PROCESSING DEPENDENT TASKS ON A HETEROGENEOUS GPU RESOURCE ARCHITECTURE

By

RUI YANG

Bachelor of Science in Computer Science
University of Electronic science and Technology of China
Chengdu, China
2005

Master of Science in Information Security
University of Electronic science and Technology of China
Chengdu, China
2009

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
July, 2013

PROCESSING DEPENDENT TASKS ON A

HETEROGENEOUS GPU RESOURCE

ARCHITECTURE

Dissertation Approved:

Dr. Johnson Thomas

Dissertation Adviser

Dr. Subhash Kak

Dr. Blayne Mayfield

Dr. Guoliang Fan

ACKNOWLEDGEMENTS

There are many people who helped me along the way, and I would like to show my sincere gratitude. Firstly, I would like to thank my advisor Dr. Johnson Thomas for his enthusiasm, his encouragement, and his resolute dedication to the strangeness of my research which have been major driving forces thoughout my graduate career at Oklahoma State University. I would like to thank my committee members Dr. Subhask Kak, Dr. Blayne Mayfield, and Dr. Guoliang Fan for their guidance over the years. I would also like to thank members past and present of the computer science department at Oklahoma State University for their help and support.

I would like to thank my parents for their love and support during my odyssey in Oklahoma. I am eternally obliged to my wife for supporting me in everything. And I would also like to thank all my relatives in China and the United States for their help and kindness.

I would also like to thank all my friends for their encouragement, but especially Mr. Sheng Yu. He gave me valuable advice and ideas.

Once again, thank you all.

Name: RUI YANG

Date of Degree: JULY, 2013

Title of Study: PROCESSING DEPENDENT TASKS ON HETEROGENEOUS GPU RESOURCE ARCHITECTURE

Major Field: COMPUTER SCIENCE

Abstract: In this dissertation, a heterogeneous GPUs system means the system consists of a variety of different types of GPUs. Many problems in science and engineering can be represented as a two dimensional grid where updating of each grid point value is dependent on its nearest neighbor’s values. The grid size used may be too large to be handled on a single computing node. If a distributed and heterogeneous processors system is applied two crucial issues are introduced, namely, minimizing inter-processors communication and load balancing. Firstly, a novel partitioning algorithm for heterogeneous processors (NPHP) is proposed which is based on gird shape to choose an efficient way to divide blocks as square as possible to minimize communication cost. Secondly, a functional performance model with communication (FPMC) is proposed to estimate the absolute speeds of processors accurately. This method can accurately divide the workload proportional to the speeds of GPUs. Based on these two partitioning algorithms, a heterogeneous GPU system (HG) is implemented. The HG is different from other distributed GPU systems because HG can process dependent tasks which indicate the tasks in HG can communicate with each other. Furthermore, a dynamic component is designed and implement in HG system. Hence the neighbor relationship can change at run time. Using this architecture HG can deal with more complex task dependent applications. To validate our approach, a HG system running heat transfer and Gaussian Elimination is implemented. The results of experiment demonstrate that the heterogeneous GPU system has an essential advantage over traditional homogeneous GPU and CPUs system. For the static neighbor application, heat transfer, HG is at least 8 times faster than a MPI program running on CPU. For the dynamic neighbor application, Gaussian Elimination, HG can get 2.75 times speedup. Also we propose and implement some optimizations to improve performance. These include NPHP which reduces communication cost by at least 10%, and FMPC which improves the load balance by 10% on average. Optimization in the form of the data reuse technology in the computing kernel to utilize shared memory to reduce the global memory accesses yields a 7 times speedup.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

With the increasing sizes of datasets, analyzing these huge data requires novel and efficient utilization of limited computing resources, especially when dealing with high complexity scientific computing applications such as matrix multiplication, solver of partial differential equations, Gaussian emulation and so on. Researchers and scientists find it very difficult, if not impossible, to find a single super computer to process huge scientific problems. Recently, the general purpose graphics processor unit (GPGPU) has been proposed to take advantage of the single instruction multi-data (SIMD) architecture with hundred scale cores. GPGPUs can achieve hundreds of times speedup when compared to CPU. A heterogeneous GPGPUs system presents a feasible approach to handle scientific computations with vast data volume.

1.1. Dependent tasks on grid problems

Philipp Colella divided the majority of scientific computing algorithms into seven categories, the so-called seven dwarfs of parallel programming (Asanovic et al. 2006), which includes the following algorithm classes: Dense Linear Algebra, Sparse Linear Algebra, Spectral Methods, N-Body Methods, Structured Grids, Unstructured Grids, and Monte Carlo. The Structured Grids scheme is represented by a regular grid and points on the grid are conceptually updated together

(Asanovic et al. 2006). The grid may be subdivided into finer grids in the areas of interest (M. Berger and Colella 1989). AMR (Adaptive Mesh Refinement) system is one of the classic grid processing systems. There exist several AMR systems, like GrACE (Parashar 2012) (Grid Adaptive Computational Engine). When the grid is fixed and static this algorithm class includes solving partial differential equations, like the heat equation. Only the Monte Carlo method in the seven dwarfs is highly and an embarrassingly parallel computing. If we define a task as the basic computing unit in these dwarfs, then a task means a value in the matrix, a point on the grid, and a pair of key and value in hadoop(Apache 2013) which is a kind of Monte Carlo method. All tasks of these dwarfs are dependent except for the Monte Carlo method.

Many problems in science and engineering can be represented as a two or three dimensional grid where updating of each grid point value depends on its nearest neighbors' values. Crandall and Quinn comprehensively discussed this kind of problem in (Phyllis E. Crandall and Michael J. Quinn 1993). This kind of problem includes weather forecasting project like ARPS(Xue, Droegemeier, and Wong 2000), solver of Partial Differential Equation, thermal conduction, fluid dynamics and so on. The computational process of these problems consists of iterations. In iteration, values of all grid points have to update in response to the values of its nearest neighbors. Conventionally for such a problem space, the grid is too large to run on just a processor. The problem space is divided to several pieces to run on a distributed system. Usually the number of pieces equals the processors, but the sizes of pieces may vary in a heterogeneous collection of processors to ensure load balancing. Because grid points need the values from adjacent grid points, some inter-processors communications will take place. These communications feature high latency. For example, the workstation network is usually connected by a 10 Mbps Ethernet or 16 Mbps Token Ring. The cost for a single communication consists of message preparation latency and transmission time. Hence, inter-processors communication brings a bottleneck which greatly degrades the performance. Because of this cost, the right partitioning algorithm is a critical issue.

1.2. Heterogeneous processors

Because of Moore’s Law, processor speeds and communication network technology have improved and are improving substantially. The upgrade for companies or research labs is a waste of money if they just simply change the obsolete and less powerful processors to new much more powerful models and abandon the old ones, especially for the large cloud computing companies, such as Amazon. So most supercomputing centers and laboratories have to cope with heterogeneous collections of nodes. We cannot apply traditional partitioning algorithm in a heterogeneous processors environment to divide the problem space evenly since the faster processors always wait for the slower ones which is waste of resource. It is crucial to find an algorithm to reduce inter-processors communications and to achieve load balancing to maximize utilization of heterogeneous processors’ capabilities. This will yield the sub-optimum performance.

Based on the capability of processors, these partitioning algorithms can be categorized into two groups, homogeneous and heterogeneous processors. For homogeneous processors the main challenge is dividing the grid to sub-grids which have an even workload and minimize communication cost. Then a sub-grid is assigned to a processor. The basic way for partitioning a 2-D grid is to find the two largest numbers which when multiplied together equals the number of processors and divide by a larger number along the larger sides and a smaller number along the smaller side. Nicol (Nicol 1994) proposed a rectilinear partitioning algorithm, Berger and Bokhari (M. J. Berger and Bokhari 1987) introduced an adaptive method, and Belkhale and Banerjee (Belkhale and Banerjee 1990) proposed a recursive partitioning method addressing partitioning a non-uniform grid across homogeneous processors. If the grid is uniform, the partitioning is straightforward as the goal is to make the size of each sub-grid equal and their perimeters as short as possible. Equally sub-grids mean each processor is of the same size and the shorter the perimeter the smaller the inter-processors communications.

When the computation is running on heterogeneous processors, the partitioning becomes more complicated. Even for a simple linear algebra kernel as matrix multiplication on heterogeneous processors the problem of optimal partitioning has proved NP-complete (Beaumont et al. 2000). The partitioning algorithms (Phyllis E. Crandall and Michael J. Quinn 1993), (Bowen, Nikolaou, and Ghafoor 1992),(Nedeljkovic and M J Quinn 1992), (P E Crandall and M J Quinn 1993), and (Dovolnov, A Kalinov, and Klimov 2003) aim to make the size of a sub-grid appropriate to the capability of a processor as well as ensure the perimeters is short as possible to minimize the communication cost. Crandall and Quinn (P E Crandall and M J Quinn 1994) furthered their work (P E Crandall and M J Quinn 1993) in proposing the partial-homogeneous parallel algorithm that took advantage of any available processor homogeneity. One of the big deficiencies of all the existing work is that they assumed the grid is square. However, not all problems can be mapped to a square grid space.

1.3. General purpose graphic processing unit

Because of the limited performance of single-core CPUs (M. Ekman, F. Warg 2005) and the increasing programmability and performance of the graphics processor unit (GPU), more and more General Purpose GPUs (GPGPU) have been utilized in scientific and commercial computing areas in recent years. OpenCL (Khronos 2012) and CUDA (Nvidia 2012a) are two parallel computing program packages or architectures of GPU, released by Apple Inc. and Nvidia, respectively. Right now OpenCL is a 1.2 version and CUDA has been updated to 5.0 version. Both of them are very easy to learn and use. For example, to program CUDA GPUs, we use a language known as CUDA C. CUDA C is essentially C with a handful of extensions to allow programming of massively parallel machines like NVIDIA GPUs (Sanders and Kandrot 2010). Most importantly the performance of GPU demonstrates a hundred times speedup (Farivar et al. 2009) than CPU in a broad variety of applications.

GPU has already entered many cores era rather than the multicore era of CPU. A CPU in mainframes or workstation computers such as Intel i7-39xxX has only 6 cores, while Nvidia Tesla C2050 (Nvidia 2012c) has 448 CUDA cores and the latest Nvidia Tesla M2090 has up to 512 CUDA cores. Even more significantly, GPU owns a higher memory bandwidth of up to 144GB/sec (Nvidia 2012c) compared to around 20GB/sec for a CPU. In short, a GPU has a much higher FLOPS (floating point operations per seconds) of up to 1.03 Tflops whereas the most powerful CPU in a workstation is the Intel Core i7 X980 (4515MHz) which can run at 20 Gflops. Hence in recent years, more and more researchers have utilized GPUs to build their applications and platforms. Harvard Engineering, Harvard Medical School, and Brigham & Women's Hospital have teamed up to use GPUs to simulate blood flow and identify hidden arterial plaque without invasive imaging techniques or exploratory surgery (Nvidia 2013b). A molecular simulation called NAMD (nanoscale molecular dynamics) gets a large performance boost with GPUs(Nvidia 2012a). In addition, many researchers have proposed implementing existing cloud computing models like MapReduce on GPU systems (Farivar et al. 2009; He et al. 2008; Hong et al. 2010). It’s expected that GPUs will become the major player in leading edge research in fields such as bioinformatics, data mining, graph computing, and even in astronomy.

However, researchers or software developers who want to use GPGPUs (General Purpose GPUs) to analyze big data should understand their complex hardware architecture because they are significantly different from the CPU. Firstly, SM (streaming multiprocessor) is the basic process unit in the GPU and it executes threads in groups of 32 threads called a warp. So the 32 threads in one warp execute the same code path and access memory in the adjacent address to improve performance (Nvidia 2012b). GPGPUs also use a different hierarchical memory architecture. For example, CUDA uses three levels. The inner level consists of registers and local memory which can only be accessed by the thread that is using the registers and local memory. The intermediate level is shared memory which can be accessed by threads in one block. The outside level is global, constant and texture memory which can be accessed by all threads. Software developers should therefore be very careful on when to invoke communication between threads. Ideally the threads in the same block of shared memory should be involved since this will be much faster than global memory. Hence it's very difficult for researchers and scientists without computer science programming knowledge to build their own single or distributed GPU test-beds. Some kinds of universal platforms which can hide the complex architecture of the GPU are needed to satisfy the ever increasing requirements.

