure 9: Data Matrix for Slope

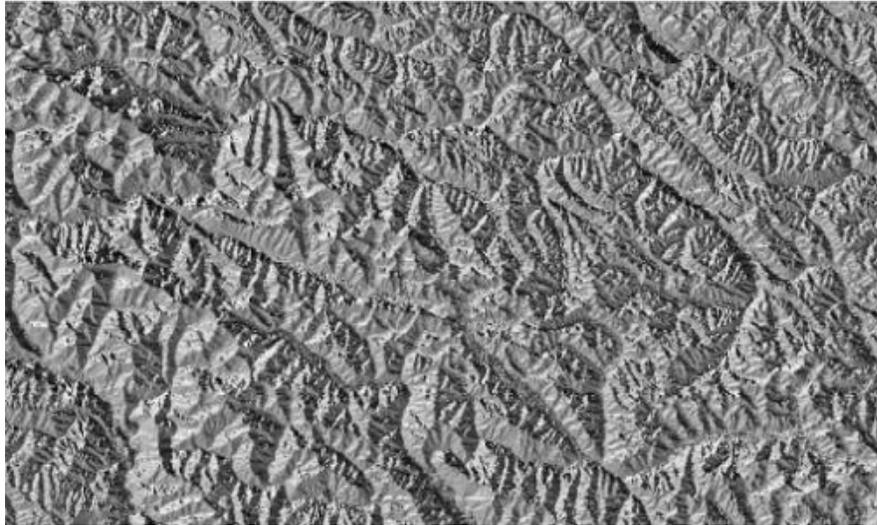


Figure 10: Data Matrix for Aspect



Figure 11: Data Matrix for Geoid

### 2.3. System Dynamic Analysis

With clear structure static analysis of SRM framework, in this section, dynamic analysis of the system is investigated with different criteria. Based on the performance of system by each criterion, we could find out the system bottleneck, which would lead to further optimization strategies. In this part, we evaluated the system performance based on three criteria: I/O bound, memory bound, and computation amount.

According to Figure 6 and discussion in previous, there are several I/O operations from memory to hard disk in the SRM system, such as reading DEM from disk, reading input parameters from disk, store sky-view and geoid data into disk as intermediate results, and writing data product to hard disk for further usage. I/O operations are more expensive than other common instructions like computing or logic control, which might be a potential system bottleneck. To evaluate the I/O overhead, an efficient way is to measure total I/O overhead running time. Table 1 shows the I/O overhead in SRM system.

Besides I/O bound, other relatively expensive operations especially memory dynamic allocation need to be taken into consideration. There are large amount of memory allocation and releasing operations when SRM application running. The memory dynamic manipulation might be a potential system bottleneck. We can instrument the running binary code to measure the memory manipulation amounts. Table 2 shows the amount of memory allocation with different input area size. We can see that the amount of memory manipulation operation grows large as the input size grows. To evaluate the performance overhead caused by memory manipulation, the time

costs of malloc instructions are needed. Table 1 also provides time costs of malloc operations in the running SRM binary code. The time cost for each criterion is measured by C++ standard library *<chrono>* (Appendix C). The experiments are all running at machine Slave-Titan, as shown in Appendix B. The data come from average of 10 experiments.

Table 1: Time Cost for System I/O and Memory Allocation

<b>Area Size</b>	<b>I/O cost/sec</b>	<b>Malloc Cost/sec</b>
11*11	0.0003	0.00001
100*100	0.005	0.00002
200*200	0.02	0.00004
400*400	0.09	0.00013
800*800	0.37	0.00025

Table 2: Amount of *malloc* Operations for Different Input Area Size

<b>Area Size (pixel*pixel)</b>	<b>Amount of malloc Calls</b>
11*11	17230
100*100	36644
200*200	96946
400*400	339838
800*800	1314662

Another criterion that we use to evaluate system performance is computation amount. That is the amount of pixels in the DEM matrix. As described in Section 2.1, spectrum for each pixel in input matrix has to be processed. Therefore the input size will affect system time performance.

Time cost for each computing phase is different. If we use the parameters configuration provided in Appendix A, the time cost for computing in each phase is shown in Table 3. (We only compute the irradiance result for task).

Table 3: Time Cost for Different Phases in SRM (in second)

<b>Area Size</b>	<b>Phase 1</b>	<b>Phase 2</b>	<b>Phase 3</b>	<b>Phase 4</b>	<b>Total</b>
11*11	0.000233	0.004	0.000062	0.05	0.05
100*100	0.018	5.35	0.0005	6.32	6.32
200*200	0.073	19.22	0.0023	23.13	23.13
400*400	0.285	68.35	0.0103	91.56	91.56
800*800	1.148	253.12	0.0433	369.27	369.27

Figure 12 illustrates the time cost for computing in each phase. We can find that as area size grows large, the time cost will increase. And performance bottleneck of the computing exists in Phase 2 and 4.

Figure 13 illustrates the comparison of time cost of three different criteria. We

can find that the criterion computation amount dominates the time cost for the system.

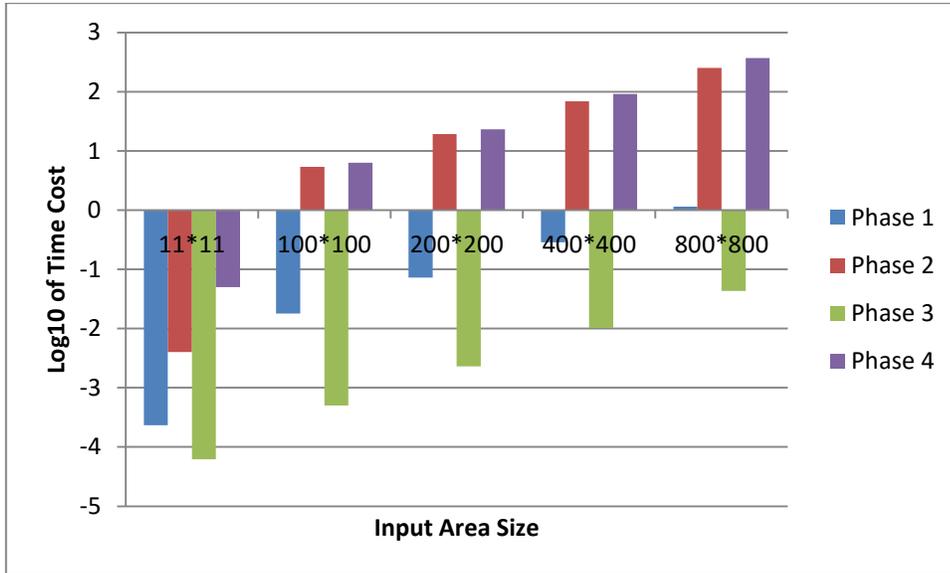


Figure 12: Time Cost for Different Computing Phase

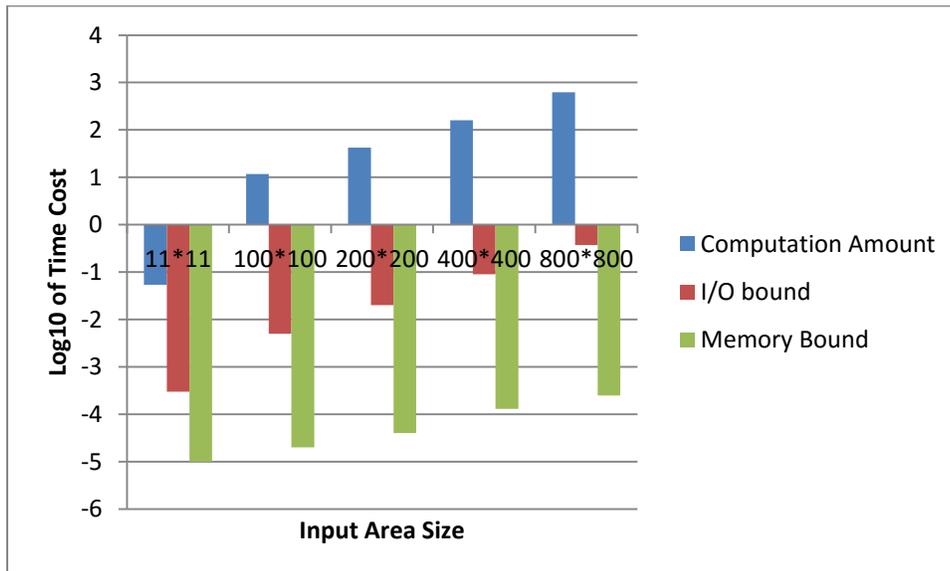


Figure 13: Comparison of Time Cost for I/O Bound, Memory Bound, and Computation Amount

With dynamic analysis of system performance based on different criteria, we can make the conclusion that computing amount especially cost in SRM phase 2 and phase 4 is the system performance bottleneck. Although there are I/O bound and memory bound in system, compared with computation amount, they are not the dominated bottleneck. Therefore to improve the system performance, taking the computation work for each pixel in parallel might be helpful. In the next section, I proposed the parallelization strategies on the level of area pixel level.

### 3. FINE GRANULARITY PARALLELIZATION

#### 3.1 GPGPU Programming and CUDA Model

In 2007, NVIDIA introduced their GPU programming model CUDA <sup>[2]</sup>, which allows software developers to use a CUDA-enabled GPU for general-purpose processing in popular languages like C/C++ or FORTRAN without knowing the details about the graphics programming skills, which are required by GPU programming in early days <sup>[13]</sup>. CUDA programming is relatively easy to learn and it is an extension of standard ANSI C, FORTRAN, with some keywords that indicate some special CUDA new functions and corresponding data structure on devices <sup>[2]</sup>. All recent released NVIDIA GPUs support CUDA <sup>[18]</sup>.

CUDA abstracts the thread-level parallelism of the GPU into a hierarchy of threads (grids of blocks of warps of threads) <sup>[2]</sup>. Threads are mapped into a hierarchy of hardware resources. Blocks of threads are executed within streaming multiprocessors. While the programming model uses collections of scalar threads, the SM more closely resembles an eight-wide vector processor operating on 32 wide vectors. Streaming multiprocessors are the hardware that implements physical parallelized processing of GPU. To optimize GPU performance, GPU memory hierarchy and resource management must be taken into consideration. Memory on different levels has different access latency and storage capability. It would be a challenge to write CUDA code that is fully optimized. In the following research I have translated the performance bottleneck modules in SRM from serial version to CUDA device parallelized code.

### 3.2 Parallelization Model

Data in Table 3 shows that as the matrix size input to Phase 1, 2 and 4 of SRM become larger, the time cost would become more expensive. In this section I would take the operation in phase 4: *Terrain::RunPixelBased4* () as an instance, and propose the parallelization model.

We use Understand <sup>[19]</sup> to generate the UML control flow graph for function *Terrain::RunPixelBased4* (), as illustrated in Figure 14. This module corresponds to one element of phase 4 in Figure 5. Four classes are used in this function. The Terrain object is the abstraction of the area we focused on. Class Planet, Atmosphere and Orbit are referred to during processing the focused area, which are responsible for earth planet model, earth orbit model based on orbital parameters passed in, and the atmosphere above the ground based on the spectrum data, ozone and aerosol parameters respectively. After entering *Terrain::RunPixelBased4* (), first it reads two sets of data that has been prepared before from disk: the sky-view factors and geoid. (The preparation of these two sets of data is another phase.) Then, the minimum and maximum values of the elevations in the focused area are calculated. Next the orbit class will initialize some parameters for the area matrix. Then every pixel in the matrix will be processed. Models of planet, atmosphere and orbit will participate in the processing. SRM would use some scientific model to calculate the solar radiation value for each pixel. Note that there is a small loop for each pixel. That is for the shadow evaluation mentioned in Figure 3: go to neighboring pixels in azimuth direction within a max range, in each step use some conditions to judge whether the current pixel is in shadow.

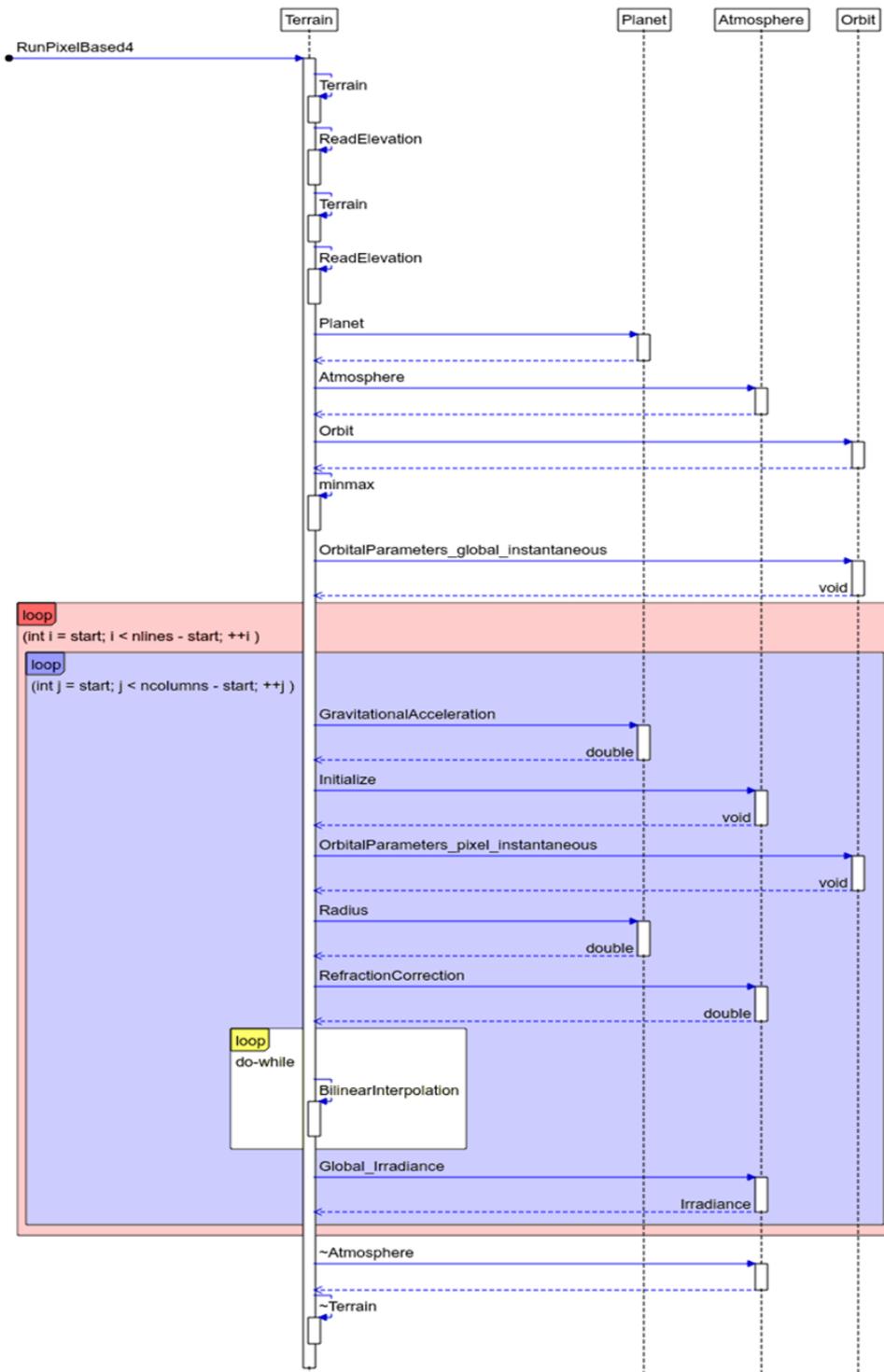


Figure 14: UML Graph for Terrain::RunPixelBased4 ()