Furthermore, the development of GPU is very fast and we can find Moore's Law in the GPU area. For example, if we make comparisons between NV40 published in 2004 August, G70 published in 2005 July, and G71 published in 2006 March, we observe that the performance of GPUs is improving nearly two times a year(Govindaraju et al. 2006). Hence, for companies or research labs, they cannot just simply change the old and less powerful GPUs to new more powerful GPU models and abandon the old ones, especially for large cloud computing companies, such as Amazon. Recently we can rent NVIDA Tesla M2050 GPU cluster on Amazon EC2. However NVIDA introduced Tesla M2090 GPU in 2011 which performance is four time faster than M2050. It's impossible for Amazon to change all the Tesla M2050s to Tesla M2090s and not to use the M2050s anymore. In the nearly future, it is expected that we will enable to rent Tesla M2050s and Tesla M2090s or later models from Amazon EC2. The key point is how to leverage the heterogeneous GPU resources to build cloud computing systems. Maximum and efficient utilization of limited computing resources is the only feasible solution, in the absence of more resources.

1.4. Contributions

Nowadays, most platforms based on GPUs focus on specific applications or independent task models or highly and embarrassingly parallel computing models like MapReduce in which tasks do not need to communicate with each other. Hence, if we want to use these platforms to deal with dependent task models like ARPS(Xue et al. 2000) or solver of Partial Differential Equations (PDE) in which tasks have to exchange information it become very complex and even impossible.

This work addresses the main challenges in executing scientific applications on GPUs:

1. reducing internet communication between GPUs is addressed in chapter 3,
2. achieving load balancing to maximize utilization of heterogeneous GPUs resources is addressed in chapter 4,
3. improving CUDA device memory accesses addressed by data reuse is discussed in chapter 5.6.
4. implementing a general platform to support fixed and dynamic task dependent applications is described in chapter 5.

In this contribution we propose a novel platform, Heterogeneous GPUs (HG) that leverages heterogeneous GPU resources to process task dependent applications, such as heat transfer and Gaussian Elimination. This platform utilizes the heterogeneous GPUs power and partitions jobs according to their capabilities. In this work we have implemented and provide a set of APIs. Researchers can use these APIs to implement their own task dependent applications.

CHAPTER II

RELATED WORK

Because HG (heterogeneous GPUs) system includes different types of GPUs which have different computing resources such as speed and memory capacity, and even communication latency if they are not in the same communication environment, The HG system is firstly concerned with how to partition the workload. There is little research work done in heterogeneous GPUs systems.

2.1. Partitioning data on heterogeneous processors

The challenge of facing partitioning algorithms on heterogeneous processors is load balancing and minimizing communication cost if tasks in the application are dependent. Most partitioning algorithms on heterogeneous processors firstly get the absolute speeds of processors by benchmark experiments in advance or assume them as some constant positive numbers. Hence, the kernel of a partitioning algorithm on heterogeneous processors is to find a way to estimate the performance of processors accurately. Basically, there are two ways for estimating the performance; one is the traditional constant performance models (CPMs)(Beaumont et al. 2000; Alexey Kalinov and Alexey Lastovetsky 2001) that is characterized by a positive constant which is measured by the processor's hardware configurations such as CPU clock rate; the other one is the functional performance model (FPM)(Alexey Lastovetsky and Ravi Reddy 2010) that it is characterized by speed functions.

(CPMs) proved to be accurate enough for heterogeneous distributed memory systems if the performance of the application has a linearly increasing relationship to the speed of processors. However if the performance of the application doesn't only depend on the speed of processors but also on memory speed and internet communication latency, CPMs become less accurate.

FPM uses a function to estimate the performance of processors which involves parameters as much as possible on which the performance depends. It can be formulated as follows (Alexey Lastovetsky and Ravi Reddy 2010): assume we have n independent chunks of computations, each of equal size (i.e., each requiring the same amount of work). How can we assign these chunks to p $(p < n)$ physical processors $P_1, P_2, \dots, P_p$ with their respective full FPMs represented by speed functions $S_1(n_1), S_2(n_2), \dots, S_p(n_p)$ where $\sum_{i=0}^{p} n_i = n$, so that the workload is best balanced? For example, the paper(Alexey Lastovetsky and Ravi Reddy 2010) proposed a heuristic algorithm with a complexity of *O(p×log2n)* to partition the workload.

All the previous works focused on independent tasks running on CPUs which do not consider the communication cost of inter-processors. The inter-processors communication cost of GPUs is larger than CPUs because data is not only transferred within the network but also on the system bus between GPU memory and main memory. We propose a novel partitioning algorithm based on gird shape to minimize the communication cost. We also propose a FPM algorithm that considers communication costs to estimate the absolute speeds of processors accurately when processing dependent tasks.

2.2. Programming in CUDA

CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)

(Nvidia 2013c). Since its first version released in 2006, CUDA has been supported by over 300 million devices including notebooks, workstation, compute clusters, and supercomputers and has been widely used though thousands of applications and published in many academic research papers (Nvidia 2013a). The CUDA programming language is very easy to learn and is very similar to C(Farber 2011; Nvidia 2011). Optimizing CUDA program performance can be found in (Kirk and Hwu 2010).

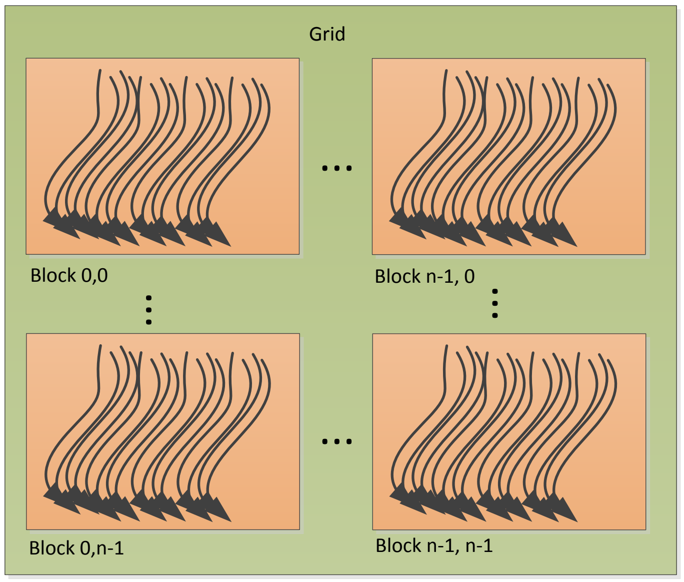


Figure 2-1. CUDA thread organization

Usually a program in CUDA devices features millions of threads. A CUDA device supports up to thousands of thread to run simultaneously. CUDA uses a 2 level hierarchy to organize threads as shown in figure 2-1. Threads are grouped by blocks. The threads in a block can be organized as one, two, or three-dimensional space. The blocks can be organized in either one or two-dimension space. So if we want to locate a specific thread, we first locate the block using block location information *blockX* and *blockY* to find the block that has the thread and then use the thread location information *ThreadX*, *ThreadY*, and *ThreadZ* to locate the thread in the block. All the blocks constitute a grid which is like a process in a CPU.

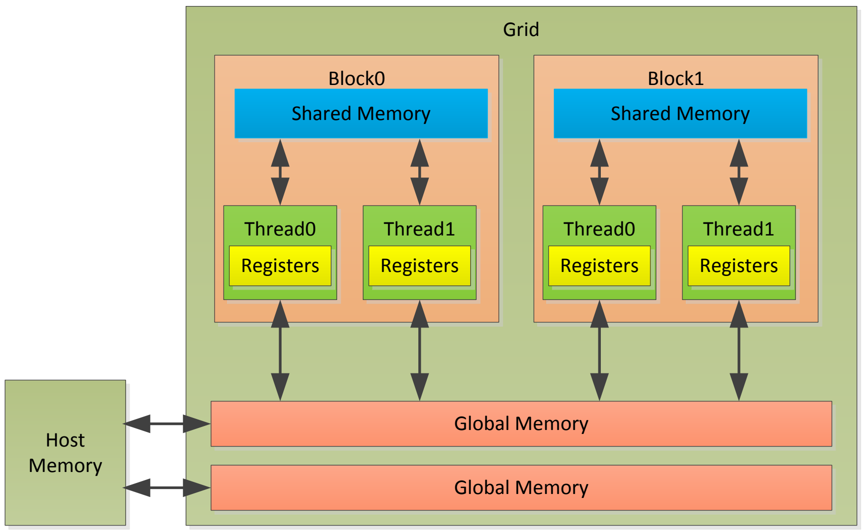


Figure 2-2. CUDA memory architecture

From the view of data access scope, the memory of GPU has 3 levels as shown in figure 2-2. From the outer to inner memory layers the sequence is global and constant memory, shared memory, and registers. Only the global and constant memory can communicate with host memory. Hence the data copied from the host needs to be transferred to the global memory or constant memory first. The global and constant memory can be accessed by all threads. The threads in a block can communicate with each other through a shared memory. However if the threads are in a different block, they cannot access the same shared memory. A thread has its own registers which can be accessed by the thread itself.

Table 2-1. The features of GPU memory

Type	Scope	Size	Performance
Global memory	Grid	Several gigabytes	Long-latency
Constant memory	Grid and read only	Tens of kilo bytes	Short-latency and high-bandwidth
Shared memory	Block		
Register	Thread	Tens of thousand	Very high speed

From a performance view, the memory of GPU has 4 types shown in table 2-1. The global memory is the largest. The memory size of a common commercial GPU can reach up to several gigabytes. However the disadvantage is that global memory access needs hundreds of GPU clock cycles. The constant memory is faster than the global. However its size is much smaller than the global and it's immutable within the GPU computation. The critical memory resource of GPU is shared memory and registers. Both of them are very fast and their sizes are small. The shared memory has tens of kilo bytes and the size for each block should be evenly divided by the number of blocks. Usually there are tens of thousands of registers on a GPU. However the number of registers for each thread is evenly divided by the number of threads. The size of shared memory for each block and the number of registers for each thread is relative small. So if there are a huge number of threads involved, the programmer should very carefully use the shared memory and registers to avoid them becoming out of range.

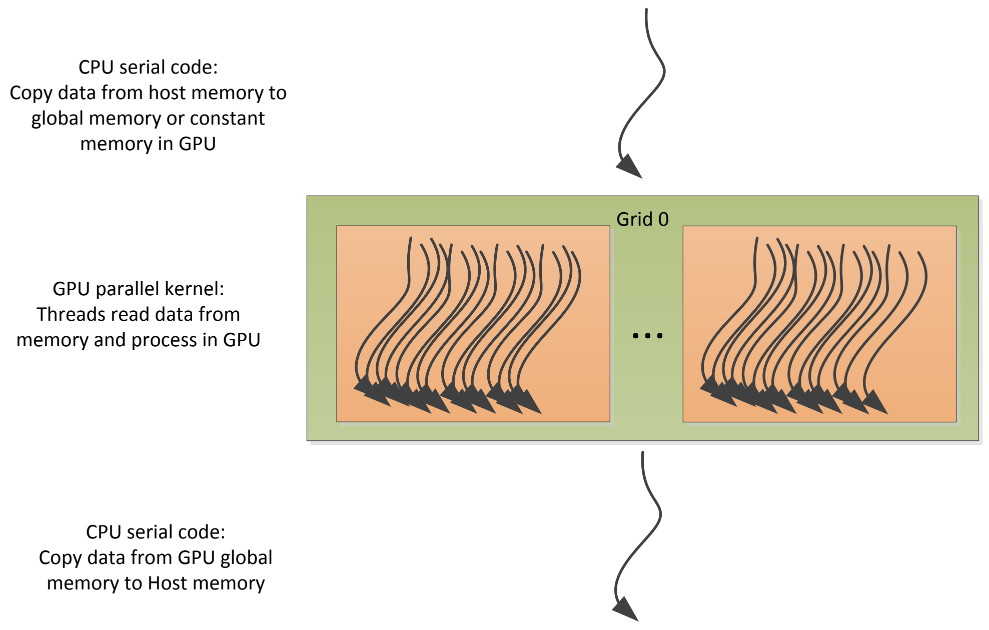

CPU serial code:
Copy data from host memory to global memory or constant memory in GPU
Grid 0
GPU parallel kernel:
Threads read data from memory and process in GPU
CPU serial code:
Copy data from GPU global memory to Host memory

Figure 2-3. CUDA execution sequence

The CUDA program is running on both CPU and GPU. There is no CUDA program that is just running on the GPU. The entrance of CUDA program is on CPU and CPU starts GPU to process data. The execution sequence of a CUDA program is shown in figure 2-3. Firstly, CPU gets or generates data and copies them from the host memory to the global memory in GPU. Secondly, CPU starts GPU and calls the GPU kernel function which is running on all threads simultaneously. Finally, when the GPU finishes, CPU copies the results from GPU global memory to the host memory.

The main difference of programming in CPU and CUDA is that there is no automatic optimization of memory access. So the programmer should know the architecture and features of the CUDA device memory including the specific memory type used by CUDA codes to improve device performance. Usually the way to take advantage of the memory hierarchy architecture is for threads to read data from the global memory to the shared memory or registers first. The threads in the same block read data from the shared memory or registers instead of the global memory. The idea is to use lower level memory as much as possible and store data in faster memory so that if the data is access again only the faster memory is searched. This technology is defined as data reuse which is described in section 5.6.

2.3. Processing dependent tasks model in GPU

This type of computing model features a structured grid (Asanovic et al. 2008) and task dependence that can include stencil, matrix computations and so on. Stencil computations solve partial differential equations (PDEs) over multi-dimensional Cartesian grids, such as weather and seismic waves (Maruyama et al. 2011). Figure 2-4 shows an example of a grid and grid point. We described the dependent tasks model in our previous work (R. Yang and Thomas 2012).

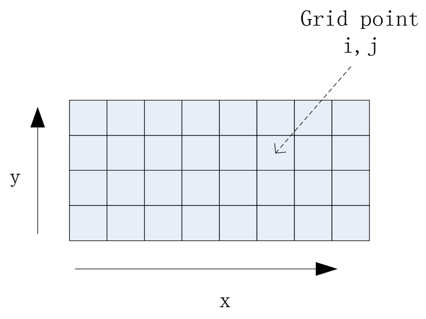


Figure 2-4. 2 dimensional grid

In stencil computations, each grid point is repeatedly updated by only using neighborhood points (Maruyama et al. 2011). However in Gaussian Elimination when it is working on column i, each grid point is influenced by the pivot row which has the maximum absolute value in column i. In other words, the neighbors of the grid point in stencil computations are fixed, surrounding and immediate. In Gaussian Elimination, the neighbors are dynamic and not immediate.

From these two types of computation models, the dependent structured grid tasks computation pattern uses neighbors to calculate the value or output of a grid point based on the neighbors it is dependent upon. The grid model we use in this work consists of a 2-dimensional Cartesian or space grid for each time unit. Hence this becomes a 3-dimensional grid with time on one axis and x, y Cartesian coordinates in the other two dimensions (see Figure 2-5). This model can be extended to n-dimensions. We also use a 2-dimensional model where one dimension is space and the other is time.

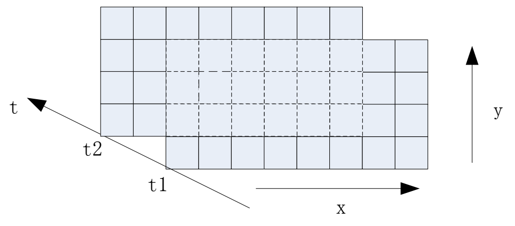


Figure 2-5. 2 dimensional grid with time coordinate

Each grid's value or output can be computed by the formula below:

$$Vp_{x,y}^{t(n)} = f(Vp_{x,y}^{t(n-1)}, Vp_{neighbors}^{t(n-1)}, Vp_{new}) \tag{2-1}$$

In (2-1), $Vp_{x,y}^{t(n)}$ is the value of grid point P in time step *n* at spatial location *x*, *y*. $Vp_{x,y}^{t(n-1)}$ is the value of grid point *P* in time step *n-1*. $Vp_{neighbors}^{t(n-1)}$ is a set of values of grid point P's neighbors in time step *n-1*. Neighbors are application dependent and can be immediate or distant neighbors. Vp_{new} represents new information to be added in time step *n* such as new data from the radar in ARPS. f represents a function or formula to compute $Vp_{x,y}^{t(n)}$.

This data process pattern is different from the MapReduce or similar models in which tasks are independent. Currently GPU systems focus on tasks which are independent. Hence existing GPU systems which use the MapReduce model cannot be directly utilized in tasks that feature a dependent structured grid model. Additional, the grids of dependent structured grid tasks model are uniform where the grid size or number of grid points is fixed. Each grid has the same field structure: index, key, value, and neighbors where an index is the unique identifier of a grid point for the system to locate a grid point, a pair of key and value is the content of the grid point and neighbors is the set of indexes of neighbors. A key is a hashed key for a user to locate a value. GrACE (Grid Adaptive Computational Engine) (Parashar 2012) uses CPUs to deal with adaptive mesh-refinement computations. In GrACE, the grids with high solution error and requiring additional resolution need to be refined. Hence the grids in GrACE are not uniform, that is, the grid size may change.

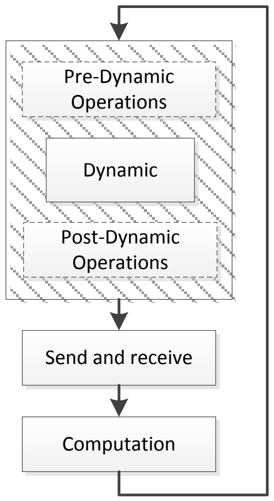


Figure 2-6. The execution sequence of dependent tasks model based on the traditional CPUs system

The dependent tasks model is categorized into fixed and dynamic dependence. In fixed dependence, the dependence of all of the points in a grid is fixed after initialization. So the process in a GPU is simple. For a computational iteration, GPU copies the values to be sent to the CPU and CPU sends and receives them and then CPU transfers the received data to GPU. GPU starts the kernel to compute. The process is presented in Figure 2-6 ignoring the slash zone. The slash zone includes pre-Dynamic Operations, Dynamic, and Post-Dynamic Operations processes.

In dynamic dependence, the dependence of all of the points in a grid is changed while the system is running. In Figure 2-6, the slash zone represents the dynamic operations where the dependence of points in a grid is changed. For example, in the Gaussian Elimination, before the *m*th computation step the pivot should be selected which has the maximum absolute value in the *m*th column and switch the pivot row and the *m*th row. The selection of pivot and the row exchange are the pre-Dynamic operations in Figure 2-6. Then use (2-3) to update the point value. We define the dependence relationships as neighbors of points change between iterations. These changes are defined as dynamic in this work. In Gaussian Elimination, there are no Post-Dynamic operations. Hence the dash squares representing pre-Dynamic and Post-Dynamic operations are optional. Dynamic operations are not required for some applications like heat transfer because the neighbors of a point in heat transfer are surrounding, immediate, and fixed.

The heat transfer application (Barney 2012) is used (2-2) to compute the value of grid point i, j at time *t+1* as follows:

$$T_{t+1}(i,j) = T_t(i,j) + ax[T_t(i+1,j) + T_t(i-1,j) - 2T_t(i,j)\,] + ay[T_t(i,j+1) + \ T_t(i,j-1) - \ 2T_t(i,j)] \quad (2\text{-}2)$$

where the temperature at time *t* plus one time step ($T_{t+1}(i,j)$) at grid point *i*, *j*, is a function of the current temperature distribution about i, j, and the thermal diffusivity in the *x* and *y* directions (*ax*, *ay*). More details of the heat transfer process in can be found in (Barney 2012).

In Gaussian Elimination, the computation consists of several iterations. In an iteration, this scheme processes a row. Hence for a *n* row matrix, it needs *n* iterations. While computing the value of a point at the *m*th iteration, the computing unit $a_{i,j}$ has to get the pivot value $a_{x,y}$, the computing unit $a_{x,j}$ which is in the same row of the pivot, and $a_{i,y}$ which is in the same column of the pivot. The pivot has the max absolute value in the *m*th column. If the computing unit $a_{i,j}$, *i* is not selected as pivot and *j* is larger than *m* at the *m*th iteration, then use (2-3) to update the point $a_{i,j}$.

$$a_{i,j} = a_{i,j} - \frac{a_{m,j} \times a_{i,m}}{a_{m,m}}, if\ i > m\ and\ j > m \tag{2-3}$$

where the value of point $a_{i,j}$ is dependent on $a_{m,j}$, $a_{i,m}$, and $a_{m,m}$.

CHAPTER III

PARTITIONING ALGORITHMS OF CONSTANT PERFORMANCE MODEL

The constant performance model assumes the performance of processors including CPU and GPU is a positive number. The goal of a partitioning algorithm in the system includes: load balancing and minimal inter-processors communication. Because the system runs on a heterogeneous GPUs environment the load balance mechanism should consider the different capabilities of GPUs.

The system uses eq. (3-1) shown below to quantify the capability of each worker. In eq. (3-1), $P_{comp.}$ is the computation power of a worker which can be measured by the GPU core number or GPU speed. $P_{comm.}$ presents the network bandwidth of the worker and P_{memory} is the size of worker memory. α, β and δ are factors which are application dependent. We can tune their values to fit different types of applications. For computing intensive applications such as Gaussian Elimination, we should enlarge the value of α. If the application is a text search which results in large communication cost, we should enlarge the value of β. Similarly, for memory intensive applications, δ will be given a larger weight. The capability of a worker $C_{workerID}$ is defined as in eq. (3-1).

$$C_{workerID} = \alpha P_{comp.} + \beta P_{comm.} + \delta P_{memory} \tag{3-1}$$

Using the information about all hardware resources in the system, HG decides the proportion of workload to be assigned to each worker by using eq. (3-2) .

$$WorkSet_{ID} = C_{workerID} / \sum_{i=1}^{n} C_{workerID} \tag{3-2}$$

For example, assume HG has 4 workers and each worker's capability is 40, 30, 20, and 10, respectively. So we can get $P_1 = 40\%$, $P_2 = 30\%$, $P_3 = 20\%$, and $P_4 = 10\%$.

The problem space is considered to be a 2 or 3 dimensional grid as mentioned before. In this thesis, we assume a 5-point stencil communication pattern in a 2-dimensional uniform grid where the grid size (grid points) represents workload. The partitioning algorithms take care of dividing the grid and assigning grid points to each processor. Based on the number of grid points assigned to each processor, these algorithms are categorized as even and uneven partitioning. Even partitioning is applied for homogeneous processors and uneven partitioning for heterogeneous processors. Based on the locations of grid points, they are categorized as scatter, contiguous point, contiguous row (column), interleaved row (column), rectangle, and arbitrary polygon partitioning. In the rest of the paper, p denotes the number of processors and n is the side length of a square grid.

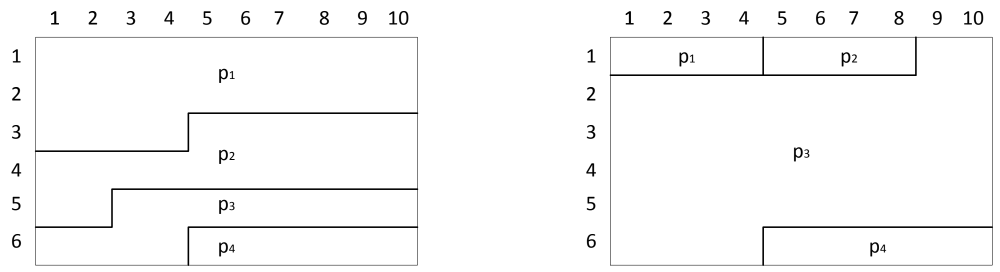
Scatter, contiguous point, and arbitrary polygon partitioning are able to achieve fine-grained load balance by assigning proportional grid points to each processor according to their relative speeds. These kinds of methods work very well in embarrassingly parallel computing, because there is no communication between processors and load balancing is the first concern. However if the communication pattern is a 5-point stencil then a grid point needs to communicate with its nearest four surrounding neighbors. In the worst case, scatter and interleaved row (column) need $O(p^2)$ communications during each iteration when every processor is adjacent to every other processor. The worst case number of grid points that must be transferred is $O(n^2)$ when no neighboring grid point resides on the same processor.

The communication cost of arbitrary polygon partitioning depends on the sides of the polygon and the communication message size depends on the length of its sides. Arbitrary polygon partitioning is able to minimize communication cost but it has to use a $O(n^2)$ lookup table to restore processors location of each grid point. So for heterogeneous processors and dependent data applications, the realistic partitioning algorithms are contiguous point, contiguous row (column), and rectangle.

3.1. Contiguous Points

The grid point is the basic partitioning unit of contiguous points. So this method achieves fine-grained load balancing. Each processor is assigned the number of grid points in proportion to its relative speed. For example, in Figure 3-1 (a), the grid is 10 by 6. Processor p_1 gets the first 24 points which consist of the first two rows and the beginning 4 points in the third row in Figure 3-1 (a) if the grid points are in row-major order. p_2 gets 18 points; p_3 gets 12 points; p_4 gets the last 6 points.

The communication cost is varied. In Figure 3-1 (a), p_2 and p_3 generate the maximum communication number 4 because both of them are adjacent to two other processors and for each processor it needs two communications, one for sending and one for receiving. The total number of communications is 12. A message is transmitted for each communication. The message size is defined as the number of grid points on the boundary between two processors. The maximum message size is 10 data items which is on the boundaries between the pair p_1 and p_2, and the pair p_2 and p_3. The message size of the pair p_4 is 6 data items. So the total number of grid point values transmitted during each iteration is 52 (2×(10+6)) data items. If the relative speeds of processors are changed the communication cost may change as well. For example in Figure 3-1 (b), the total number of communications is 16 and the total number of grid point values transmitted is 30. The total number of grid point values transmitted is decreased but the number of communications is increased.