The matrix process pattern is very suitable for GPU: Single Program Multiple Data. Each pixel follows same instructions with just different parameters given. And the data size is huge: about 160000 to 10000000 pixels, which can be considered as basic computation threads in CUDA model. Based on this idea, we can praise the basic

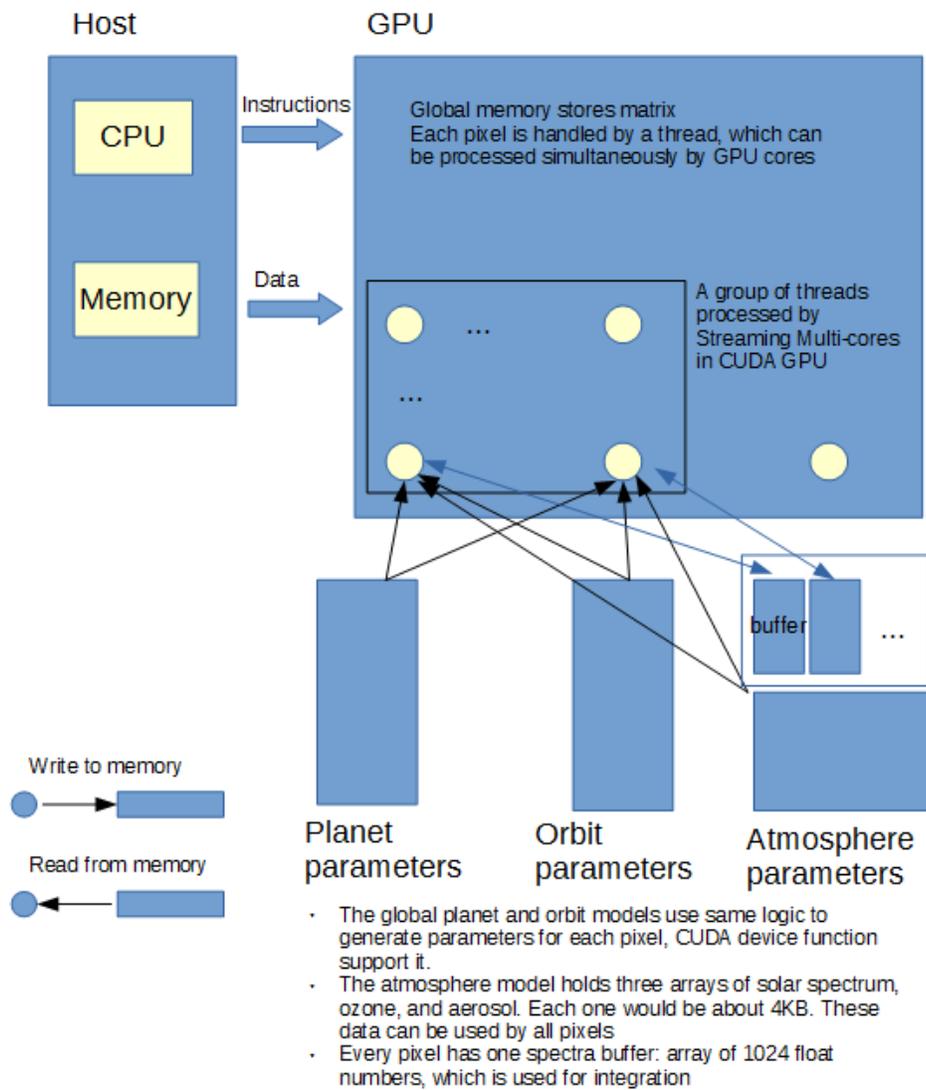


Figure 15: Pixel Level Fine Granularity Parallelization

solution: map pixels processing to threads of GPU kernel, which could be operated in parallel by SM. And store parameters needed by model in GPU memory. Figure 15 illustrates this basic fine granularity parallelization model in CUDA. Pixels in area matrix are mapped into threads in GPU. Parameters including planet, orbit, and atmosphere are loaded from main memory in host, and stored in GPU memory shared by all threads. Each thread requires an extra buffer for mediate results. The buffer could be allocated in global memory of GPU.

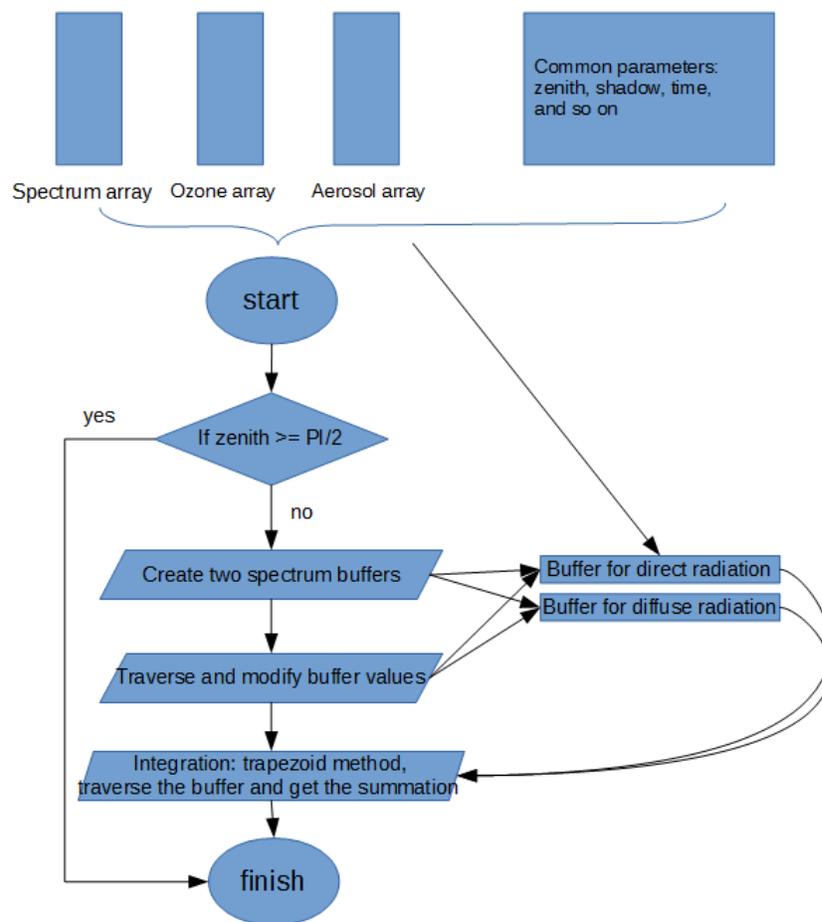


Figure 16: Work Flow of Global\_Irradiance()

In Figure 15 we can notice that the Atmosphere model has to use data buffers. That corresponds to the function *Atmosphere::Global\_Irradiance ()* in Figure 14. Figure 16 illustrates the work flow in *Atmosphere::Global\_Irradiance ()*. First some parameters such as zenith, shadow, time stamp, and three read-only data arrays which include solar spectrum, ozone spectrum and aerosol spectrum were given as input, then it check whether zeniths is larger than 90 degrees. If it is, set irradiance to zero and finish the process. If it is not, two buffers are created, whose x-value corresponds to wavelength, and y-value corresponds to energy value. These buffers are used to stored intermediate spectrum data for each wavelength.

There is a problem in the strategy shown in Figure 15. We can see that each pixel would need two buffers of 512 elements for spectrum calculation. The buffer size is  $2 * 512 * 4 = 4096$  Bytes. The buffer used by all pixels in a  $400 * 400$  matrix would be  $4096 * 400 * 400$  Bytes= 625 MB; in a matrix of  $2500 * 2500$ , it would be 24 GB. That is a very heavy memory burden. To solve the problem, the work flow in Figure 16 should be redesigned. . Figure 17 illustrates that the buffers are needed in two phases. First, the input spectrum was scanned wavelength by wavelength, and we update values in buffer for each wavelength. In Phase 2, the buffer would be scanned wavelength by wavelength to calculate the integral. The two-time scanning can be merged into one scan. During the scanning for each wavelength, buffer values are updated, and are accumulated for integral.

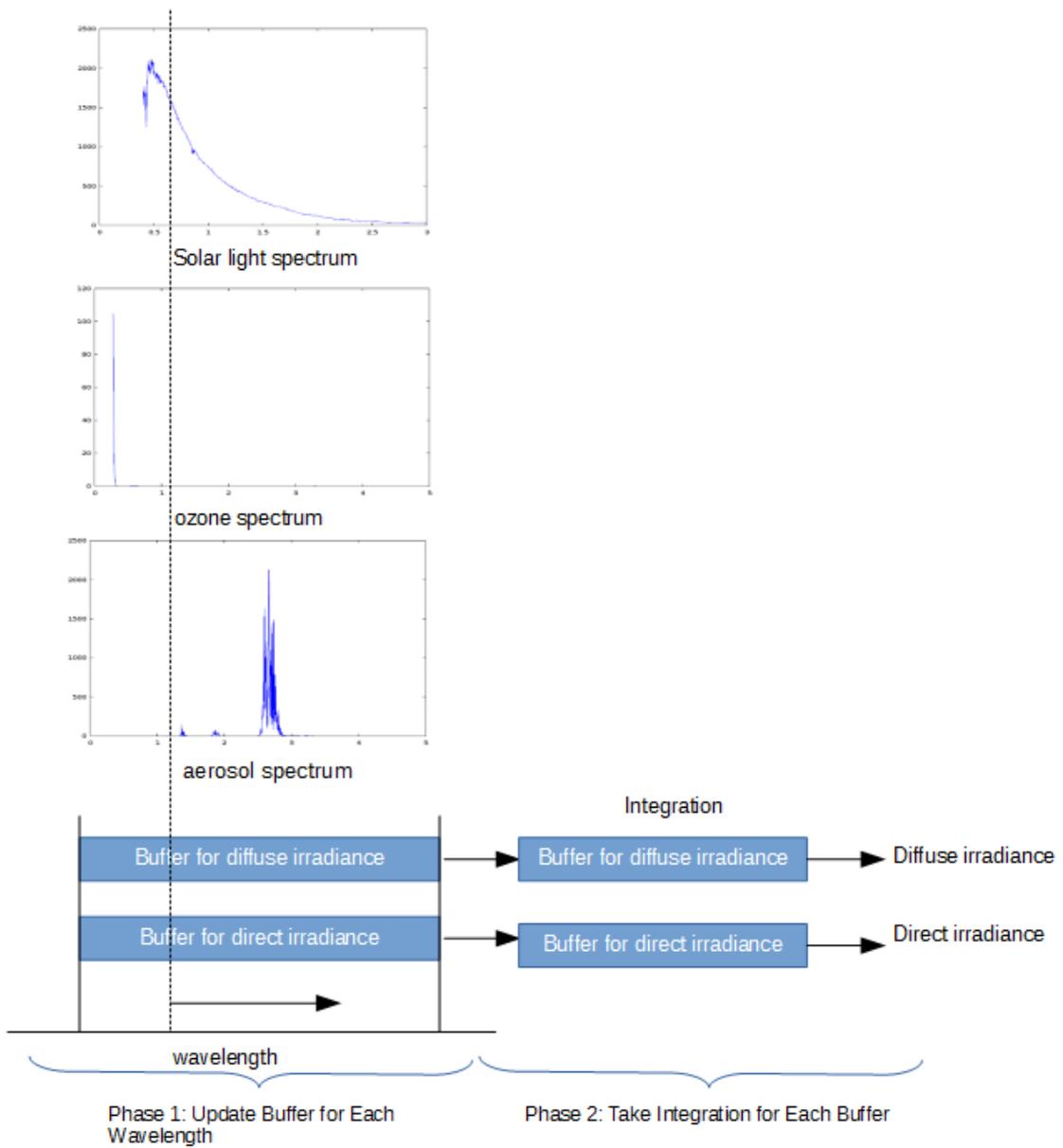


Figure 17: Usage of Buffer in Global\_Irradiance()

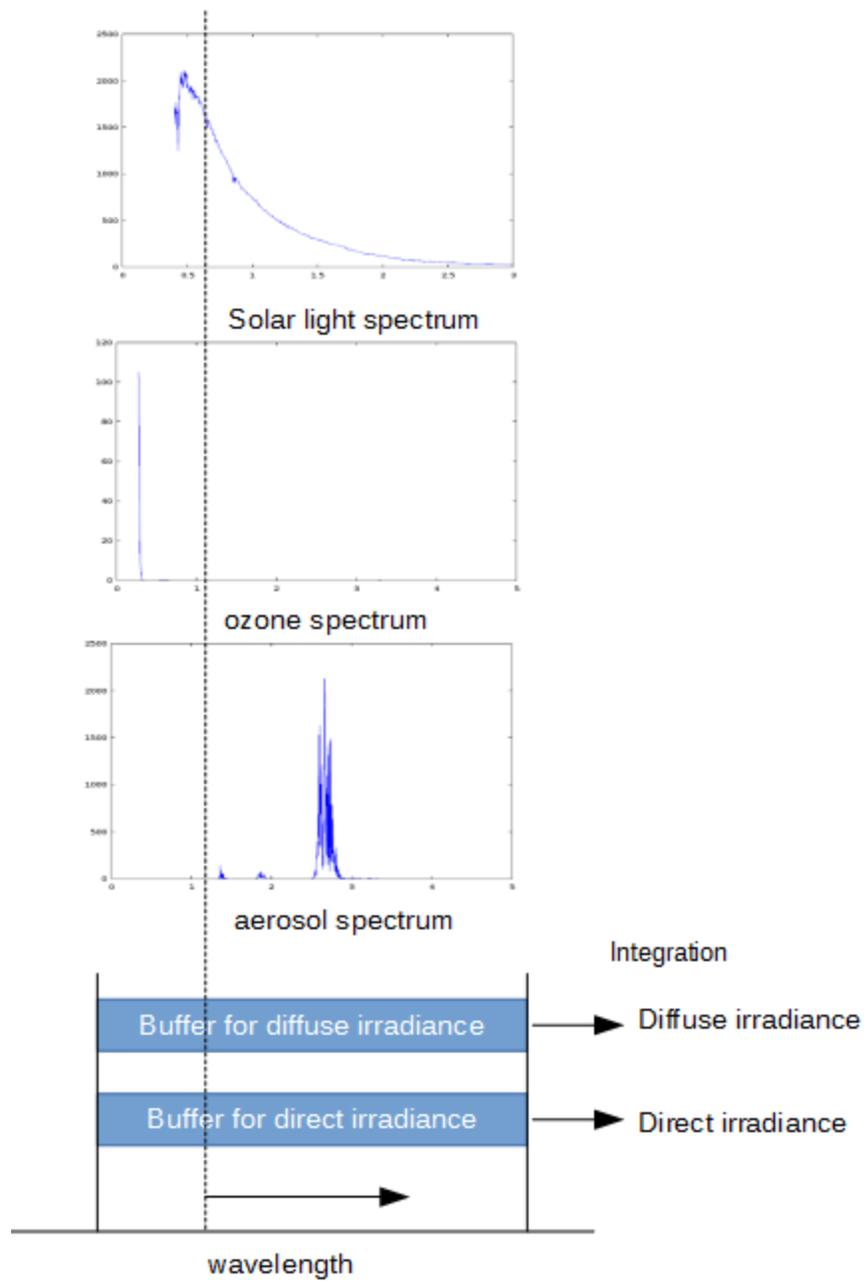


Figure 18: Eliminating Buffer in Global\_Irradiance()