(a)The relative speeds of processors are (4, 3, 2,1) (b) The relative speeds of processors are (1, 1, 11, 2)

Figure 3-1. Contiguous points partitioning for 4 processors

In the worst case, the maximum number of communications for a single processor as p_3 in Figure 3-1 (b) reaches up to $2(p-1)$ where p is the number of processors when the relative speeds of other processors are small enough to be assigned less than one row or column of grid points. Usually the grid size is very large and the difference of relative speeds of processors is within one or two orders of magnitude. So it's seldom that two or more processors are assigned less than a row or column grid points. In short, in most cases the communication cost of contiguous points is close to the cost of contiguous row.

3.2. Contiguous Row (Column)

This method is similar to contiguous points. The main difference is it uses the row or column in the grid as the basic partitioning unit. So its load balancing is slightly worse than contiguous points. However, the communication cost of this method is very stable and easily estimated. The total number of communications is $2p-2$ and the total number of grid point values transmitted is $2n(p-1)$.

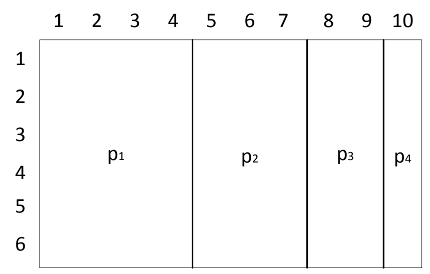
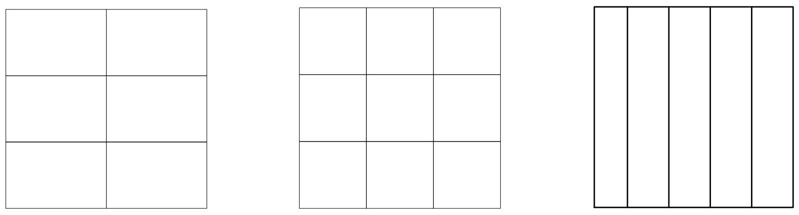


Figure 3-2. Contiguous column partitioning for 4 processors with relative speeds(4, 3, 2, 1)

If the grid is not a square, we can divide along the larger dimension to reduce the communication message size. For example, in Figure 3-2 the grid is 10×6 and we divide along x dimension according to the relative speeds of processors. The processors are organized as a chain. The most neighbors of a processor are 2. Processor, p_2 in Figure 3-2 sends to and receives from p_1 and p_3. The advantage of this method is it generates the least number of communications in the worst case of all methods examined here. However the number of grid point values transmitted $2n(p-1)$ is usually large.

3.3. Rectangle partitioning

To avoid using an $O(n^2)$ lookup table storing the processors' locations of all the grid points, we partition the grid to different size of rectangles whose sizes are proportional to the relative speeds of processors. For locating the grid points of a processor, we store the points of top-left and bottom-right corner of all rectangles. If we want to find a grid point is on which processor, we just need to compare the coordinate of a grid point to the coordinates of top-left and bottom -right points of all rectangles. If the grid point lies within the top-left and bottom-right points area the grid point should be in the processor assigned the rectangle area. The size of the rectangle is the workload and its perimeter denotes communication size measured by the number of grid points transmitted. For a particular area of a rectangle, the shortest perimeter is when the rectangle is a square. Hence, the idea of this method is to make the rectangle as square as possible to reduce communication size. The basic partitioning method for homogeneous processors is finding the two largest factors whose product equals the number of processors. Then divide the longer edge of the grid by the larger of these factors and divide the shorter edge of the grid by the smaller of these factors. Figure 3-3 demonstrates the different partitionings of a square grid with different numbers of homogeneous processors. In Figure 3-3(a), there are 6 processors. The two largest factors are 3 and 2 as shown in the figure. The best case is when p is a square number and we are able to divide the grid to p even squares such as Figure 3-3(b). The number of communication is $4p$ but the message size is reduced to $n/\sqrt{p}$. The worst case is that p is a prime number, we have to use contiguous column method as in Figure 3-3(c).



(a)The number of processors is 6   (b) The number of processors is 9   (c) The number of processors is 5

Figure 3-3. A basic rectangle partitioning for homogeneous processors

(P E Crandall and M J Quinn 1994) modified the basic partitioning algorithm for homogeneous processors, called single group partition (SGP), to reduce the message size when the number of processors is a prime. This method firstly decides whether the number of processors is a prime or not. If it is not a prime divide the grid using the basic partitioning method. Otherwise, create two subgroups: subgroup 1 has just a single processor and subgroup 2 has the remainder. Then divide the current grid proportionally according to the total speeds of each subgroup along the current large dimension. Use single group partitioning to process subgroup 2. For example, in Figure 3-4 there are 5 processors. Firstly, divide the processors to 2 groups: group 1 has 1 processor and group 2 has 4. Because the ratio of relative speeds of the two groups is 1:4 we divide the grid as in Figure 3-4(a) and assign the smaller part to p_1. Then we partition the rest of the grid for the group 2 which has 4 processors. The largest multiplier factors of 4 are 2 and 2. So we can get the final result as Figure 3-4 (b). The length of cut lines in Figure 3-4 (b) is $\frac{14}{5}n$ which is less than $4n$ in Figure 3-3 (c).

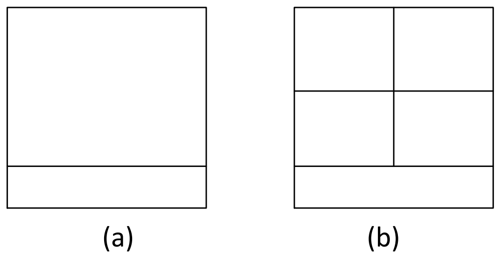


Figure 3-4. Single group partitioning algorithm for homogeneous processors for 5 processors

To take advantage of homogenous partitioning, Phullis E. Crandall (P E Crandall and M J Quinn 1994) proposed the partial homogeneous partitioning algorithm (PHP) for heterogeneous processors. This method groups the processors by relative speed and treats each group as a unit. They then use a heterogeneous partitioning algorithm to divide grid to sub-grids according to the relative speeds of each unit. The heterogeneous partitioning algorithm applied in (P E Crandall and M J Quinn 1994) is a kind of bisection method. It recursively divides the groups of processors into two sets and makes sure each set has total relative speeds as close as possible until only a single group is left. In each sub-grid, use single group partitioning method to process each unit.

An example of this method is given in Figure 3-5. 29 processors are divided into 5 groups. 3 groups have 5 processors each, 1 group has 10 processors and the remaining group has 4 processors The relative speeds of the processors in the 3 groups are 1, 3, and 4. i.e., the processor in one group has relative speed 1, 3 in another group and 4 in the 3rd group. The group with 10 processors has relative speed 2. The last group with 4 processors has relative speeds 5. So there are 5 groups with total relative group speeds (20, 20, 20, 15, 5). Firstly we divide the five groups into two sets (20, 20) and (20, 15, 5) so that the two groups are balanced. The sub-grids for the two sets are shown in Figure 3-5 (a). We divide the left sub-grid into two parts as shown in Figure

3-5(b). For the group with the total relative speed of 20 consisting of 10 processors at relative speed 2, we use SGP to partition these 10 processors in the grid at the top-left corner of the grid as in Figure 3-5(c). Using this process, we get the final result shown in Figure 3-5(k).

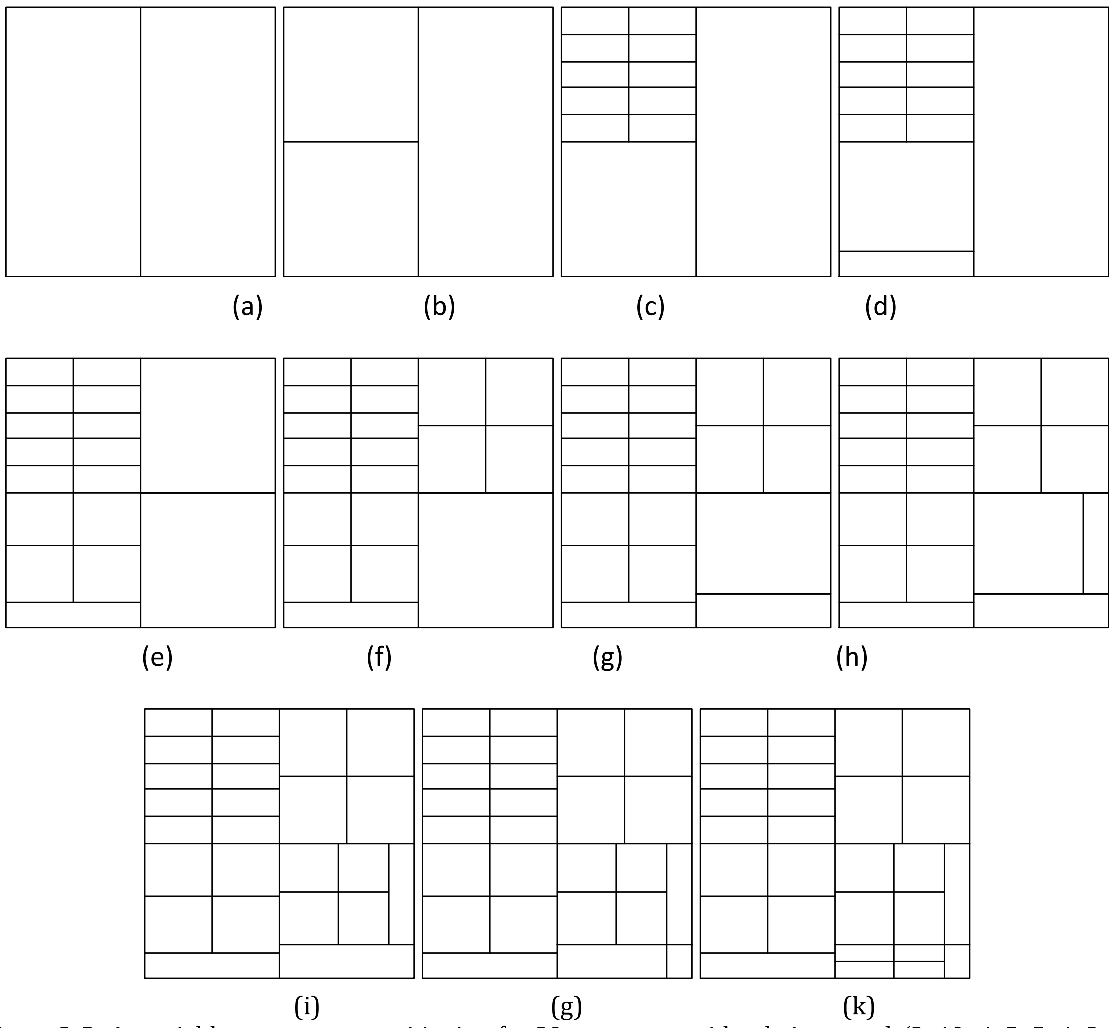


Figure 3-5. A partial homogeneous partitioning for 29 processors with relative speeds(2x10, 4x5, 5x4, 3x5, 1x5) where 2x10 means 10 processors with relative speed 2

3.4. A novel partial homogeneous partitioning based on grid shape

A novel partial homogenous partitioning algorithm based on grid shape (NPHP) is a variant of partial homogeneous partitioning (PHP). The algorithm is given below. NPHP consists of two methods, advanced single group partition (ASGP) and multi-group partition (MGP). The NPHP calls partitioning method ASGP if it deals with only 1 group otherwise it calls MGP. Before

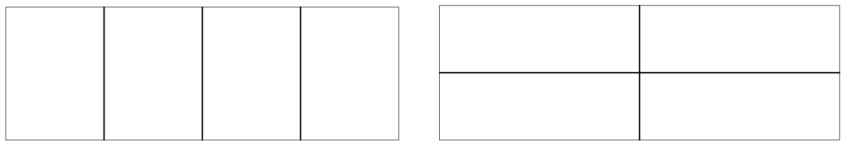
NPHP some preprocessing is necessary. Group the processors by relative speed and sort the group speeds in descending order. If some groups have the same group speeds they are sorted by processor number in descending order. The main advantage of our approach is that it is able to generate smaller message sizes by avoiding cutting along the larger dimension and using a square dimension.

Algorithm 3-1 Partial homogenous partitioning algorithm based on grid shape (NPHP)

```
input:
        set of groups, *groups*;
        information of current dimension, *dimension*;
        working grid, *grid*;
1:   If there is only 1 group,
2:       Advanced Single Group Partition ( groups, dimension, grid);
3:   Else
4:       Multi-group Partition (groups, dimension, grid);
```

PHP takes advantage of the homogenous partitioning algorithm. However the homogenous partitioning method applied in PHP is not smart enough. When the grid is not a square, contiguous row may be better than single group partitioning (SGP). This is because SGP, which finds the two largest multiply factors of p and then divides the grid according to these two factors, does not make the sub-grids as square as possible.