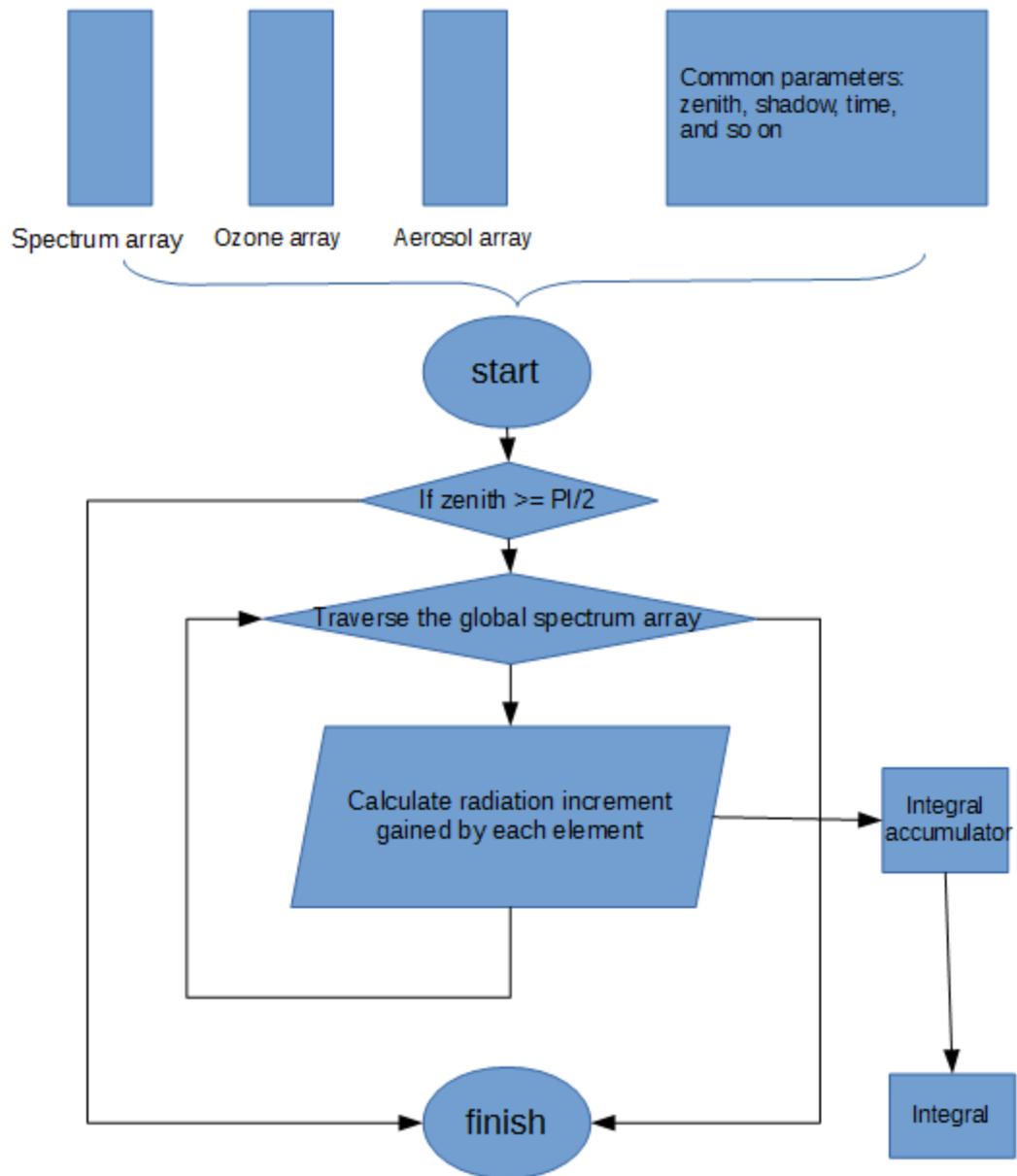


Figure 19: Modified Work Flow of `Global_Irradiance()`

Figure 18 illustrates the merging of two phases in Figure 17. What's more, Figure 19 illustrates the modified work flow of function `Global_Irradiance()`. In this

case, buffers are eliminated. We only loop along with the wavelength once and use one intermediate value to update the energy and accumulate the integration. The space complexity is reduced from  $O(n)$  to  $O(1)$ , where  $n$  is the length of spectrum wavelength. In practice, we need only two variables of diffuse irradiance and direct irradiance, which only cost  $2*4 = 8$  Bytes for single float precision in each pixel. Compared to original 4096 Bytes, we save 512x memory space.

The parallelization strategy for phase 1 and 2 is similar with the phase 4: Map pixels of the input area matrix to threads in GPU. Figure 20 illustrates the relations between the fine granularity parallelization model for the bottleneck modules in SRM and CUDA model. Constant memory in GPU is small but has fast access time; we can put a small part of parameters that are accessed most frequently into that space, and put other parameters and data into the GPU global memory, which provides larger space but is relatively slow to access. When a GPU kernel has been launched, the data of each pixel thread would be pushed into the SM register and SM then process each thread in parallel.

When matrix area comes to large, it might exceed the threads capacity that GPU could process. In this case we have to partition the area into small pieces, which could be processed by GPU at one time.

Figure 21 shows the software architecture of SRM with support of the fine granularity parallelization support. The system architecture is the similar with Figure 6, but the computation module for Phase 1, 2 and 4 are implemented with GPU, not in CPU. In the modified SRM, for Phase 1 and 2 input areas might be cut into small pieces, and

encapsulate each piece into a task in the job queue for further GPU process. And tasks in Phase 4 are different with those in Figure 3.3. Tasks are encapsulated with information about pieces cutting in an area besides time stamp. Tasks in queue would be completed by one machine in serial or different machine simultaneously. Next Section would discuss this issue in details.

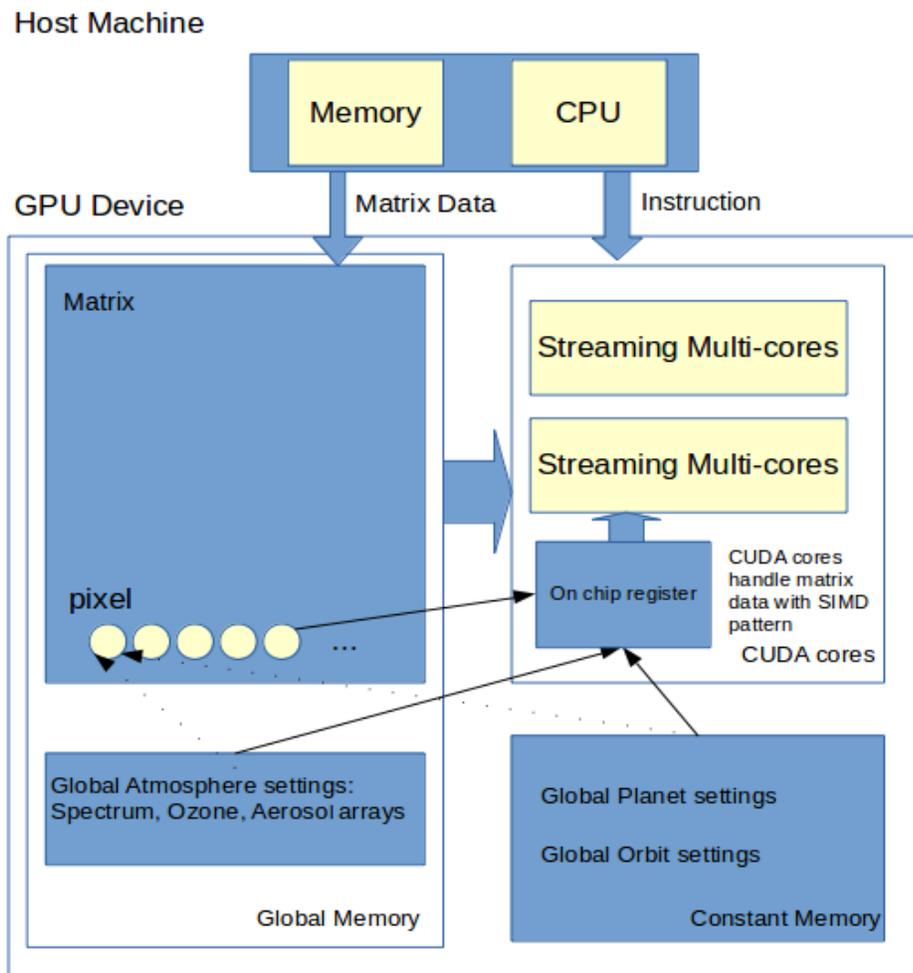


Figure 20: SRM Fine Granularity Parallelization Model in CUDA Model

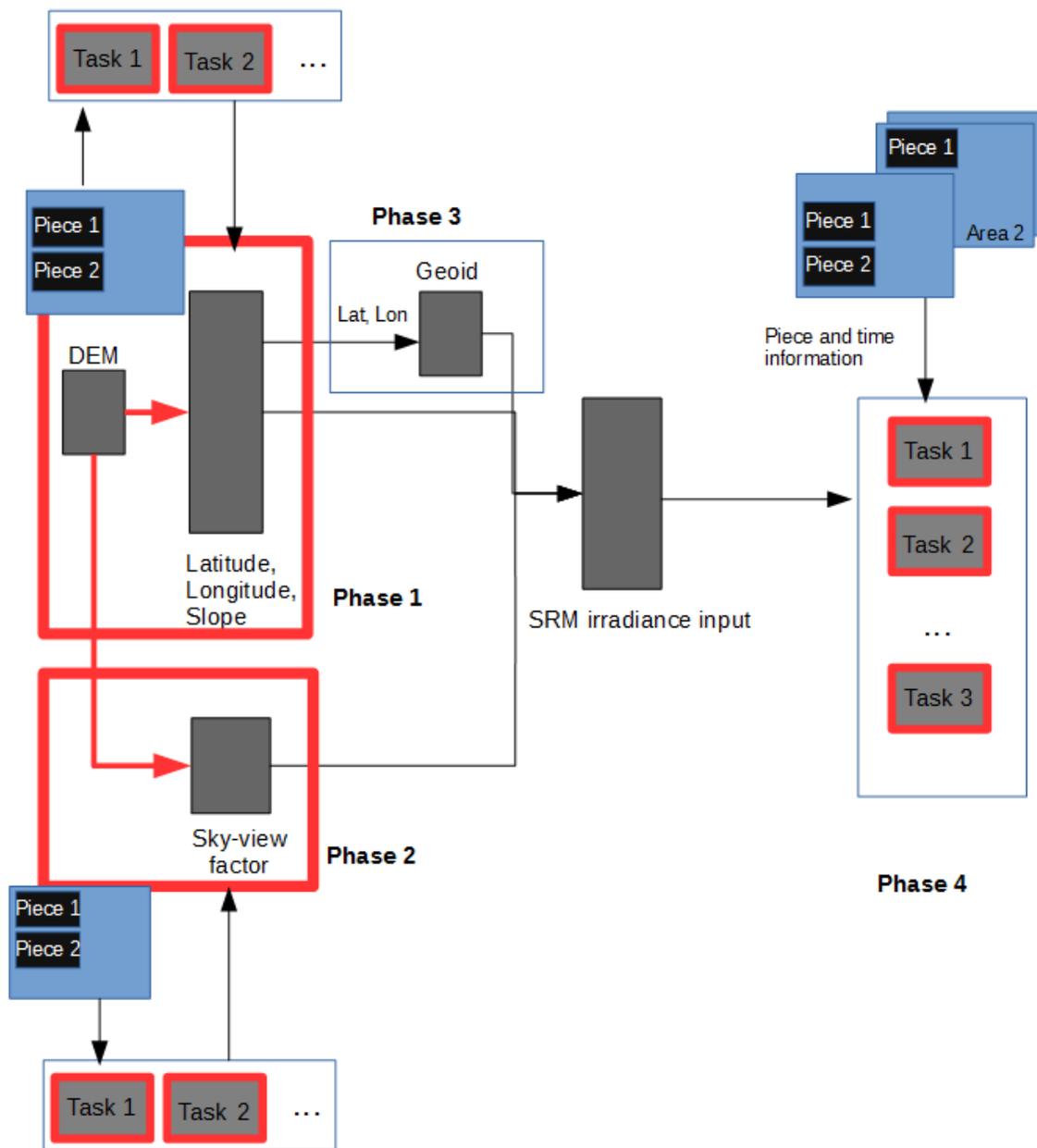


Figure 21: Data Flow and SRM Architecture with GPU

### 3.3 Performance

The performance of these bottleneck modules with fine granularity parallelization implementation on different types of GPU is shown in Table 4 to Table 9. The hardware specification is shown in Appendix B. There are three types of GPU: Titan-X, GTX 680, and Tesla 1060C. The time cost for is measured by CUDA timer (Appendix C). The data come from average of 10 experiments. The input configuration is the same with that of Table 1, and only one task was tested in Phase 4.

Table 4: Performance of GPU Implementation of Phase 1, 2 and 4 (in sec, Slave-Titan)

<b>Area Size</b>	<b>Phase 1</b>	<b>Phase 2</b>	<b>Phase 4</b>
<b>11*11</b>	0.00023	0.004	0.055
<b>400*400</b>	0.008	0.23	1.08
<b>2500*2500</b>	0.256	17.14	37.13
<b>7540*4529</b>	1.39	152.77	193.26

With data in Table 4, Table 6 and Table 9, it's obvious that performance of Titan X is better than that of GTX 680, which is better than that of Tesla C1060. From the GPU specification in Appendix B, we can see that there are 240 SMs in Tesla, 1536 in GTX 680, while 3072 in Titan, which determine the computation power. What's more, the threads limit in Tesla is 512\*65535 if we only use the x-dimension, which is smaller than 7540\*4279. Therefore Tesla C1060 is not able to process the area size with

7540\*4529 pixels.

Table 5: Improvement Speed Up in Each Module (in times, Slave-Titan)

<b>Area Size</b>	<b>Phase 1</b>	<b>Phase 2</b>	<b>Phase 4</b>
<b>11*11</b>	0.19	0.5	0.03
<b>400*400</b>	35.26	295.69	85.05
<b>2500*2500</b>	45.6	463.14	99.14
<b>7540*4529</b>	43.65	488.27	104.34

Table 6: Performance of GPU Implementation of Phase 1, 2 and 4 (in sec, Slave-Tesla)

<b>Area Size</b>	<b>Phase 1</b>	<b>Phase 2</b>	<b>Phase 4</b>
<b>11*11</b>	0.00036	0.00157	0.277
<b>400*400</b>	0.0402	0.656	7.43
<b>2500*2500</b>	1.55	59.09	388.86
<b>7540*4529</b>	-	-	-

Table 7: Improvement Speed Up in Each Module (in times, Slave-Tesla)

<b>Area Size</b>	<b>Phase 1</b>	<b>Phase 2</b>	<b>Phase 4</b>
<b>11*11</b>	0.647222222	2.547770701	0.180505415
<b>400*400</b>	7.089552239	104.1920732	12.3230148
<b>2500*2500</b>	7.212903226	134.1716026	9.466028905
<b>7540*4529</b>	0	0	0

Table 8: Performance of GPU Implementation of Phase 1, 2 and 4 (in sec, Slave-GTX)

<b>Area Size</b>	<b>Phase 1</b>	<b>Phase 2</b>	<b>Phase 4</b>
<b>11*11</b>	0.00181933	0.005	0.0700826
<b>400*400</b>	0.010289	0.42	1.3004
<b>2500*2500</b>	0.365	40.077	43.317
<b>7540*4529</b>	1.85	238.795	219.88

Table 9: Improvement Speed Up in Each Module (in times, Slave-GTX)

<b>Area Size</b>	<b>Phase 1</b>	<b>Phase 2</b>	<b>Phase 4</b>
<b>11*11</b>	0.128069124	0.8	0.713443851
<b>400*400</b>	27.69948489	162.7380952	70.40910489
<b>2500*2500</b>	30.63013699	197.8241884	84.97726066
<b>7540*4529</b>	32.82702703	312.3608116	92.01246134

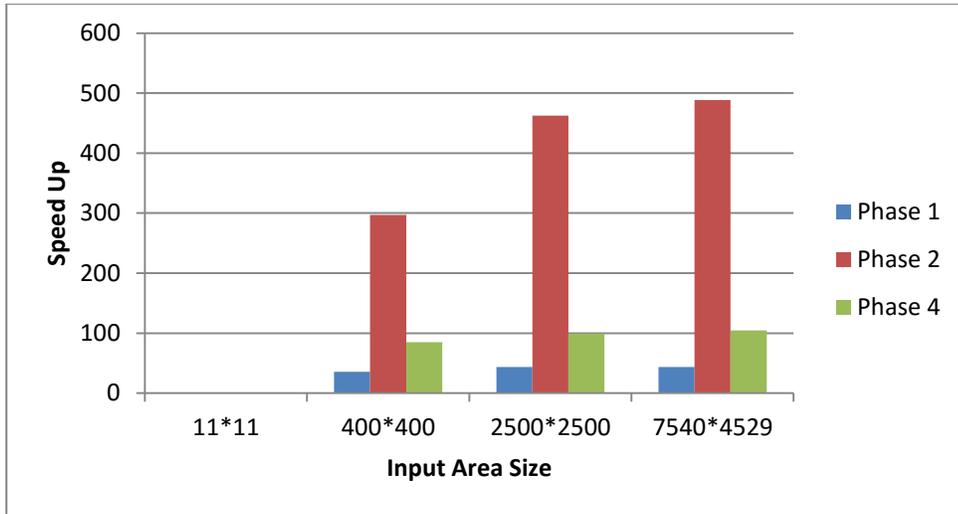


Figure 22: Speeds Up of SRM with Fine Granularity Parallelization (Slave-Titan)

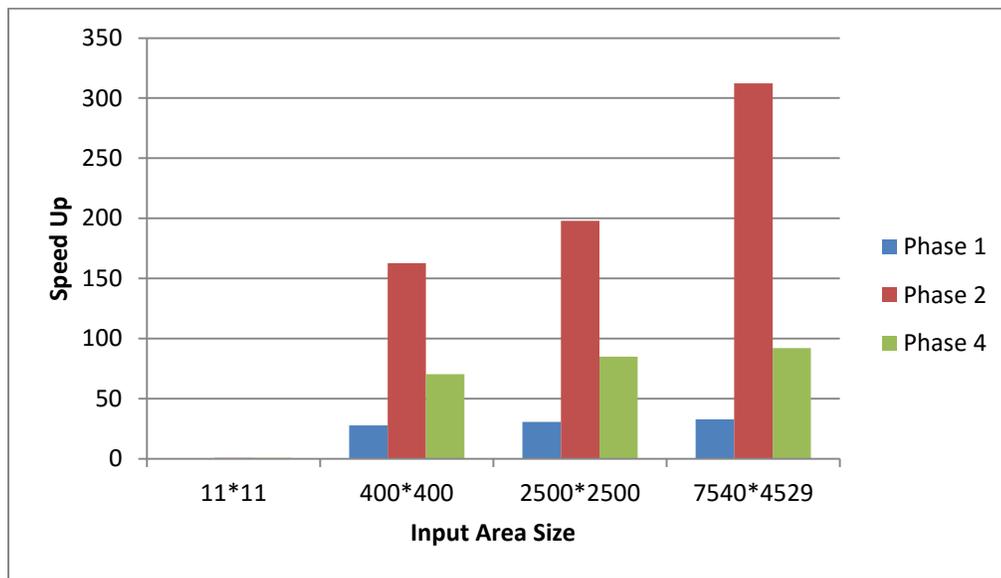


Figure 23: Speeds Up of SRM with Fine Granularity Parallelization (Slave-GTX)

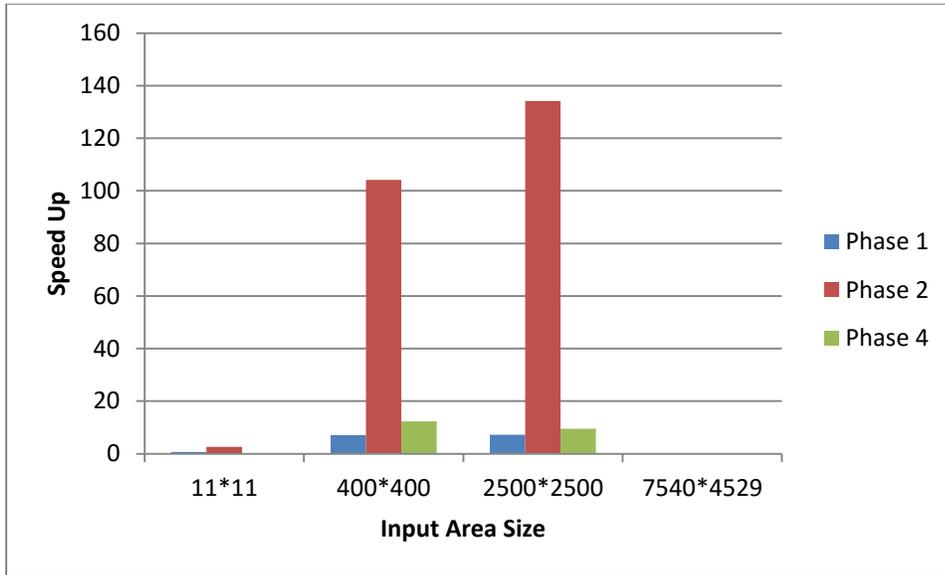


Figure 24: Speeds Up of SRM with Fine Granularity Parallelization (Slave-Tesla)

Figure 22 to Figure 24 show the improvement speed up for each module on different types of GPU illustrates the speedup for each phase with GPU implementation. Since hardware computation power differs, speed up of Titan-X is larger than that of GTX 680, which is larger than that of Tesla. It's clear that for a large area the speedup is about several hundred times. However for tiny area, such as area with 11\*11 pixels, the CPU SRM runs much faster than the GPU SRM. It can be explained that the GPU implementation introduces some overhead. It needs to transfer data in host memory to device. And executing a kernel on GPU is an expensive operation, since the kernel code needs time to be transferred to the device. In GPU, *warp* is the basic parallel processing unit. At a moment only one instruction is able to be executed in one warp. There are 32 threads in one warp, all of which must share the same one instruction at one moment. If

instructions for these 32 threads are different, warp would execute the instructions one by one until all threads finish their work. This is called control diverge, which reduces the GPU performance.

In Figure 25, we can see the performance of phase 2 with GPU improves as the area size increases. As discussed in Section 3.2, each pixel is treated as an independent thread in GPU. The thousands of streaming processors in GPU process the computation of thousands of threads in parallel, which would make a significant improvement. However, after the point of input size, the speedup remains stable, because it reaches the hardware bound. The maximum number of threads that could be run simultaneously in GPU is  $UpperBound = cudaCores * warpsPerCore * threadsPerWarp$ . In NVidia GPU,  $warpsPerCore = 32$  and  $threadsPerWarp = 8$ . The threads upper bound points for Titan-X, Tesla, and GTX 680 are 786432, 61440, and 393216 pixels, respectively. For square input area, the length of side would be  $\sqrt{786432} = 886$ ,  $\sqrt{61440} = 247$ , and  $\sqrt{393216} = 627$  pixels. When input size below that point, increasing of input can result in improvement of speedup since more hardware is available to participate in computation. While if data size exceeds that point, since hardware limit the improvement could not become larger.

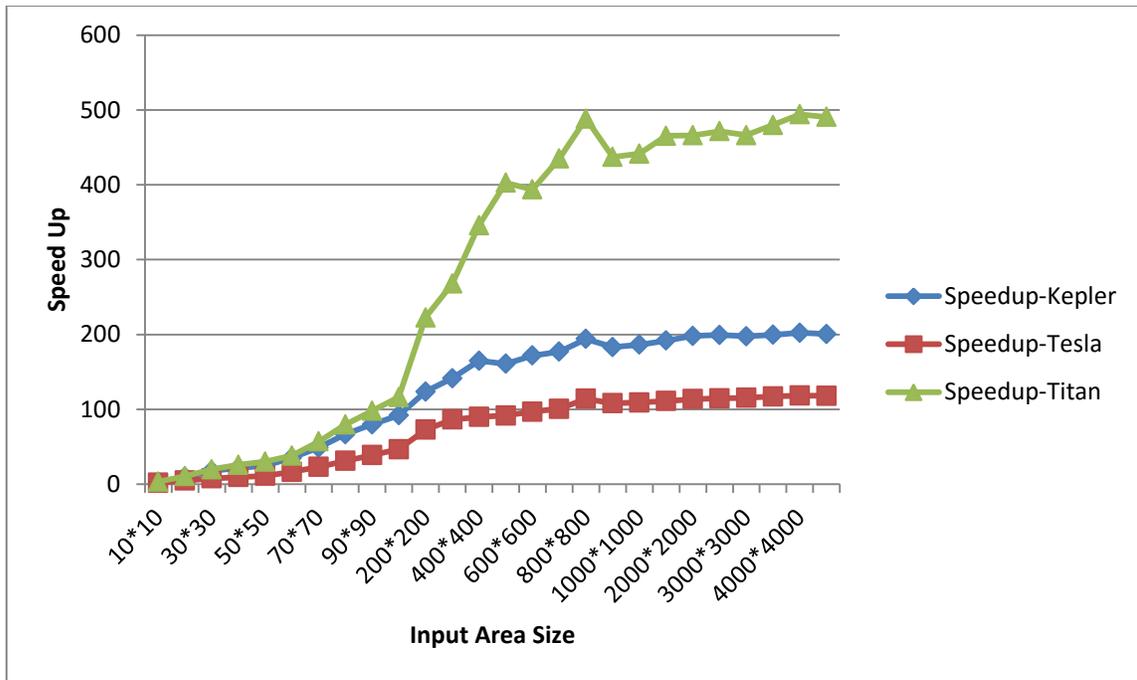


Figure 25: Speedup for Phase 2, with Different Area Size

As number of threads becomes large, overhead caused by data transferring, kernel launching and control diverge would be compressed compared to the improvement. Figure 26 shows the ratio of CPU-GPU communication with GPU computation time for different input data size. The computation time dominate the GPU process

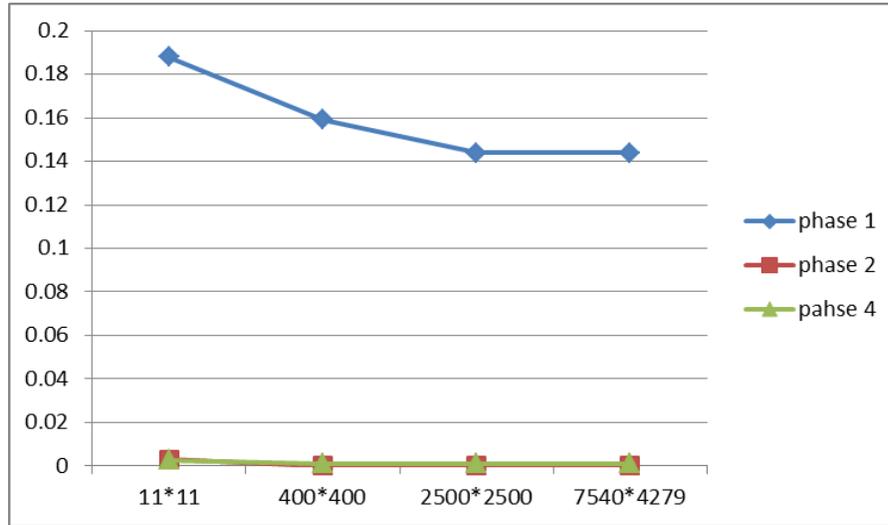


Figure 26: Ratio of GPU-CPU Communication Cost / GPU Computation Cost

Figure 27 and Figure 28 shows the output of serial code and GPU. The difference between the results from GPU implementation and the original SRM can be expressed as an error matrix, as shown in Figure 29, where we take the area size 2500\*2500 as an example. The matrix is the subtraction of matrix of original data product and the matrix of GPU data product. The magnitude of error is about  $10^{-3}$  to  $10^{-2}$ . The expectation of error is 0.0013852, and variance of error is 3.5699e-07. The relative error can be expressed in Figure 30. The expectation of relative error is 1.702e-6. And variance of error is 3.9024e-13. The error is very small compared to the magnitude of original data. Values both in serial code and parallel code are stored with single floating precision. It's the GPU hardware implementation that brought some tiny errors.