SGP has not considered the shape of the working grid. When the grid is not square and it is narrow it would be better to cut along the smaller dimension instead of following SGP to cut along the larger dimension. For example, in Figure 3-6, the grid is a $3n \times n$ rectangle. SGP uses a cross to cut the grid to four parts. The number of data items transmitted is *8n*. If the contiguous column is applied they are reduced to *6n*. In PHP, the sub-grids for each group are usually not square and can be very narrow.


(a)Continuous column (b) Single group partitioning
Figure 3-6. An example where single group partitioning is worse than continuous column

When it comes to a square grid, SGP fails to make the sub-grids as square as possible. The advanced single group partition algorithm (ASGP) takes advantage of a square number to divide *p* to the largest square number *s* less than *p* and the remainder. Since *s* is a square number we can get the minimum number of data items transmitted. For example, in Figure 3-7 the two methods (SGP and ASGP) are running on a square grid for partitioning 10 processors. The largest multiply factors of 10 is 2 and 5. So SGP divides the grid to 2 by 5 and the number of data items transmitted is 10*n* where *n* is the length of the grid. ASGP shown in algorithm 2 finds the largest square number less than 10 which is 9. It divides the grid as 1:9 and assigns the bottom narrow part to a processor. Then it is divided into 9 sub-grids as in Figure 3-7 (b). The number of data items transmitted in ASGP in Figure 3-7(b) is *9.6n* which is less than SGP.

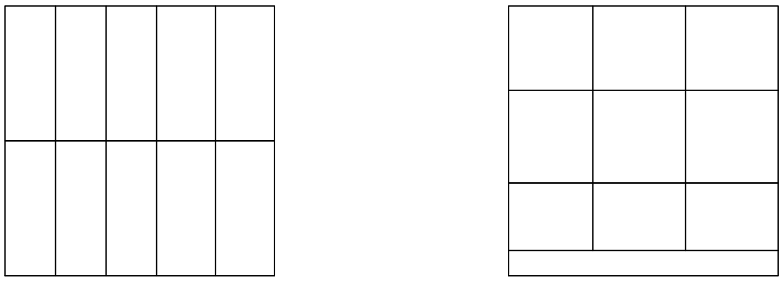

(a)single group partitioning (SGP) (b) Advanced single group partitioning (ASGP)

Figure 3-7. Partitioning on a square grid for 10 homogeneous processors

We assume the largest square number less than *p* is *s* and the largest two multiply factors of *p* is α and β. The grid is *x* by *y* and we assume *y* is not less than *x* and α is not less than β. If we use α and β to divide the grid the length of the cutting lines in the grid is

$$x(\alpha-1) + y(\beta-1). \tag{3-3}$$

If we use square number *s* the length of the cutting lines is

$$x(\sqrt{s}-1) + y\times \frac{s}{p}(\sqrt{s}-1) + O(p-s). \tag{3-4}$$

where *O(p-s)* denotes the length of cutting lines in the narrow grid for the remainder. For the remainder part we can call ASGP to partition. However for analysis we just consider the worst case in using the contiguous partitioning in this sub-grid. So for the worst case the length of cutting lines does not become larger than $x+ y\times \frac{(p-s-1)(p-s)}{P}$. We can therefore conclude (3-5).

$$x(\sqrt{s}\text{-}1) + y\times \frac{S}{P}(\sqrt{s}\text{-}1) + O(p\text{-}s) < x(\sqrt{s}\text{-}1) + y\times \frac{s}{p}(\sqrt{s}\text{-}1) + x+ y\times \frac{(p-s-1)(p-s)}{P} \qquad (3\text{-}5)$$

From (3-3) – (3-5):

$$x(\alpha\text{-}\sqrt{s}\ \text{-}1) + y(\beta\text{-} \frac{S}{P}\sqrt{s}\text{-} \frac{P-S-(P-S-1)(P-S)}{P}) \qquad (3\text{-}6)$$

Since (3-3) – (3-4) > (3-3) – (3-5), if we want (3-3) – (3-4) > 0 which means partitioning by square number is better than by two multiple factors, we need (3-6) to be greater than 0.

$$x(\alpha\text{-}\sqrt{s} - 1) + y(\beta\text{-} \frac{S}{P}\sqrt{s}\text{-} \frac{-(P+1)^2-(S+1)^2+2PS}{P}) > 0$$

$$\frac{x}{y} > \frac{-P\beta+S\sqrt{s}-(P-1)^2-(S+1)^2+2PS}{P(\alpha-\sqrt{s}-1)} \text{ where } x \leq y \text{ and } \alpha \geq \beta \qquad (3\text{-}7)$$

For example, if we have 10 processors, that is p is 10 and s will be 9. Then (3-7) turns to be $\frac{x}{y} > \frac{4}{5}$ and $x \leq y$.

So if the ratio of smaller and larger dimension is larger than 4/5 for 10 processors, the grid should be partitioned by the square number otherwise by the largest two multiply numbers. We use μ to denote the value of $\frac{-P\beta+S\sqrt{s}-(P-1)^2-(S+1)^2+2PS}{P(\alpha-\sqrt{s}-1)}$.

Algorithm 3-2 Advanced single group partitioning (ASGP)

```
Input:
        set of processors, processors;
        information of current dimension, dimension;
        working grid, grid;
1:   set the larger dimension as current dimension;
2:   If the cardinality of the processors is 1,
3:       return;
4:   Else if cardinality of the processors is a square number,
5:       If the ratio of the two current dimension (larger/smaller) is larger than the squared root of cardinality
            of the processors,
6:           Divide the processors into 2 subgroups:
                 Assign one processor to subgroup 1;
                 Assign remaining processors to subgroup 2;
7:           Divide the grid proportionally according to the total speeds of each set along the current dimension;
8:           Set the current grid to the section assigned to subgroup 1;
9:           ASGP(subgroup 1, current dimension, current grid);
10:          Set the current grid to the section assigned to subgroup 2;
11:          ASGP (subgroup 2, current dimension, current grid);
12:      Else
13:          Divide the two edges of grid by the square root of cardinality of the processors;
14:  Else the ratio of smaller and larger dimension of this grid is not larger than μ,
15:      Find the two largest factors whose product equals the cardinality of the processors;
16:      Divide the longer edge of the gird by the larger factor;
17:      Divide the shorter edge of the grid by the smaller factor;
18:  Else
19:      Find the largest square number less than the cardinality of the processors;
20:      Divide the processors into 2 subgroups:
             Assign the square number of processors to subgroup 1;
             Assign the rest of processors to subgroup 2;
21:      Divide the grid proportionally according to the total speeds of each set along the current dimension;
22:      Set the current grid to the section assigned to subgroup 1;
23:      ASGP (subgroup 1, current dimension, current grid);
24:      Set the current grid to the section assigned to subgroup 2;
25:      ASGP (subgroup 2, current dimension, current grid);
```

MGP introduced in (P E Crandall and M J Quinn 1994) is shown as algorithm 3-3. Its purpose is to divide groups into two sets where the relative group speeds between the two sets are as close as possible. It avoids making the two sets very narrow and makes each group use squared number partitioning as much as possible.

Algorithm 3-3 multi-group partitioning (MGP)

```
Input:
        set of groups, groups;
        information of current dimension, dimension;
        working grid, grid;
1:   set the larger dimension as current dimension;
2:   Divide the processors into 2 almost even sets:
3:   Divide the grid proportionally according to the total speeds of each set along the current dimension.
4:   Set the current grid to the section assigned to set 1;
5:   NPHP(set 1, current dimension, current grid);
6:   Set the current grid to the section assigned to set 2;
7:   NPHP(set 2, current dimension, current grid);
```

An example of NPHP – see algorithm 3-1 is given in Figure 3-8. The combination of processors is the same as for Figure 3-5: 4 processors with relative speed 5, 5 processors with relative speed 4, 5 processors with relative speed 3, 10 processors with relative speed 2, and 5 processors with relative speed 1. The sorted groups are (20, 20, 20, 15, 5) where the first 20 consists of 10 processors with speed 2, the second 20 consists of 5 processors with speed 4, and the last 20 consists of 4 processors with speed 5. MGP divides them into two sets (20, 20) and (20, 15, 5). Figure 3-8(a) shows the division of the grid for these two sets. Utilize MGP to divide the left sub-grid into two parts shown as Figure 3-8(b). The top-left corner of the grid will be assigned to the group which has 10 processors at relative speed 2. For this group ASGP is involved. Since the number of processors is 10 the largest squared number less than 10 is 9. The μ of 10 processors is 4/5. In this sub-grid the ratio of larger and smaller dimension is 1 which is larger than 4/5. Hence we can divide this sub-grid by the square number method. ASGP divides the 10 processors to 1 and 9. So ASGP assigns one tenth to one processor and divides the sub-grid as in Figure 3-8(c). A call to ASGP processes the rest of the sub-grid. The remaining number of processors is 9 which is a squared number. First check the ratio of the larger and smaller dimensions. The ratio is around 1 which is smaller than the squared root of the remaining processors' number 3. So we divide the two edges of the grid by 3 shown as Figure 3-8(d). The second group consisting of 5 processors at speed 4 is divided into two sets of 1 processor and 4 processors since its ratio 1 is larger than its μ=2/3. Assign one fifth of this sub-grid to one processor and divide the rest of the grid by square number method as in Figure 3-8(e) and (f). In the same way, we can partition the right sub-grids.

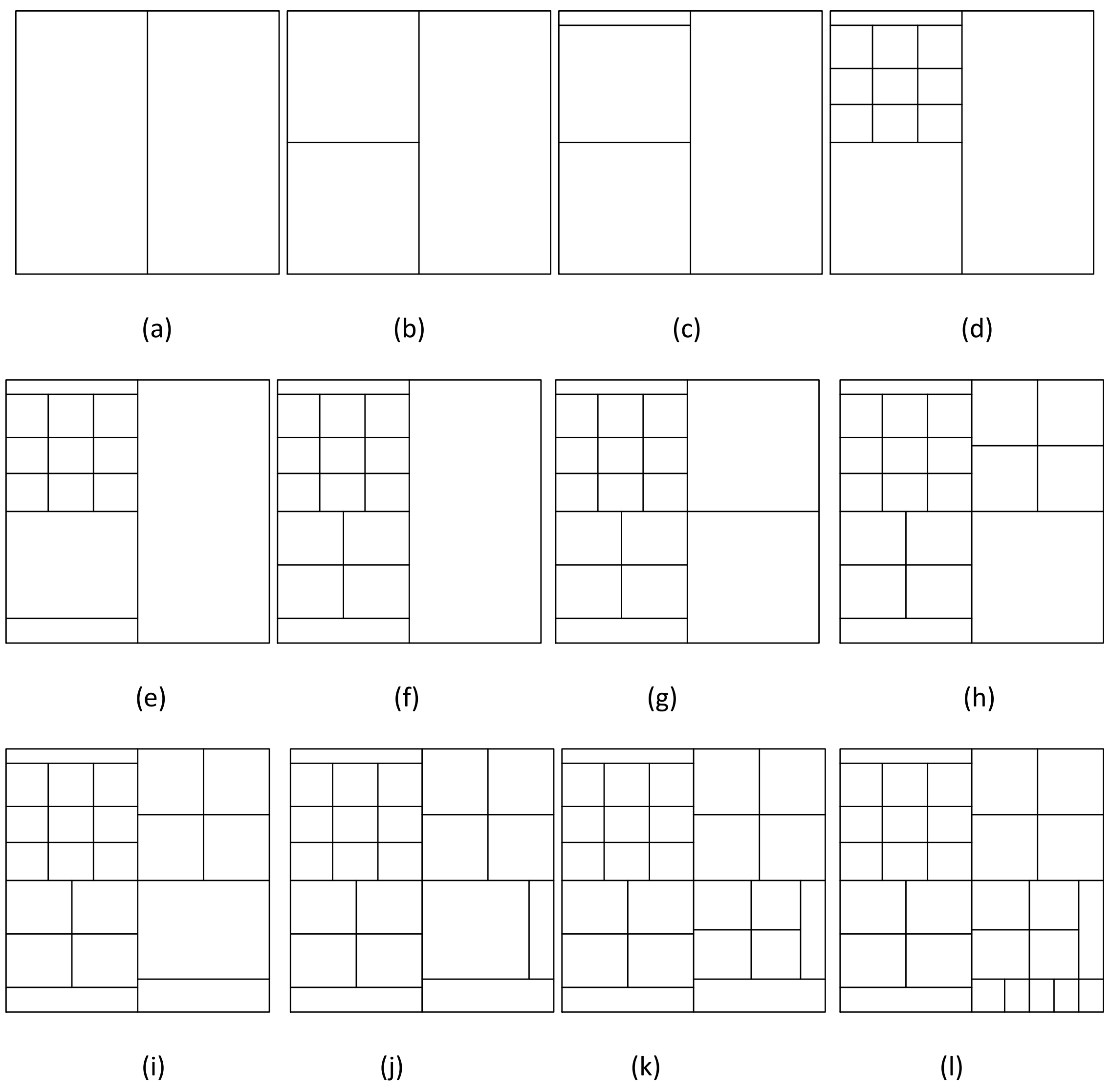


Figure 3-8. A novel partial homogeneous partitioning (NPHP) for 29 processors with relative speeds (2x10, 4x5, 5x4, 3x5, 1x5) where 2x10 means 10 processor with relative speed 2

3.5. Performance of the algorithm

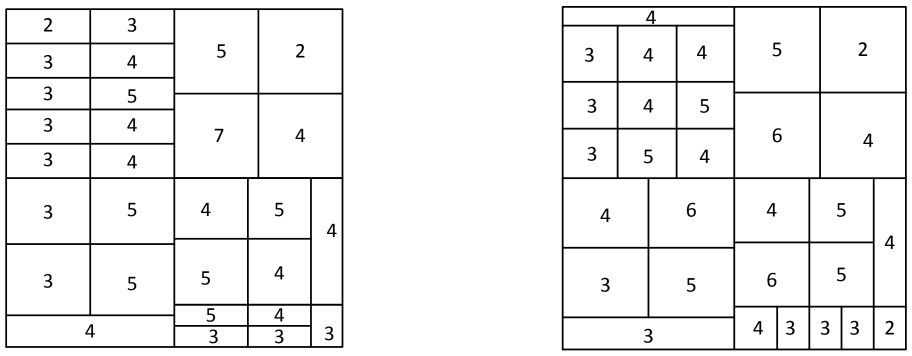
We make a comparison between the partitioning results of NPHP and PHP including the number of communications which is *O(p)* and data items transmitted of all processors which is *O(n).* In realistic applications the number of processors, *p* is not big and less than thousands. However the side of the grid, *n* is usually very large at more than millions. So the data items transmitted by all processors plays a major role in impacting the communication time. In algorithm 2, NPHP guarantees the least number of data items transmitted of all processors.