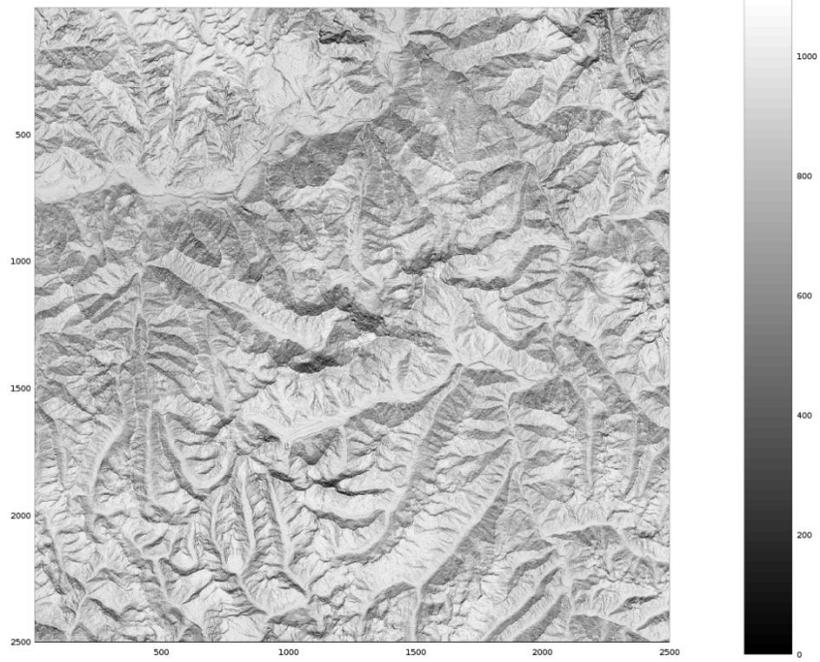


Figure 27: Data Matrix Produced by Serial SRM

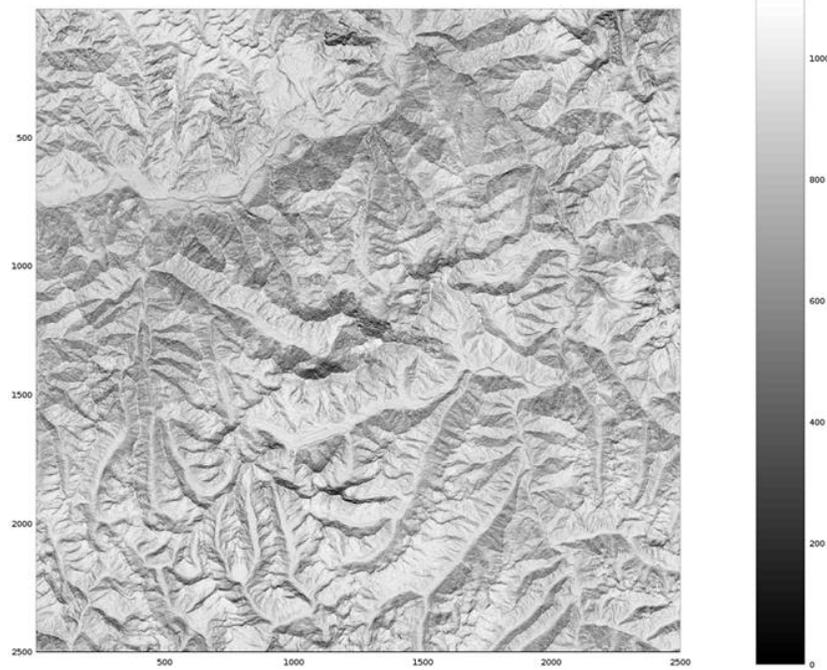


Figure 28: Data Matrix Produced by GPU

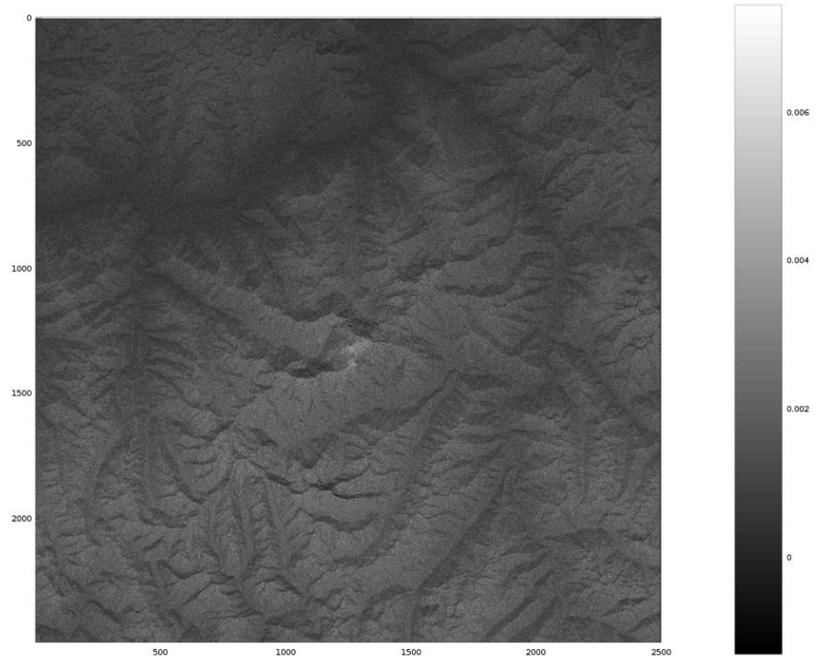


Figure 29: Absolute Error Matrix

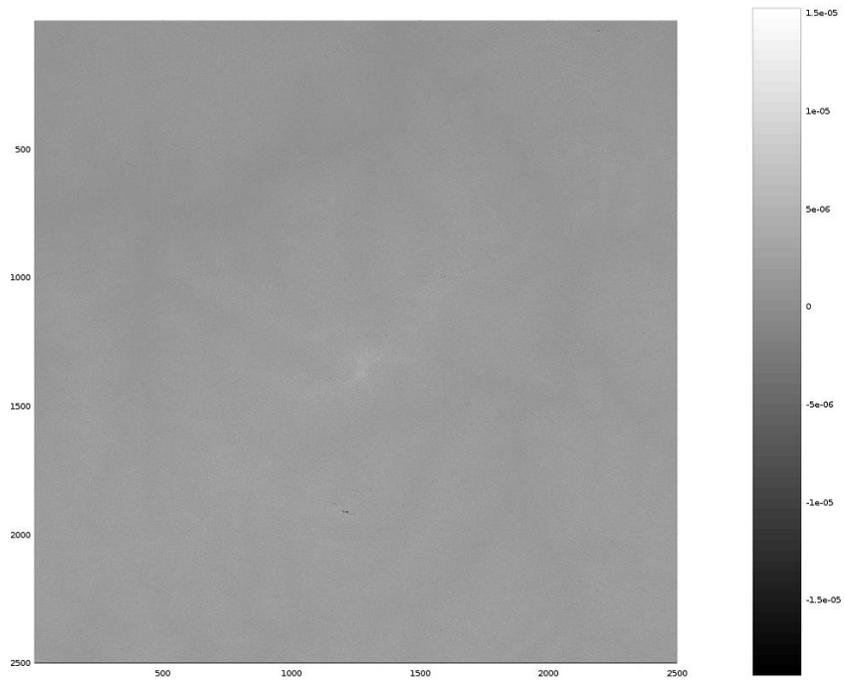


Figure 30: Relative Error Matrix

#### 4. COARSE GRANULARITY PARALLELIZATION

As shown in Figure 21, there is a job queue in parallelized SRM. Each task in queue represents one piece of area at a time. Figure 31 illustrates a simple solution to process tasks in the job queue: use one machine with GPU to process the tasks in serial.

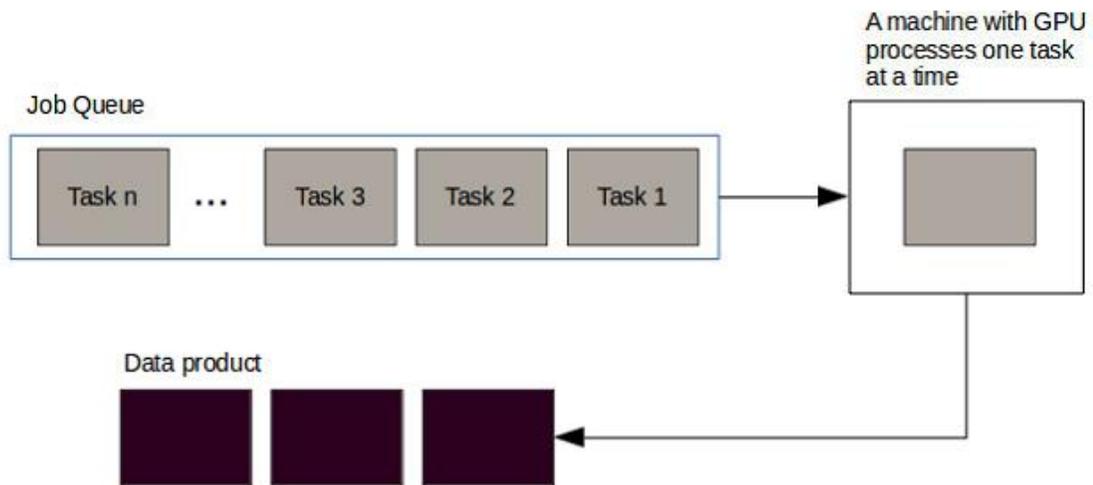


Figure 31: Simple Solution for SRM Phase 4

To accelerate the processing time for tasks in queue, a better way is to process tasks in parallel. Therefore, it's necessary to introduce heterogeneous cluster, where there are several GPU-equipped nodes (servers or desktops) which are able to process computation tasks in parallel. In this section, I will discuss the tools and the algorithm to set up the cluster.

#### 4.1 Tools: MPI and Open MP

The Message Passing Interface (MPI) is a message passing library standard. It's a specification of libraries that implement message passing functions. In my research and development, I chose MPICH libraries since documents and tutorials are abundant.

MPI was designed for parallelization at the processes level. It runs on patterns of distributed memory architecture or hybrid distributed memory system where shared memory is combined over networks. All parallelism is explicit for the programmer, who will be responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. Although MPI supports shared memory communication in hybrid architecture, explicit API is needed to use these functions. Sometimes C++ standard libraries are not supported. In order to reduce extra data structure redesigning, and to use C++ STL, Open MP will be used to implement parallelization that requires shared memory architecture. In my work, the MPI only works in distributed memory architecture pattern.

Open MP, or Open Multi-Processing, is an API that support multi-threaded, shared memory parallelism. Open MP is an abstraction of POSIX threads API, and it provides an easy programming solution. A program written with Open MP can run on most processor architectures and operating systems. It accomplishes parallelism explicitly through use of threads, which exist within processes. The underlying architecture can be described as uniform memory access. Figure 32 shows the memory model for Open MP.

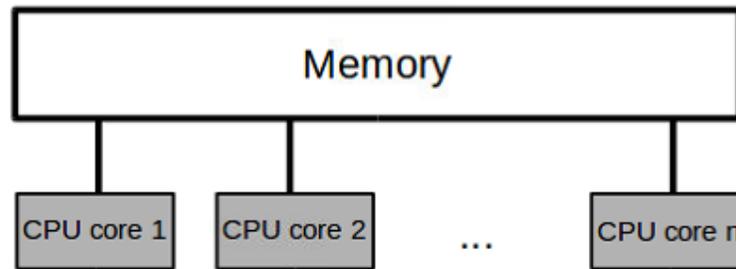


Figure 32: Open MP Shared Memory Model

#### 4.2 Coarse Granularity Parallelization Model

The architecture for our heterogeneous cluster can be described in Figure 33. The cluster has four nodes, one is master node, which is responsible for tasks assigning and result rendering. Other three slave nodes are only responsible for computing and data transferring to shared disk. The shared disk between nodes is implemented using Network File System (NFS). As data product is big, the transferring time might be long. So at slave nodes, parallelization on data transferring and task computation is necessary. Since data transferring and data computation are independent work, they can be handled simultaneously by overlapping using multi-threads or multi-processes techniques. There is one MPI process in master node, which is responsible for scheduling tasks to slaves. There is one MPI Process for each slave node, and within one process, two threads were created. Thread 2 for CUDA kernel launching, and the other Thread 1 is for data transferring. Open MP provides us a convenient shared memory programming model as show in Figure 32. Within one slave node, a piece of memory was shared by two threads, where data product was stored. Thread 2 produce the data product and push it into the

memory, while Thread 2 pop data product from the shared memory and transfer it to NFS.

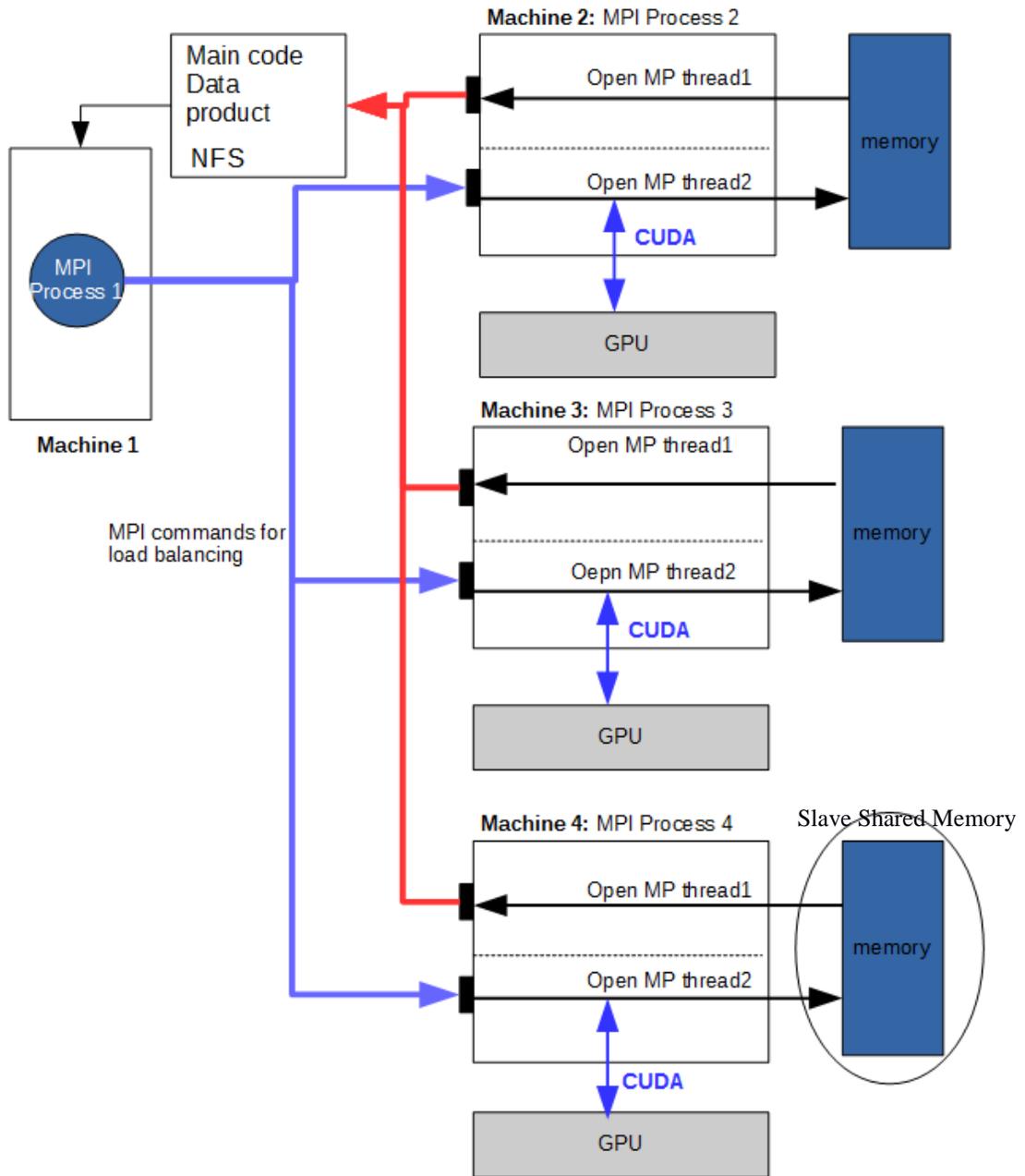


Figure 33: Heterogeneous Cluster System Model

For customized computation service, system configuration such as task size, job queue length in slave memory, numbers of GPU nodes that are needed and NFS storage space can be configured by user in master node before launching the computation service. A user command console is provided to receive the environment settings.

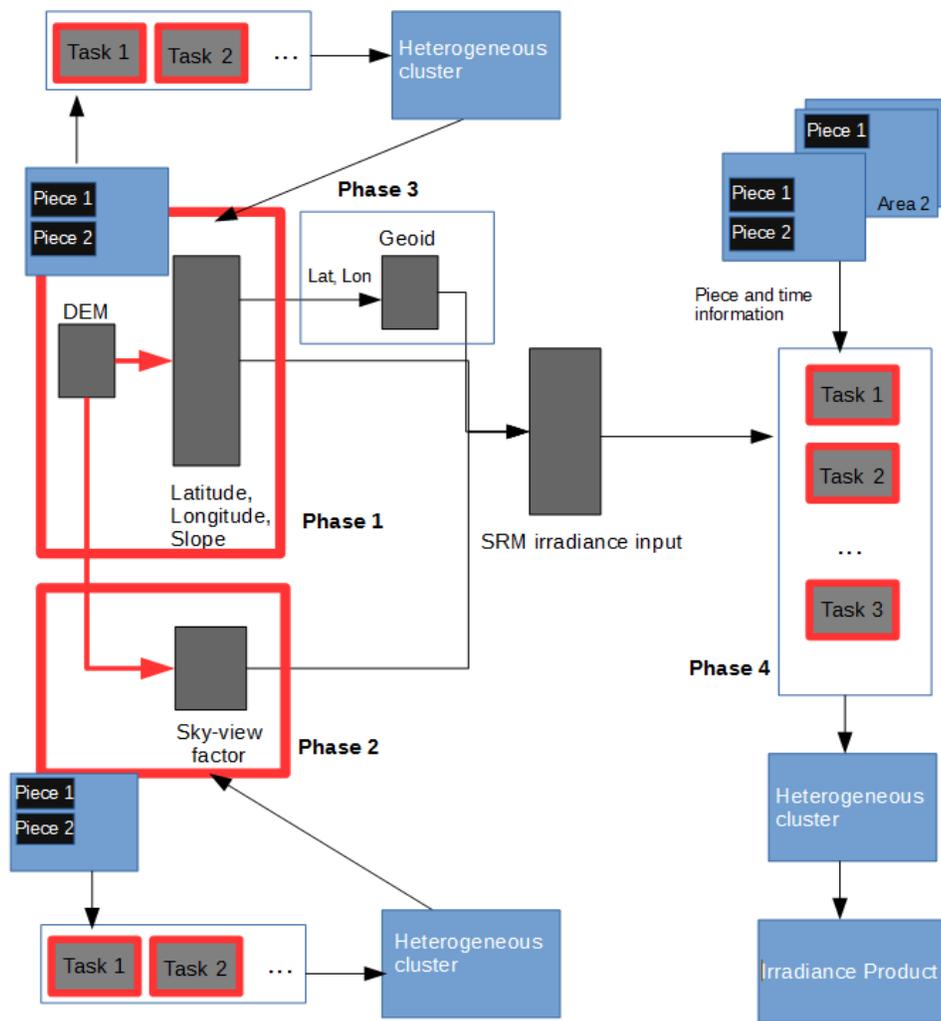


Figure 34: SRM Work Flow with GPU Cluster

Figure 34 shows the SRM work flow with implementation of heterogeneous cluster. Tasks in Phase 1, 2 and 4 are processed and completed in cluster.

### 4.3 Load Balancing and Network Overhead

As shown in Table 4, Table 6 and Table 8, different GPUs have different processing time for the same input. In the cluster, the processing time in each node varies a lot since GPU type at each node is different (Appendix B). Therefore job balancing and scheduling the tasks burden in each node is very important, because we do not want to let any of the GPU becomes idle and waste computation power.

Network transferring bandwidth is an important overhead introduced by distributed nodes, since that data is transferring among nodes. There are three kinds of data transferring: data from previous phase, data product to NFS, and task assigning overhead. Data from previous phase are needed to be transferred on network only once, which takes almost constant time. In this section, I will discuss in the SRM cluster implementation, other two kinds of data transferring overhead can be almost eliminated.

Figure 35 illustrates the structure of the load balancing and scheduling module. Jobs are divided into atomic task for each node based on the time stamp and the certain part of the focus area. At each time one node computes one part of the radiation matrix for only one time stamp. When computation is finished an acknowledgement message will be sent back to the master node. Once received the acknowledgement message, master picks the next task in queue and send the new task to the node.

In each slave node, data product produced by each node need to be uploaded to

shared disk every time. We can use multi-thread implementation to make it run in simultaneously with GPU computation. So the overhead caused by data product transfer can be overlapped. Furthermore, information contained in one task is only about the piece and time, which is extremely small, and is only 3 bytes for each task. Since that it is clear that the network bandwidth is not the system bottleneck.

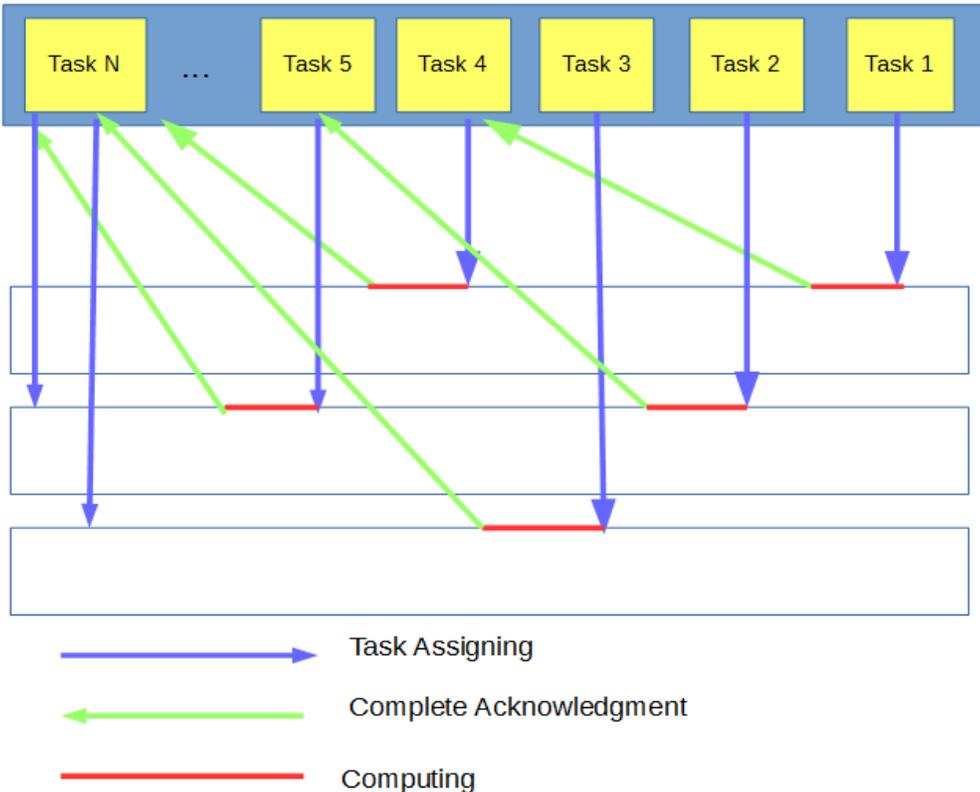


Figure 35: Loading Balancing Protocol

For master node and slave nodes, the algorithm is described as Figure 36, Figure 37, and Figure 38.

---

**Algorithm 1, Master Scheduler Process****Data:**

Task object *task*: contains job information about time and area, and a FINISH\_FLAG indicating no tasks left in master, which is initialized as false

A task vector *All\_Tasks*: store all the tasks objects

Acknowledgement object *ACK*: contain one ID that tells master which slave finishes a task.

**Result:**

Assign tasks to each node and guarantee the load in each node is balanced.

**Process:**

1: Initialize *All\_Tasks* with the input tasks whose FINISH\_FLAG is false

2: **For** all slaves from 0,1, ... I, to N, **Do**:

3:   job ← dequeue(*All\_Task*)

4:   assign slave[I] with job <http://geographiclib.sourceforge.net/>

5: **End Do**

6: **Repeat:**

7:   **If** receive *ACK* from slave I, **Do**:

8:     job ← dequeue(*All\_Task*)

9:     assign slave[I] with job

10: **End Do**

11: **Until** *All\_Task* is empty

12: **For** all slaves from 0,1, ... I, to N, **Do**:

13:   assign each slave with a *task* whose FINISH\_FLAG is set to true

14: **End Do**

**Finish**

---

Figure 36: Algorithm for Master Schedule Process

In master, a tasks queue is maintained. For each task in the queue, information of time and cutting piece is encapsulated. Typically we make one task responsible for at most 5000\*5000 (Tesla 1060C can only process at most 335539020 threads at one time). Master assigns each node with one task at first, and waiting for response. Once it knows one node finishes its task, it would assign the next available task in queue to that node.

In slave, a product pool is maintained, which is also a queue. The computation thread launch GPU kernel to complete its task and push data product into the queue, while the transferring thread pops the queue and upload data to cluster disk.

### Algorithm 2, Slave Computation Schedule Thread

---

**Data:**

Task object *task*: As above

Acknowledgement object *ACK*: As above

Data buffer object *Buffer*, the buffer stored the data product

Data product management object *Pool*: a queue that manage data product while guarantee no memory leak occurs. Details are shown in Section 5.3.2 \*\*

Pool full indicator *FULL*: FULL=true means Pool is full, vice versa \*\*

**Result:**

Waiting for orders and then complete a computation task. Stop if an FINISH order was received

**Process:**

```
1: Repeat:
2:   Listen port, If no task order incoming, block
3:   Else accept port and receive a task order task
4:   If task.FINISH_FLAG == true:
5:     Break;
6:   Else:
7:     Launch computation task with information in task , result is in Buffer
8:     While FULL:
9:       Continue
10:    End While
11:    Push (Pool, Buffer)*
12:    Send Master Scheduler Process with ACK, attached with slave node ID
13:  End If
14: End Repeat
```

**Finish**

\*Push operation will be discussed in Algorithm 4

\*\* Pool and FULL are shared by threads of computation and thread of transferring

---

Figure 37: Algorithm for Slave Computation Thread

### Algorithm 3, Slave Data Product Transferring To NFS

**Data:**

Data product management object *Pool*: As above

Data buffer object *buffer*: As above

Pool full indicator *FULL*: As above

**Result:**

Transferring data product to Master Receiver Process

**Process:**

```
1: Repeat:
2:   Buffer ← Pop (Pool)*
3:   Set FULL false
4:   Send back Buffer to NFS
5: Until Pool is empty.
```

**Finish**

\*Pop operation will be discussed in Algorithm 4

---

Figure 38: Algorithm for Slave Data Product Transferring

The shared memory model in slave nodes introduces a potential memory race risk to the system, Figure 39 illustrates the model of the process of data producing and consuming in slave node, which is the Slave Shared Memory part in Figure 33. A queue *Pool* in shared memory was used to manage the data product. The size of queue in memory is determined by the rate of producing  $R_p$  minus the rate of transferring  $R_t$  (in Bytes/sec). If  $R_p - R_t > 0$ , as time elapsing the queue will becoming more and more larger, and may cause a memory leak. If  $R_p - R_t < 0$ , the queue will maintain empty. Producing rate  $R_p$  is determined by the size of task and the capabilities of the GPU, and transferring rate  $R_t$  is determined by the bandwidth of LAN. For memory safety, it will be difficult to guarantee the difference between  $R_p$  and  $R_t$  to be less than zero since they can be affected by many factors.

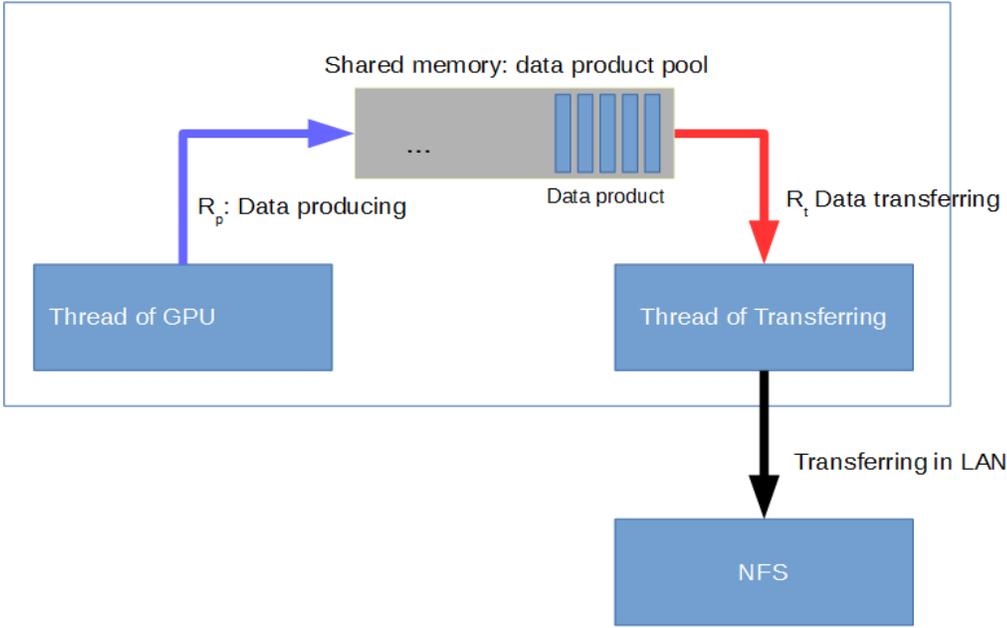


Figure 39: Shared Memory Data Pool in Slave

Figure 40 is the strategy of data management to avoid memory leak in slave node's shared memory. We keep a queue called *Pool* to store the data product from GPU. Data product for each task order from host is treated as the basic element in the queue, which is stored in main memory and called *Product Item*. There is upper bound for queue size  $N_q$ . At the very beginning, new product from GPU is pushed into queue. The transferring thread pops out the data product memory from the queue and uses it as a buffer to directly send back to NFS. If the queue size extends to  $N_q$ , the GPU computation thread will be blocked until new space is available.