Figure 3-9 shows the partitioning results of both methods for 5 groups of processors as mentioned in Figure 3-5 and 3-8. Each sub-block is assigned to a processor and the number in the sub-block indicates the number of communications required by that sub-block during each iteration. The maximum number of communication of PHP is 7 and it is 6 in NPHP. The most important factor is the data items transmitted of all processors for PHP which is 18.4n compared to 16.9n for NPHP. In this example NPHP is able to reduce the data transmitted by more than 8%.

For transmitting one message, the time includes the preparation for this connection and the transfer time which depends on the message size. Hence the time of communication for one iteration is estimated as the maximum number of communications and the total size of data transmitted as in (3-8).

$$\sigma\, max\, C_i + \omega \sum_{i=1}^{p} M_i \tag{3-8}$$

Where σ is the message preparation latency, ω is the transmission speed measured by seconds per byte, p is the number of processors, $max\, C_i$ means the maximum number of communications required by any single processor, and M_i denotes the number of bytes transmitted by processor i.



(a) PHP (b) NPHP

Figure 3-9. A comparison of NPHP with PHP. The number within each sub-grid presents the number of communications required by that sub-block during each iteration.

In the following analysis, we assume that σ is 1 msec per message, and bandwidth is 10 Mbytes per second and each date item has 4 bytes. Figure 3-10 presents the size of data that needs to be transmitted for the n x n grid problem shown in Figure 3-9. In Figure 3-9, there are 29 processors so the number of data transmitted during each iteration of contiguous row method is 56*n*. NPHP is able to reduce the amount of data transmitted by 8% compared with PHP and 70% compared with contiguous row (Figure 3-10).

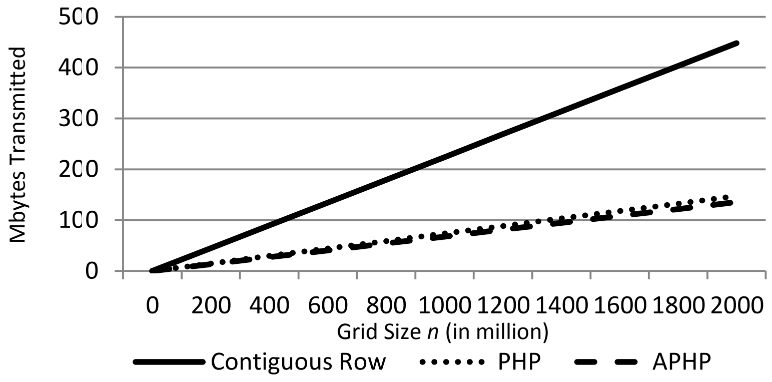


Figure 3-10. The size of data transmitted for n x n grid problem in Figure 3-9 partitioned by contiguous row, PHP, and NPHP.

For the grid problem in Figure 3-9, the maximum communications of contiguous row, PHP, and NPHP are 2, 7, and 6, respectively which means contiguous row requires the least time for preparation of connections. A shown is Figure 3-11, when the grid size n is less than 200 million, the contiguous row method performances best of all of the three. However with the increase of grid size *n*, NPHP has an essential advantage over the other two methods. NPHP is able to reduce the communication time 10% compared with PHP and 58% compared with contiguous row.

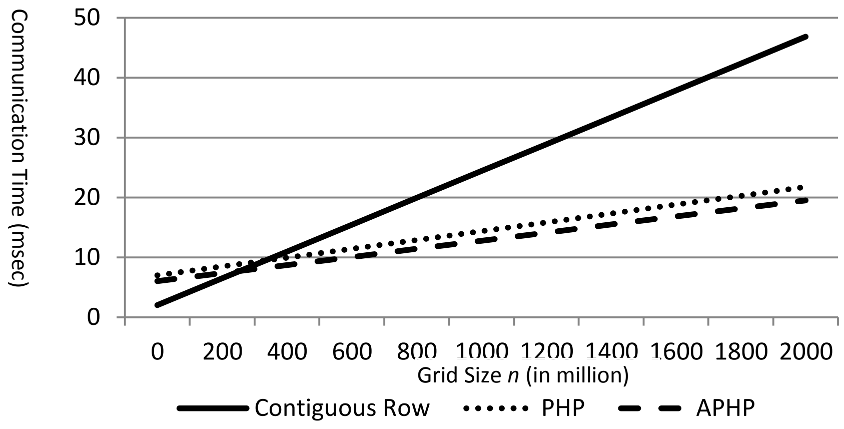


Figure 3-11. The communication time for n x n grid problem in Figure 3-9 partitioned by contiguous row, PHP, and NPHP.

In this chapter we have looked at the problem of minimizing inter-processors communications. We have proposed a novel partitioning algorithm for grid problems. This algorithm depends on the grid shape to divide homogeneous processors by square number or the two largest multiply numbers. Our algorithm improves the partial homogeneous parallel partitioning algorithm and is able to reduce by 8% the size of data transmitted and by 10% the communication time compared to partial homogeneous parallel partitioning.

3.6. Conclusion

We have proposed a novel partitioning algorithm called NPHP for grid problems. This algorithm depends on the grid shape to divide homogeneous processors by a square number or the two largest factors. Our algorithm (NPHP) improves the partial homogeneous parallel partitioning algorithm and is able to reduce by 8% the size of data transmitted and by 10% the communication time compared to partial homogeneous parallel partitioning. In future work we plan to apply our algorithm to 3-D applications.

CHAPTER IV

PARTITIONING ALGORITHMS OF FUNCTIONAL PERFORMANCE MODEL

In the functional performance model the speed of the processor is a function against some arguments, usually the problem size. For CPUs, when the problem size increases and cannot fit into main memory, the data items will be placed at lower levels of the memory hierarchy thus resulting in a decrease of speed of execution of the application. However for GPUs, with an increase of the problem size, more activated threads could hide the high latency in accessing global memory thus resulting in increased execution speed of the application.

Lastovetsky and Reddy (A Lastovetsky and R Reddy 2004) (Alexey Lastovetsky and Ravi Reddy 2007) proposed using the functional performance model (FPM) to partition independent chunks of computations over heterogeneous processors. FPM needs to run the benchmark or real applications to get the absolute speeds of processors for the full range of problem sizes which incurs a very high cost. A relatively efficient sub-optimal solution (Alexey Lastovetsky, Ravi Reddy, and Higgins 2006) was proposed to deal with building the FPM of a processor.

The functional performance model (FPM) assumes the functions of the absolute speeds of all processors against the workload sizes are already known. For example, the functions of absolute speeds of processors 1 to 3 are ready known as shown in Figure 4-1. If x_1, x_2, and x_3 are the workload sizes assigned to processor 1 to 3, their execution time can be expressed as $\frac{x_1}{S_1(x_1)}$, $\frac{x_2}{S_2(x_2)}$, and $\frac{x_3}{S_3(x_3)}$, respectively where $S_i(x_i)$ is the absolute speed of processor *i* with workload size x_i. For the loads to be balanced, their execution times should be the same, that is $\frac{x_1}{S_1(x_1)} = \frac{x_2}{S_2(x_2)} = \frac{x_3}{S_3(x_3)}$. The partitioning problem changes to finding the optimally sloped line crossing the point of origin and these functions and the sum of the values of x-axis of these intersections should be equal to *n*, the total workload size. Because the slopes of the intersections are the same, the execution time of each processor is same.

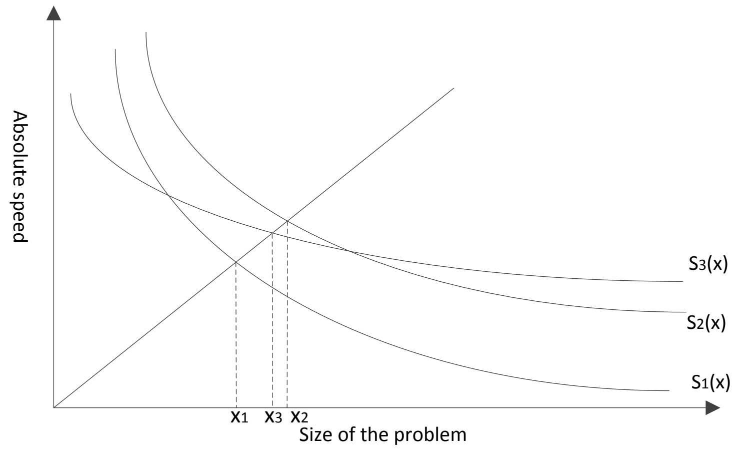


Figure 4-1. Load balancing in function performance model.

4.1. Bisection Method in Function Performance Model

Lastovetsky and Reddy (A Lastovetsky and R Reddy 2004) (Alexey Lastovetsky and Ravi Reddy 2007) proposed a bisection method in FPM to find the optimally sloped line as shown in Figure 4-2. This method firstly finds the absolute speeds of all processors when the total workload is divided evenly which is *n/p* in Figure 4-2 where *n* is the total workload size and *p* is the number of processors. Then we can get the upper bound of the optimally sloped line, line 1 and the lower bound line 2. We divide the angle of line 1 and line 2 evenly and get the line 3. If the sum of x-axis values of the intersections of line 3 and the functions is less than the total workload size then we set line 3 as new upper bound. We divide the angle of line 3 and line 2 evenly and get the line 4. If the sum of the workload sizes of intersections of line 4 and the functions is larger than the total workload size, we set line 4 as new lower bound. We continue to run the bisection steps until we find the optimally sloped line which satisfies $x_1 + x_2 + x_3 = n$. This method does not consider the communication cost. So we cannot use it directly to task dependent applications.

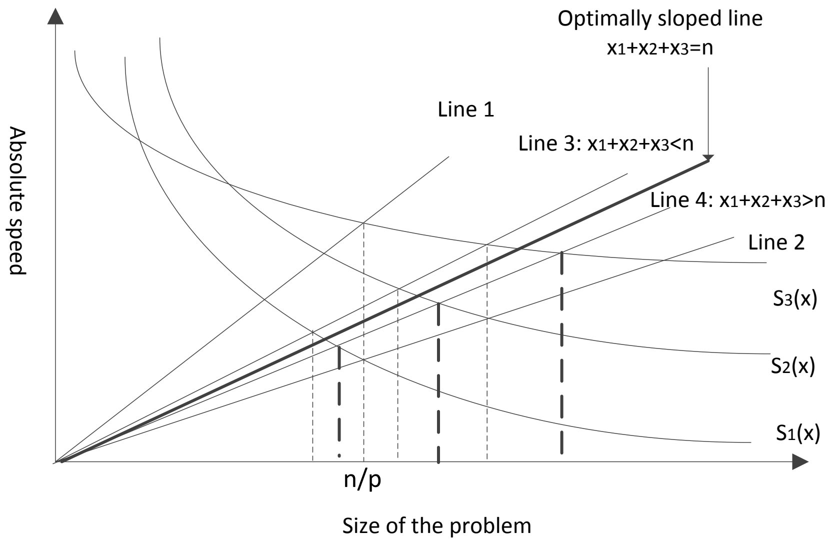
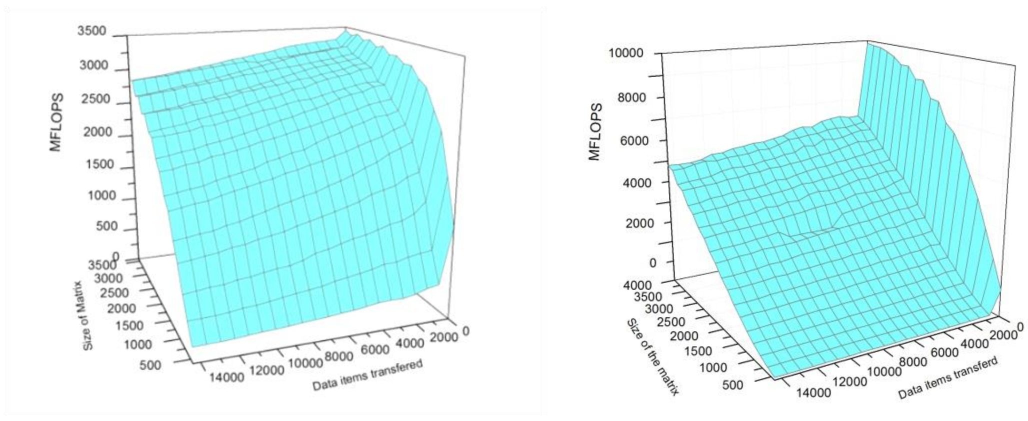


Figure 4-2. A bisection method in Function performance model for partitioning

4.2. Estimating the performance of GPU on dependent tasks

It is well known that the bottleneck of GPU performance is the high latency of accessing global memory. There are two ways to increase GPU performance. One is using lower and faster memory levels of the memory hierarchy such as loading data from global memory to shared memory and the thread to read data from shared memory instead of global memory which is data reuse as discussed in the section 5.6. The other way to increase GPU performance is to use large data volumes. Since a larger data volume involves more active threads, this can hide the high latency of global memory access. The performance function of GPUs is increasing against the workload size as shown in figures 4-3 (a) and (b).

In Figure 4-3, we run the 2-D heat transfer application on two types of GPUs, GeForce GTX650 and Tesla C2050. The test-bed consists of two nodes: one is for computation and the other is for communication. For each GPU, we measure the execution time with different workload sizes. For each workload size, the computation node will send and receive different number of data items to and from the other node. So we can get a performance surface as shown in Figure 4-3 which is increasing with the workload size and decreasing with the communication cost.



(a) GTX650 (b)C2050

Figure 4-3. Absolute speeds of the GPUs against the size of the problem and the communication cost in heat transfer application. The unit of data items transferred is a float.

4.3. Function Performance Model with Communication

In the model we propose the absolute speed of a processor does not only depend on its workload size but also on the communication cost. The absolute-speed of processor i is $S_i(w_i, c_i)$ where S_i is the absolute speed function of processor i, w_i is the workload size of processor i, and c_i is the communication cost of processor i. Depending on the different partition algorithms, the communication cost c_i varies. For example, if the contiguous row partitioning algorithm is applied, c_i is fixed and equals to $2n$. If the homogeneous partitioning algorithm (P E Crandall and M J Quinn 1994) is utilized, c_i depends on the relative speeds of all processors, $S_1, S_2, \dots, S_n$ that is $c_i = f(S_1, S_2, \dots, S_n)$, but it is very hard to find a function to calculate c_i. We make the following assumptions in this dissertation:

1. If $w_i < w_i'$, then $c_i < c_i'$;

If the workload of processor i increases from $\mathrm{w_i}$ to $\mathrm{w_i'}$ and the workloads of other processors do not change, its proportion of the total workload will increase. So the size of block assigned to processor i increases and the perimeter of the block will increase as well which means the communication cost c_i increase to c_i'.

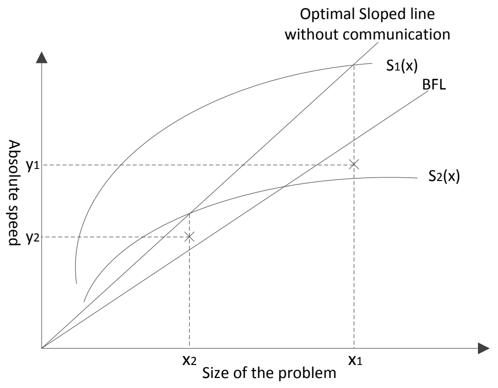
2. If $w_i < w_i', c_i < c_i'$ and $T_i(w_i, c_i) = \frac{w_i}{S_i(w_i,c_i)}$, then $T_i(w_i, c_i) < T_i(w_i', c_i)$, $T_i(w_i, c_i) < T_i(w_i, c_i')$ and $T_i(w_i, c_i) < T_i(w_i', c_i')$.

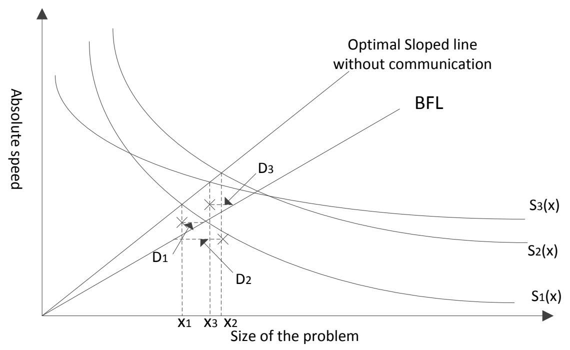
The execution time T_i of processor i, can be represented by $\frac{w_i}{S_i(w_i,c_i)}$. If either of the workload of processor i or its communication cost increases, its execution time will increase as well. So the function of execution time of a processor against the size of problem is monotonically increasing.

3. If $c_i < c_i'$, then $S_i(w_i, c_i) > S_i(w_i, c_i')$

This assumption means the function of absolute speed of a processor against the communication cost is monotonically decreasing. If the number of transferred data items increases from c_i to c_i' of processor *i*, the speed of processor *i* will decrease. However the function of absolute speed of a processor against the size of problem w_i, $S_i(w_i, c_i)$ may increase or decrease depending on the processor architecture. For example, if the problem size fits into main memory of the processor, its speed should increase with increasing the problem size to a limit of the processor as shown in Figure 4-4 (a), because large data volume and a large number of threads can hide the high latency of memory access operations. On the other hand, if the problem size cannot fit into main memory of the processor its speed will degrade because of the paging involved as shown in Figure 4-4 (b). The functional performance model with communication (FPMC) works well in both of situations.

Ignoring communication cost, we can use the bisection method to get the optimal sloped line where $\frac{S_1(x_2)}{x_2} = \frac{S_2(x_3)}{x_3} = \frac{S_3(x_1)}{x_1}$ such as in Figure 4-4 (b). Then we can get the intersections of the optimal sloped line and the speed functions of all processors. The values of x-axis of these intersections represent the workloads assigned to each processor. However the real speed of a processor is expected to be lower since communications is involved. In Figure 4-4, the x points, which are below the speed function S(x), denote the real speeds of processors with communications. As mentioned above, the communication cost is very hard to estimate by a function when advanced partitioning algorithms are applied. Therefore, these x points may be not on a straight line $\frac{S_1(x_1,c_1)}{x_1} \neq \frac{S_2(x_2,c_2)}{x_2} \neq \frac{S_3(x_3,c_3)}{x_3}$. In other words, the workloads of processors will be out of balance when communication cost is involved. We use the least square method to find the best fitting straight line (BFL) denoted as $y = kx$ which is the closest to all x points where $\min_{\overline{x}} \sum (\mathrm{y_i} - \mathrm{kx_i})^2$. The horizontal distance of the x points and the BFL can be expressed as d_i in Figure 4-4 (b), where $d_i = \left\| x_i - \frac{y_i}{k} \right\|$. We use α to denote the relationship between the x points and the line where $\alpha = \sum d_i^2$.





(a) (b)

Figure 4-4. Load imbalance when considering communication. (a) The speed of a processor increases with the increasing of size of the problem. (b) The speed of a processor decreases with the increasing of size of the problem.

At first, we run the benchmark application on each processor assigned different workload sizes to draw the graphs of the processors with speed versus the workload sizes without communication cost. For each workload size, the processor *i* transmits different sizes of data items through the network to get a table of $S_i(w_i, c_i)$ of all the processors with speed versus the workload sizes w_i and communication cost c_i. Then the functional performance model is used in the graph of the processors with speed versus the workload sizes without communication cost to get the optimal partitioning workload sizes of all processors without considering communication. We treat this partitioning as the initial partitioning of FPMC. Then we adjust the problem sizes of processors to get close to the optimal solution satisfying the criterion that the number of elements should be proportional to the speed of the processor with communication cost.

We utilize the move single x point method (MSXP) to move the x points horizontally to be closer to the best fitting line (BFL). While moving the point x, x must always remain in the same position relative to the BFL, that is, it must remain above or under the BFL. If the x point is under BFL and its horizontal distance of BFL is *d*, the x point is moved to the left *d/2*. Then we can estimate its real absolute speed $y' = S_1(x - \frac{d}{2}, c)$. If the speed function against workload size is increasing as in Figure 4-5 (a), we increase *d/2* to the workload of each other processors. Their absolute speeds can be expressed as $S_j\left(x_j + \frac{d}{2}, c_j\right), j \in n$ and $j \neq i$. Otherwise, for Figure 4-5 (b), we do not change the absolute speeds of other processors. Then we use y' and the absolute speeds of other processors to run the partition algorithm (P E Crandall and M J Quinn 1994) and get the new communication cost c_i'. We now get the new x point: the real absolute speed of this motion $S_i(x - \frac{d}{2}, c_i')$. If the new x point is above the line, move to the right *d/4*. Otherwise keep move until *d*, the horizontal distance becomes larger than the last motion or changes to 0. We can use the same way to move x points to the right. In Figure 4-5 (a) and (b), the first motion of the x point is from x to x′ which moves *d/2* to the left and the updated x point is $S_1(x - \frac{d}{2}, c')$ where c' can be calculated by the partitioning algorithm with y_i' and the absolute speed of other processors. Because the new x point $S_1(x', c')$ is above the BFL, we move the new x point to the right *d/4* to x'' which is $x - \frac{d}{4}$.

Proposition 1. If the x point (the real absolute speed of a processor i) is under or above the best fitting line (BFL) $y = kx$ then the x point should be moved horizontal to the left or right respectively to be closer to the BFL.

Proof. The slop of BFL is k. If the x point is under the BFL, then $\frac{S_i(x,c)}{x} < k$. In other words, the reciprocal of $T_i(x,c)$ is smaller than k. So we should decrease $T_i(x,c)$. Because the function $T_i(x,c)$ is monotonically increasing, there are two ways to make $T_i(x,c)$ smaller: decrease the workload x or the communication cost c. However, because the communication cost c is a monotonically increasing with processors speeds, we cannot simply maintain or increase the workload x and change the communication cost c to get a smaller $T_i(x,c)$. Hence the only way to decrease $T_i(x,c)$ is decrease the workload x which means the x point should be moved horizontally left to be closer to the BFL. In the same way, we can show that if the x point is above the BFL, it should move right to be closer to the BFL.

Proposition 2. If the x point is under the BFL, the moved x point which can be measured by $S_i(x',c')$, is smaller than the real absolute speed of processor i at workload size x'. Otherwise, the moved x point as measured by $S_i(x',c')$ is larger than the real absolute speed of processor i at workload size x'.

Proof. If the x point is under the BFL, the horizontal distance between the x point and BFL is d and the workload of the moved x point as measured by $S_i(x',c')$, is x' which is $x - d/2$. The communication cost c' will change since the workload of the processor changes. We cannot get the exact communication cost because we cannot determine the exact workload of other processors at this point. However according to the above assumptions we can get the smallest communication cost if we move the x point to the left. If the speed function against workload size is increasing as in Figure 4-5 (a), we increase *d/2* to the workload of each other processors. Their absolute speeds can be expressed as $S_j\left(x_j + \frac{d}{2}, c_j\right), j \in n \; and \; j \neq i$. Otherwise, as in Figure 4-5 (b), we do not change the absolute speeds of other processors. Then we use $y' = S_i(x - \frac{d}{2}, c)$ and the absolute speeds of other processors to run the partition algorithm. Actually in the real situation, the speeds of other processors will change because their workload sizes change. The proportion of the workload size of processor i and all other processors are smaller than the workload assigned to processor i in the real situation. So according to assumption 1, we can get the communication cost of the moved x point c' to be smaller than the communication cost of the moved x point in the real situation. According to assumption 3, $S_i(x', c')$ must be smaller than the absolute speed of processor i in the real situation. We can use the same way to prove if the x point is above BFL, the absolute speed of moved x point is larger than its speed in the real situation.

Proposition 3. The move single x point method (MSXP) ensures that it does not change the above or under relationship of the x point relative to the BFL.