---

**Algorithm 4, Slave Product Load Schedule**

---

**Data**

Data buffer object *Buffer*, the buffer stored the data product  
 Data product management object *Pool*: The queue stored data product in slave  
 Data product *Product Item*: Element in *Pool*, stored in main memory  
 Upper bound  $N_q$ : the maximum number of elements in queue.  
 Counter  $C_q$ : the current number of elements in queue.  
 Pool full indicator *FULL*: FULL=true means Pool is full, vice versa

**Operations**

Initialize  $C_q$  to zero, and FULL to false

Push(*Pool*, *Buffer*):

- 1: **If**  $C_q < N_q$
- 2:     Encapsulate *Buffer* as *Product Item*. And push into queue
- 3:      $C_q++$
- 4: **Else:**
- 5:     FULL  $\leftarrow$  true
- 6: **End If**

*Buffer*  $\leftarrow$  Pop(*Pool*)

- 1: Item  $\leftarrow$  pop out the first item in queue
  - 2: Transfer data product in memory item into *Buffer*
  - 3:  $C_m--$
  - 4: FULL  $\leftarrow$  false
- 

Figure 40: Slave Memory Management

#### 4.4 Performance

We will evaluate the coarse granularity parallelization performance based on criteria of computation cost, performance speed up in scalability, memory safety and overhead such as I/O and network latency of NFS and MPI communication cost.

The hardware specification is in Appendix B. In this cluster, we use one master node and three slave nodes with GPU of GTX 680, Tesla 1060C and Titan-X respectively. The LAN network bandwidth is 1Gygabytes/sec. For MPI communication we have set up SSH connections between each node. For NFS, we allocate 1 Terabytes in Slave-Titan hard disk, since Slave-Titan has the largest storage space among all nodes in cluster. All other nodes mount the shared disk in Slave-Titan using NFS protocol. For time cost measurement, MPI and Open MP timer API is used (Appendix C). All results come from average of ten experiments.

Let's evaluate system performance on computation cost, system latency and MPI overhead first. Table 10, Table 11, Table 12 show the average computation cost, transferring cost and scheduling cost in each node for different input piece size. There are 20 tasks in master in the experiments.

Table 10: Average Compute Cost for Each Task in Each Node (in sec)

<b>Task Size</b>	<b>Slave-Titan</b>	<b>Slave-GTX</b>	<b>Slave-Tesla</b>
<b>400*400</b>	1.02	1.19	7.25
<b>800*800</b>	3.99	4.66	28.42
<b>1600*1600</b>	15.63	18.47	113.59
<b>3200*3200</b>	62.35	73.91	453.48

Table 11: Average Data Transferring Cost for Each Task in Each Node (in sec)

<b>Task Size</b>	<b>Slave-Titan</b>	<b>Slave-GTX</b>	<b>Slave-Tesla</b>
<b>400*400</b>	0.09	0.18	0.28
<b>800*800</b>	0.37	0.49	0.90
<b>1600*1600</b>	1.47	1.83	3.53
<b>3200*3200</b>	6.16	7.07	13.62

Table 12: Average Scheduling Cost for Each Task in Each Node (in sec)

<b>Task Size</b>	<b>Slave-Titan</b>	<b>Slave-GTX</b>	<b>Slave-Tesla</b>
<b>400*400</b>	1E-5	1.8E-5	3.2E-5
<b>800*800</b>	1E-5	2.4E-5	3.5E-5
<b>1600*1600</b>	1.3E-5	2.6E-5	3.9E-5
<b>3200*3200</b>	1.3E-5	2.3E-5	3.6E-5

As illustrated in Figure 33, work balancing in our cluster is implemented by acknowledgement signal transferring via LAN. From Table 12, it can be found that the cost of scheduling is  $10^3$  less than that of data transferring and more than  $10^5$  times less than that of task computation. Therefore it's safe to ignore the scheduling overhead.

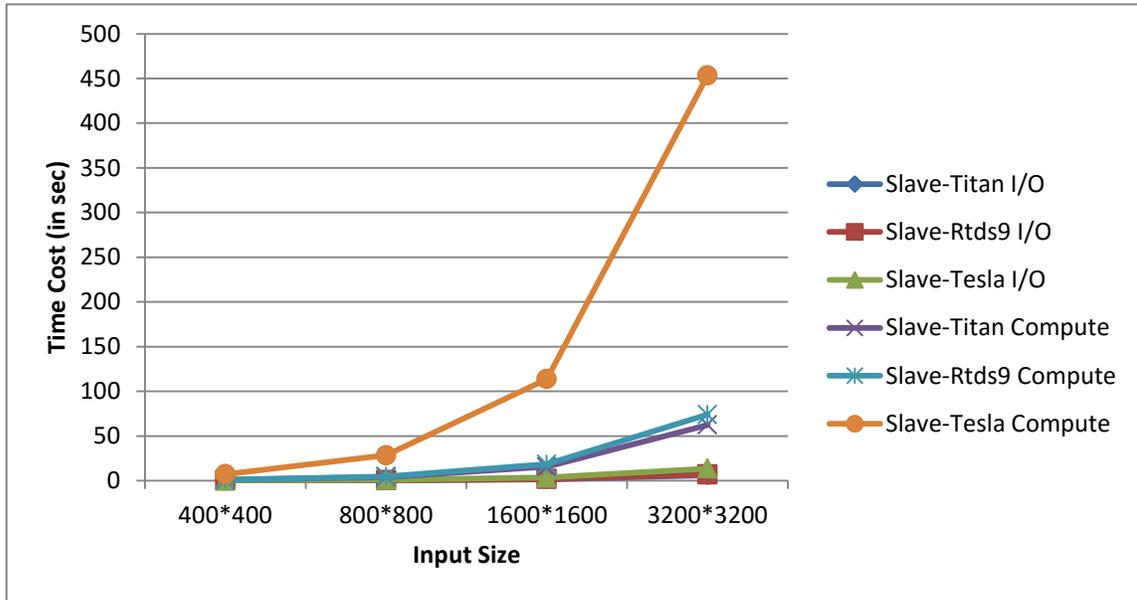


Figure 41: Cost for Compute and File System in Each Node

Figure 41 show the relation between computation cost and data transferring cost in each node. It is obvious that the in SRM the computation always dominated the process time. Since the overhead in cluster is very small. As described in Section 4.3, since thread of computation and thread of data transferring can be overlapped, if computation cost remains larger than transferring, the overhead of transferring can be eliminated.

Table 13: Average Data Product Queue Size in Each Node (Normal Network)

<b>Task Size</b>	<b>Slave-Titan</b>	<b>Slave-GTX</b>	<b>Slave-Tesla</b>
<b>400*400</b>	1	1	1
<b>800*800</b>	1	1	1
<b>1600*1600</b>	1	1	1
<b>3200*3200</b>	1	1	1

Table 14: Average Data Product Queue Size in Each Node (Simulated Congested Network)

<b>Task Size</b>	<b>Slave-Titan</b>	<b>Slave-GTX</b>	<b>Slave-Tesla</b>
<b>400*400</b>	15	15	1
<b>800*800</b>	15	10	1
<b>1600*1600</b>	10	5	1
<b>3200*3200</b>	5	1	1

For memory safety, Table 13 and Table 14 show the maximum job queue in slave node under different network environments. Table 13 shows that the queue size for slave data product results is always 1 under normal network burden. It indicates that the data transferring speed is faster the data producing speed, which further proves that overhead of network can be totally eliminated by overlapping the threads of computation and data transferring. Table 14 shows the job queue size in slave memory under the network that is congested, where data transferring speed is nearly the same with that of data producing. The queue size is guaranteed to be less than 15 in this case.

Table 15: Cluster Performance for Different Task Size (20 Tasks, in sec)

Task Size	Compete Time	Titan Time	GTX Time	Tesla Time
<b>400*400</b>	14.81	20.42	23.83	145.64
<b>800*800</b>	57.79	79.84	93.21	568.45
<b>1600*1600</b>	230.92	312.45	340.76	2271.82
<b>3200*3200</b>	921.34	1247.76	1478.54	9069.96

Table 16: Number of Tasks Processed by Each Node

Task Size	Slave-Titan	Slave-GTX	Slave-Tesla
<b>400*400</b>	10	2	8
<b>800*800</b>	10	2	8
<b>1600*100</b>	10	2	8
<b>3200*3200</b>	10	2	8

For scalability, data in Table 10 can help us make a hypothesis of the ideal performance of our cluster. Let the process rate for each node to be  $N_1$ ,  $N_2$ ,  $N_3$ . If the three nodes run simultaneously, the speed up for node 1, node 2 and node 3 should be  $\frac{N_1*N_2+N_1*N_3+N_2*N_3}{N_2*N_3}$ ,  $\frac{N_1*N_2+N_1*N_3+N_2*N_3}{N_1*N_3}$  and  $\frac{N_1*N_2+N_1*N_3+N_2*N_3}{N_2*N_1}$  respectively. Figure 42 illustrates the ideal speedup for different input size.

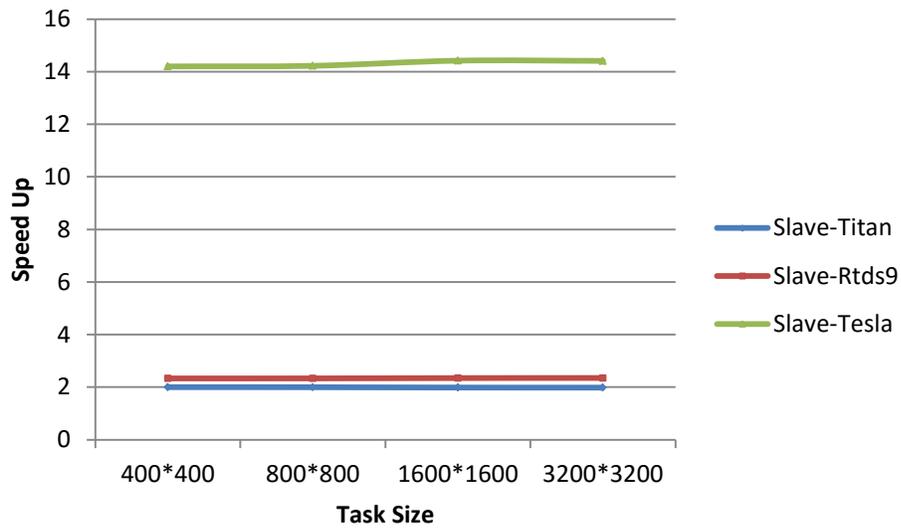


Figure 42: Ideal Relative System Speedup Observed by Individual Machines in Clusters

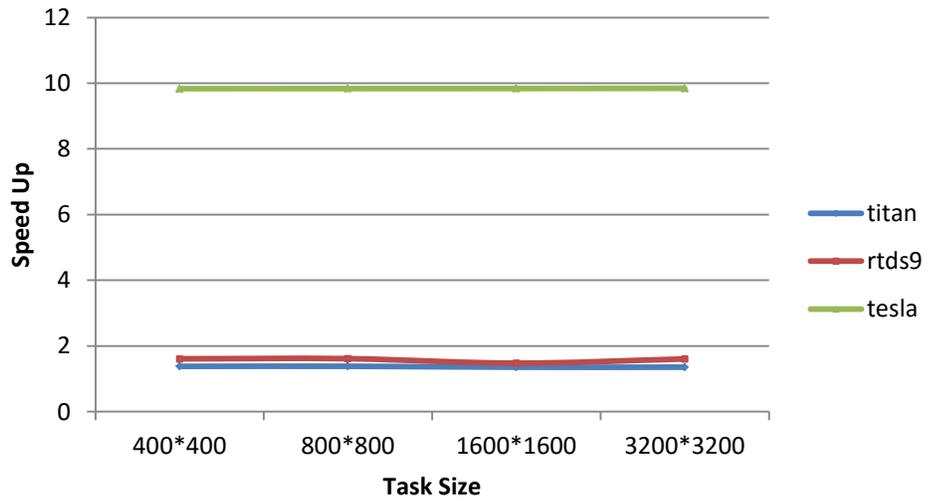


Figure 43: Practical Relative System Speedup Observed by Individual Machines in Clusters

Table 15 shows the experimental performance of our cluster. Titan Time, GTX Time and Tesla Time are the time cost of a certain node if it processes all tasks alone.

Figure 43 illustrates the speedup of the cluster compared with single node work. Compared with Figure 41, we can find the actual system speedup is less than the ideal. As discussed before, network overhead has little side effect of the system performance. The reason for the slowdown can be explained with data in Table 16, which shows the number of tasks processed by each node in the cluster, and with data in Table 10. If we take task size of 400\*400 as an example, the total computation cost would be  $10 * 1.02 = 10.2$  sec for Slave-Titan,  $8 * 1.19 = 9.52$  sec for Slave-GTX, and  $7.25 * 2 = 14.5$  sec for Slave-Tesla. Therefore Tesla would be the bottleneck of the whole system performance. When Titan and GTX finish the work, they have to wait for Tesla to finish.

As tasks numbers increases, the practical speedup would approach the ideal case. Figure 44 illustrates the speedup for different tasks number. The task size is 400\*400 in this experiment.

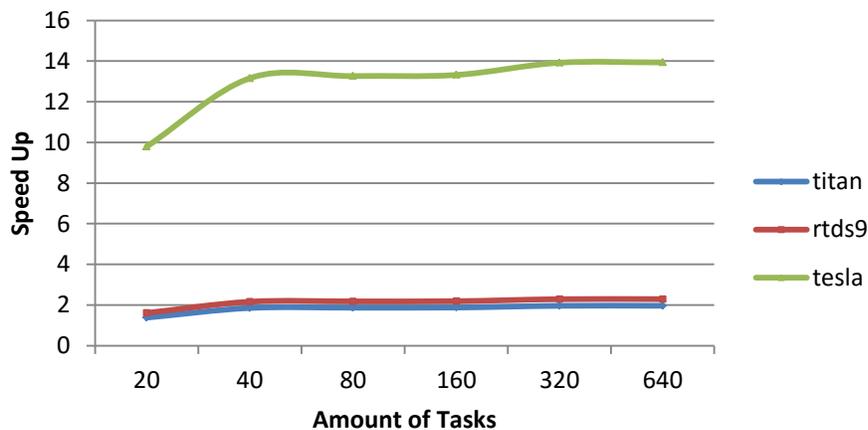


Figure 44: System Relative Speedup for Different Task Numbers Observed by Individual Machines in Clusters

The cluster overall speed up can be illustrated in Figure 45. Size of area in job queue cannot extend GPU threads upper bound. Task sizes are used as 400\*400, 600\*600, 800\*800, 2500\*2500 and 4000\*4000 pixels.

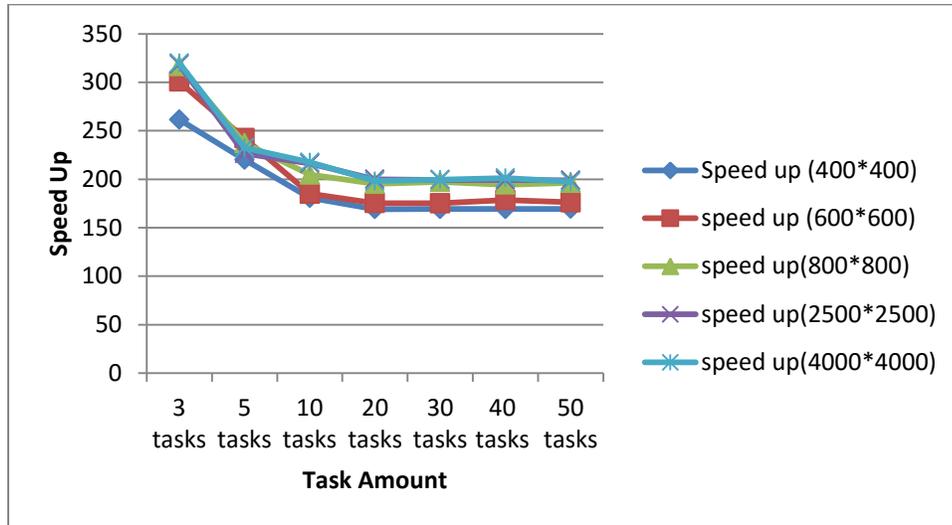


Figure 45: Overall Cluster Speed up

In each experiment with tasks amount of  $N$ , we make one task for phase 1, one task for phase 2, and  $N-2$  tasks for phase 4. Therefore when  $N$  is small, the system performance will be determined by performance of phase 2 speedup. As  $N$  grows system speedup will be dominated by performance of phase 4. Since speedup of phase 4 is smaller than that in phase 2, as shown in section 3, we can see when tasks size grows large, overall performance would be decrease and eventually stable.

## 5. IMPLEMENTATION DETAILS

With protocol of system model, in this section, I will discuss the implementation details during SRM heterogeneous distributed systems software development, which includes source code compiling, third party library loading and system setup configuration.

### 5.1 CUDA-C. MPI, Open MP Compiling and Library Loading

CUDA includes a compiler *nvcc* for development of GPU kernel device code in an extended dialect of C that supports a set of features from C++, and eliminates other language features such as recursive functions that do not map to GPU hardware capabilities. CUDA also provide a set of APIs to the host code for device management. CUDA source code file's suffix is *.cu* or *.cuh*

A simple MPI code is a combination of common C/C++ code and the MPI API. It can be compiled by the compiler *mpicc* for C or *mpic++* for C++. For more than one machines, compiled binary code must be the same and in the same file path in each node. Implicitly, each machine must have the same operating system and architecture. What's more, the connection between each node is set up using SSH protocol <sup>[19]</sup>. After code compiling and ssh configuration, we can run the code with command *mpirun*. Details about MPI configuration will be discussed in Section 5.2.

Current many main stream C/C++ compilers (such as those from GNU, IBM, Oracle, Intel, Microsoft, and LLVM) support the features of Open MP. To program with

Open MP, the header file `<omp.h>` should be included in the code. For the part of code that we want to parallelize, we need to add directives that start with `#pragma omp` in front of them to make compiler generate multi-thread codes in compiling time. Furthermore, in Makefile a command `-fopenmp` needs to be added.

```

ARCH=-gencode          arch=compute_13,code=sm_13          -gencode
arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=sm_35 -gencode arch=compute_37,code=sm_37 -gencode
arch=compute_50,code=sm_50

EXECS=run
MPICC=nvcc ${ARCH}
MPIFLAG=-I${MPI_HOME}/include -L${MPI_HOME}/lib -lmpi
OPENMP=-fopenmp
VERSION=-std=c++11
HOST_OPTION=-Xcompiler ${OPENMP} ${VERSION}

all: ${EXECS}
run: app.cpp schedule.cpp compute.cu
    ${MPICC} ${HOST_OPTION} app.cpp  schedule.cpp compute.cu -o run
${MPIFLAG}
clean:
    rm -f ${EXECS} *.o

```

Figure 46: A Makefile for Cluster Source Code

The `nvcc` compiler wrapper is more complex than the `mpic++` compiler; therefore, it is easier to make MPI code into `.cu` file and then compiles using `nvcc`. For the items wrapped by `mpic++`, we can add them explicitly as compiler configuration in Makefile. Figure 46 shows a Makefile of the heterogeneous SRM software code base, and illustrates how Open MP, CUDA and MPI were combined. The important point is to

resolve the INCLUDE and LIB path for MPI lib since *nvcc* would only find the system and CUDA libs and includes by default. Furthermore, path to GeographicLib and OpenMP settings also should be added. Since CUDA compiler *nvcc* support different GPU architecture and there are three different architectures in our cluster: Kepler, Tesla, and Maxwell, we have to indicate the architecture in Makefile for source code in each node.

## 5.2 System Configuration

Before launching the cluster, we have to take some effort to configure the system environments. The operating systems in our cluster are all Linux Ubuntu.

### Step 1: Configure Hosts File

For convenience, we want to replace IP address with machine host name in the following development. In Linux hosts file is used to map host names to IP addresses. Figure 47 is the path and contents of hosts in each node.

```
$ cat /etc/hosts
127.0.0.1    localhost
128.194.140.244  Master-Desktop
128.194.140.229  Slave-GTX
128.194.140.228  Slave-Tesla
128.194.140.217  Slave-Titan
```

Figure 47: Hosts File

### Step 2: Set Up SSH

Machines in LAN are talking over via SSH. MPI requires node is able to login to

other machines by `ssh username@hostname`, at which we will be prompted to enter the password of the machine username. To enable easier login, we generate public RSA keys and copy them to other machines. This public key authentication allows us to login to other hosts via SSH protocol without a password, and is more secure than password-base authentication <sup>[19, 20]</sup>. Figure 48 gives the commands to set up SSH. We have to set up SSH in every node in the cluster.

### **Public Key Authentication**

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
dd:02:4e:5a:cc:c9:a8:34:37:52:80:c0:29:fc:c8:e4 username@hostname
The key's randomart image is:
+--[ RSA 2048]-----+
+-----+
$ ssh-copy-id username@otherhostname
```

Figure 48: SSH Settings

### Step 3: Setting Up NFS

NFS provides a shared file system in cluster. It consists of a server and several clients. In our cluster, Slave-Titan serves as the NFS server since it has the largest available hard drive space. Figure 49 is the instructions to setup NFS. We mount the NFS shared file system to the folder `/home/liang/cloud` in each node.

```
In NFS server:
$ cat /etc/exports
/home/liang/cloud *(rw, sync, no_root_squash, no_subtree_check)
$ exportfs -a
$ sudo service nfs-kernel-server restart
In NFS client
$ mkdir cloud
$ sudo mount -t nfs Slave-Titan:/home/liang/cloud /home/liang/cloud
```

Figure 49: NFS Settings

#### Step 4: Run MPI Application

The source code folder of the SRM heterogeneous application in each node is in the NFS `/home/liang/cloud/`. Before launching the application, the hosts we use in the cluster are specified in `host_file`. Contents in `host_file` are shown in Figure 50. The machine name of master and slaves are Master-Desktop, Slave-Tesla, Slave-GTX, Slave-Titan, respectively.

```
Master-Desktop
Slave-Tesla
Slave-GTX
Slave-Titan
```

Figure 50: Host\_file

To launch the SRM cluster application, use the script in Figure 51

```
#!/usr/bin/python
import sys
import os
import subprocess

mpirun='mpirun -enable-x';
hosts = '-f host_file';
threads='2';
program_to_run='run'
sys_call='{0} -n {1} {2} ./{3}'.format(mpirun, threads, hosts, program_to_run);
print sys_call;
subprocess.call([sys_call],shell=True);
```

Figure 51: SRM Application Running Script

## 6. CONCLUSION

This thesis presents the design, implementation and evaluation of SRM running on the heterogeneous cluster. The cluster provides both fine and coarse granularity solutions for the complex model. We implemented the algorithms of tasks load balancing and translate the bottleneck of original model into CUDA kernel. The evaluation results show that the parallelization speeds up the processing time for hundreds of times.

The future work in our roadmap includes further optimizing the CUDA kernel resource management, improvement of floating precision, evaluating the algorithm performance with more nodes and GPUs, task size optimization and investigating real-time rendering solutions.

## REFERENCES

- [1] Dobрева, I. D., et al. "Development, Evaluation and Parallelization of a Spatio-Temporal, Topographic, and Spectral GIS-based Solar Radiation Model." *Proceedings of the Geocomputation Conference*, 2015.
- [2] Kirk, David. "NVIDIA CUDA software and GPU parallel computing architecture." *ISMM*. Vol. 7. 2007.
- [3] Bridges, Patrick, et al. "User's Guide to MPICH, a Portable Implementation of MPI." *Argonne National Laboratory* 9700 (1995): 60439-4801.
- [4] OpenMP, A. R. B. "OpenMP application program interface, v. 3.0." *OpenMP Architecture Review Board* (2008).
- [5] Kirk, David B., and W. Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.
- [6] Ryoo, Shane, et al. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008.
- [7] Fan, Zhe, et al. "GPU cluster for high performance computing." *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004.
- [8] Kindratenko, Volodymyr V., et al. "GPU clusters for high-performance computing." *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009.
- [9] Owens, John D., et al. "GPU computing." *Proceedings of the IEEE* 96.5 (2008): 879-899.
- [10] Chen, Long, et al. "Dynamic load balancing on single-and multi-GPU systems." *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010.
- [11] Tantawi, Asser N., and Don Towsley. "Optimal static load balancing in distributed computer systems." *Journal of the ACM (JACM)* 32.2 (1985): 445-465.
- [12] Cybenko, George. "Dynamic load balancing for distributed memory multiprocessors." *Journal of parallel and distributed computing* 7.2 (1989): 279-301.
- [13] Willebeek-LeMair, Marc H., and Anthony P. Reeves. "Strategies for dynamic load

balancing on highly parallel computers." *Parallel and Distributed Systems, IEEE Transactions on* 4.9 (1993): 979-993.

[14] Luebke, David. "CUDA: Scalable parallel programming for high-performance scientific computing." *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*. IEEE, 2008.

[15] Xiong, Qingang, et al. "Direct numerical simulation of sub-grid structures in gas–solid flow—GPU implementation of macro-scale pseudo-particle modeling." *Chemical Engineering Science* 65.19 (2010): 5356-5365.

[16] Zhao, Yong, et al. "Local acceleration in distributed geographic information processing with CUDA." *Geoinformatics, 2010 18th International Conference on*. IEEE, 2010.

[17] Howes, Lee, and David Thomas. "Efficient random number generation and application using CUDA." *GPU gems* 3 (2007): 805-830.

[18] NVidia GPU lists

[https://en.wikipedia.org/wiki/List\\_of\\_Nvidia\\_graphics\\_processing\\_units](https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units)

[19] Understand: code analysis tool. <https://scitools.com/>

[20] Doxygen. <http://www.stack.nl/~dimitri/doxygen/>

[21] GeographicLib API. <http://geographiclib.sourceforge.net/>

[22] System time. [https://en.wikipedia.org/wiki/System\\_time](https://en.wikipedia.org/wiki/System_time)

[23] C++ system clock API.

[http://www.cplusplus.com/reference/chrono/high\\_resolution\\_clock/now/](http://www.cplusplus.com/reference/chrono/high_resolution_clock/now/)

## APPENDIX

### A Experiment Input Parameters

Table A: Parameters for SRM Experiment

Resolution: 30
UTM_easting: 434040.000
UTM_northing: 4067805.000
central_meridian: 75
Hemisphere: "N"
kernel_size: 3
year: 2012
month: 8
day: 1
alpha: 1.3
beta: 0.10
GMT_offset: 5
spatial_inc: 15
seconds_start: 43200
seconds_end: 43200
atmo_profile: 2
ozone_pathlength_atmo: 2
diffuse_aerosol_model: 1

## B Hardware

Table B.1 GPU Specification

Product Name	<b>EVGA Geforce GTX Tesla C1060</b>	<b>680 classified</b>	<b>Geforce Titan X</b>
Global Memory	4096 MBytes	4096 MBytes	12288 MBytes
Constant Memory	65536 Bytes	65536 Bytes	65536 Bytes
Shared Memory per Block	49152 Bytes	16384 Bytes	49152 Bytes
Memory Bus Width	256 bits	512 bits	383 bits
Memory Clock Rate	3004 MHZ	800 MHZ	3505 MHZ
Maximum number of threads per multiprocessor	2048	1024	2048
Warp size	32	32	32
Maximum number of threads per block	1024	512	1024
Max dimension size of a thread block	(1024,1024,64)	(512,512,64)	(1024, 1024, 64)
Max dimension size of a grid size	2147483647, 65535	65535, 65535,1	2147483674, 65535,65535
Total number of registers available per block	65536	16384	65536
CUDA cores	1536	240	3072
Architecture	Kepler	Tesla	Maxwell

Table B.2 Host Specification

Nodes Name	<b>Master-Desktop</b>	<b>Slave-Tesla</b>	<b>Slave-GTX</b>	<b>Slave-Titan</b>
Mother Board	Dell 09KPNV	Dell 0D881F	GIGABYTE Z77XD3H	X99-DELUXE
CPU	Intel Xeon W3530, 2.8GHz, 4 cores	Intel Xeon E5504, 2.00GHz, 4 cores	Intel Core i7-3770K, 3.50GHz	Intel Core I7 5960X, 3.00G Hz
Memory size	8 GB	8 GB	16 GB	32GB
Memory clocks	1066 MHZ	1066 MHz	1600 MHz	280MHz
GPU	NULL	Tesla C1060	EVGA Geforce GTX 680 classified	Geforce Titan X
Bus interface with GPU	PCIE 1.0 x16	PCIE 1.0 x16	PCIE 1.0 x16	PCIE 1.0 x16

Table B.1 is the specification for GPU in cluster. Table B.2 is the host machine specification of each node.

## C. Timer

There are four kinds of timer used when evaluating the cluster performance.

For common serial code performance measurement, the C++ standard library `<chrono>` is used. This library use operating system's clock to measure epoch, which has a resolution of 1 nanosecond in UNIX <sup>[28, 29]</sup>. Figure C.1 shows an example code to measure elapsed time <sup>[29]</sup>.

For CUDA kernel code, NVIDIA provides timer API to measure CUDA kernel running time. The resolution is about 0.5 microseconds. Figure C.2 shows a typical implementation of CUDA kernel timer <sup>[30]</sup>.

```
#include <iostream>
#include <ctime>
#include <ratio>
#include <chrono>
using namespace std;
using namespace std::chrono;
int main ()
{
    using namespace std::chrono;

    high_resolution_clock::time_point t1 = high_resolution_clock::now();

    //your code here

    high_resolution_clock::time_point t2 = high_resolution_clock::now();

    duration<double> time_span = duration_cast<duration<double>>(t2 - t1);

    std::cout << "It took me " << time_span.count() << " seconds.";
    std::cout << std::endl;

    return 0;
}
```

Figure C.1 STL chrono implementation for time measurement

Timer in MPI application is implemented by the API MPI standard provides:

`MPI_Wtime()`, which computes elapsed wall-clock time since some time in the past.

Figure C.3 shows the usage of `MPI_Wtime()`. The resolution can be obtained by calling

MPI\_Wtick(). In our cluster, it is 1 nanosecond.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
//your cuda code
cudaEventRecord(stop);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

Figure C.2 NVIDIA API for time measurement

```
double starttime, endtime;
start = MPI_Wtime();
//your code
end = MPI_Wtime();
cout << "Time: " << (endtime-starttime) << " seconds" << endl;
```

Figure C.3. MPI timer

For Open MP, the API `omp_get_wtime()` is provided for time measurement in each thread. The usage is similar to that of MPI. The resolution can be gained by `omp_get_wtick()`. In our cluster, it is 1 nanosecond.