Proof. According to proposition 2, if the x point is under (above) BFL, the moved x point as measured by $S_i(x', c')$, is smaller (larger) than the real absolute speed of processor i at workload size x'. So the real moved x point should be below (above) the moved x point. MSXP ensures the moved x point is also under (above) the BFL, then the real moved x point is under (above) the BFL.

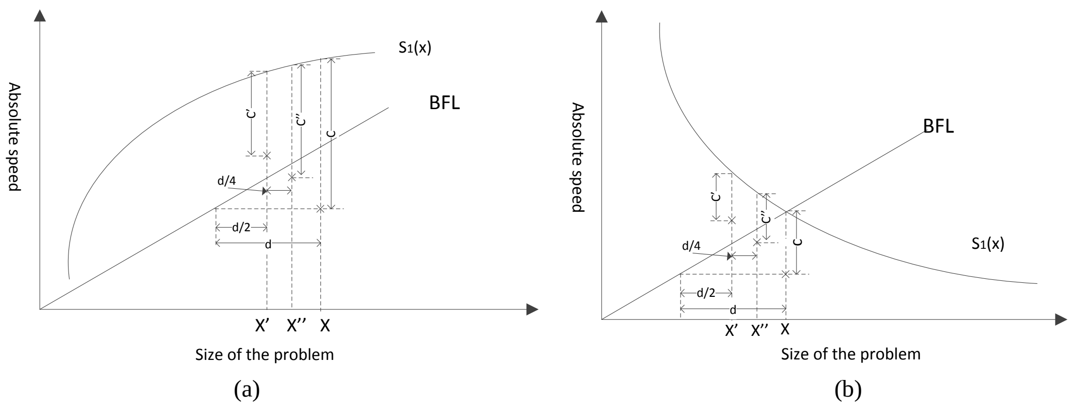


Figure 4-5. The Move Single x Point method. Use bisection method to move the x point horizontally to be closer to the line while ensuring that the above or under relationship of each x point to the line does not change.

The time complexity of MSXP is $\mathrm{O}(\Delta \log \mathrm{d})$ where Δ is the time complexity of the partition algorithm (P E Crandall and M J Quinn 1994) which denotes $\mathrm{O(p)}$ and d is the horizontal distance between a x point and BFL where d is less than n/p. Hence, in the worst case the time complexity of MSXP is $O(p\, log\frac{n}{p})$.

In the Move All x Points method (MAXP), the x points of all processors use the move single x point method to move closer to BFL while ensuring that the total sum of the workload remains unchanged. For example in Figure 4-6, x_1 and x_3 are above BFL which means processor 1 and 3 are under-loaded. So we should move x_1 and x_2 to the right to increase the workloads on processor 1 and 3. On the other hand, x_2 is under the line and we should move x_2 to the left to reduce the workload on processor 2. The rules of the motion are as follows:

1. Use the MSXP method to move the x points above tje BFL to the right. $rm_i = MSXP(x_i)$ where rm_i denotes increasing rm_i data items for processor i. The expected decrease in performance of processor *i* is represented as $dp_i = S_i(x_i, c_i) - S_i(x_i + rm_i, c_i')$. $rm = \sum rm_i$. So *rm* is the total number of data items to be added to the processors above the BFL.
2. Use the MSXP method to move the x points under the BFL to the left. $lm_j = MSXP\left(x_j\right)$ where lm_j denotes the number of data items to be removed for processor *j*. The expected increase in increasing of processor *j* is represented as $ip_j = S_i\left(x_i - lm_i, c_j'\right) - S_i(x_i, c_i)$. $lm = \sum lm_i$. So *lm* is the total number of data items that should be removed from the processors under the line.
3. If *rm* equals *lm*, then finish the motion.
4. If *rm* is larger than *lm*, $dm = rm - lm$, sort dp_i for all processors in increasing order. Get the smallest dp_i of processor *i*. If rm_i is no less than *dm*, then move left dm data

items of processor i. Otherwise, move left rm_i data items of processor i and update rm and dm by reduction of rm_i. Then go to step 3.

5. If rm is smaller than lm, $dm = lm - rm$, sort ip_i for all processors in increasing order. Get the smallest ip_i of processor i. If lm_i is no less than dm, then move right dm data items of processor i. Otherwise, move right rm_i data items of processor i and update lm and dm by reduction of rm_i. Then go to step 3.

In the worst case the time complexity of MSXP is $O(p \, log \frac{n}{p})$. Because there are p processors, there are number of p x points. For all x points, in the worst case the time complexity of MAXP is $O(p^2 \, log \frac{n}{p})$.

FPMC is presented in algorithm 1. The motion of problem sizes of processors is to adjust the workload sizes for processors to make their load balance. When the x point of a processor is above the BFL it should move right to increase the workload otherwise move left. In this process, the total workload size must be constant which means the total left motion of data items should be the same as the total right motion of data items. At the end of each motion, the distance α (see above) between the x points and BFL, is calculated. When the α of this motion is less than the last one, we update the speeds of all processors and use these new speeds shown as $y'_1, y'_2,$ and y'_3 in Figure 4-6 to partition and then run MAXP until the α of this motion is greater than or equal to the last one.

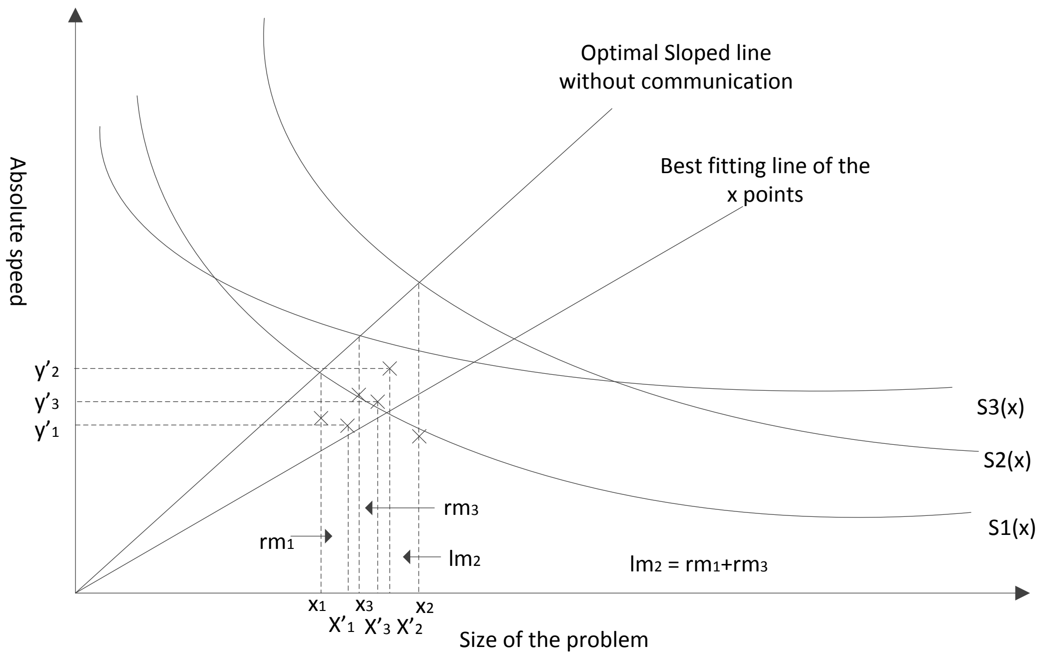


Figure 4-6. The Move All x Points Method. Move x points of all processors using the move single x point method to get closer to BFL while ensuring the total sum of workload remains unchanged.

The FPMC (Functional Performance Model with Communication) is introduced in algorithm 4-1. In FPMC, α, horizontal distances between x point and the BFL controls the loop executing MAXP method. If α increases or equals to 0, FPMC is terminated. Otherwise, MAXP is performed to get a smaller α.

Algorithm 4-1 Functional performance model with communication (FPMC)

```
Input:
    table of absolute speeds versus the workload and communication sizes of all processors, $S_i(w_i, c_i)$ for $i \in n$;
    working grid, $grid$;
Output:
    $W$ set for workload sizes of all processors , $W = \{w_1, w_2, \dots, w_n\}$;
1:  Use the realistic performance model to get the initial S set (the absolute speeds of all processors
    without communication cost.
2:  initialize the close relationship of x points and the line $\alpha' = \sum d_i^2$ to MAX and α = 0;
3:  According to S, use the partitioning algorithm to get the communication cost set C.
4:  for each i in n:
5:      $p_{i(x,y)} = S_i(w_i, c_i)$;
6:  end for;
7:  while $\alpha < \alpha'$
8:      $\alpha' = \alpha$;
9:      Construct the BFL which makes the distances of all $p_{(x,y)}$ minimum;
10:     Use MAXP($S(w, c)$, $p_{(x,y)}$) to update α and $\{S_1, S_2, \dots, S_n\}$;
11: end while;
12: According to $\{S_1, S_2, \dots, S_n\}$ (the absolute speeds of all processors with communication cost), use the
    partitioning algorithm to get $W$.
```

The time complexity of FPMC is determined by the loop number between lines 7 and 11 in algorithm 4-1. The worst case is MAXP just reduces the horizontal distance of an X point and BFL by 1 in each loop. All of horizontal distances of the x points are assumed *n/p* and there are *p* X points. So the total horizontal distance is n and the largest loop number is n as well. Furthermore, the time complexity of MAXP in the worst case is $O(p^2 \, log \frac{n}{p})$ and the loop number is n. In short, in worst case the time complexity of FPMC is $O(np^2 \, log \frac{n}{p})$.

4.4. Conclusion

In this chapter, we discussed how to accurately estimate the performance of a processor while considering communication cost. We have proposed a novel functional performance model partitioning algorithm that takes into consideration the communication cost for heterogeneous GPUs (FPMC). This algorithm depends on the realistic GPU performance with real communication cost to estimate the speed of GPUs. The estimated speeds are used to recursively partition the workload to ensure load balance. FPMC outperformances previous FPM which does not consider communication cost when the application needs inter-processor communication. However, it’s very expensive to construct the benchmark performance table of GPU with a wide range of workload and communication cost and for different kinds of applications. In future work we plan to use an adaptive method to construct the benchmark performance table of GPU which can automatically control interval of workload to reduce the number of experiments.

CHAPTER V

DESSIGN AND IMPLEMENTATION

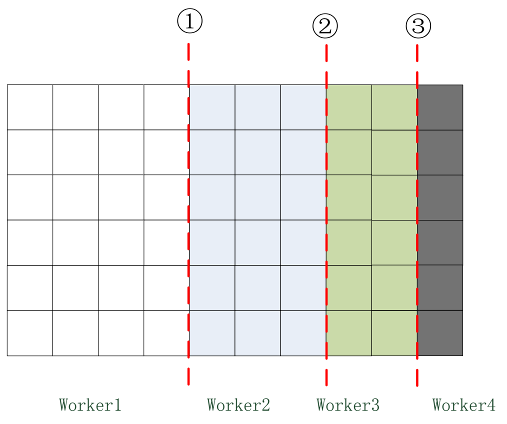
The implementation of HG is based on our previous work(R. Yang and Thomas 2012).The HG system utilizes heterogeneous GPU resources and according to their relative capabilities tasks are assigned. From an implementation view, we use a computing unit to represent a grid point. Hence the tasks mean computing units in this work. The basic computing unit in HG includes 4 fields, namely, index, key, value, and neighbors. HG uses a master/workers architecture. In this work, a worker means a processor or GPU. Each worker can be assigned a large number of computing units. The major differences from our previous work are the addition of dynamic neighbor components. Some optimizations including using partitioning algorithms described in chapter 3 and 4, and data reuse technology described in section 5.6. Using dynamic neighbor components, the user is able to add or delete neighbors of computing units between computing iterations. Hence it allows HG to support not only fixed task dependent applications but also dynamic ones, such as Gaussian Elimination application. These optimizations improve on our previous HG system performance.

In this chapter, we firstly introduce the architecture of HG and its three phases, initialization, communication, and computation. Then we analyze the implementation and performance of the system. Finally, we demonstrate some system parameters and APIs. With these APIs, users are able to implement task dependent applications, and improve the system performance by adjusting system parameters.

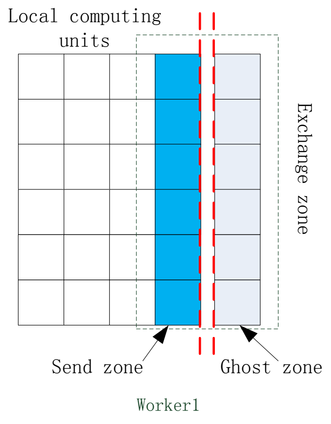
5.1. Computing Unit

A computing unit is the basic data structure in HG, including 4 fields, index, key, value, and neighbors. For GPU implementation, a thread in GPU processes a computing unit. The work zone which consists of all computing units is declared in the HG initialization phase. Figure 5-1 (a) shows a 10x6 work zone. Then the work zone is partitioned into four parts and one part is assigned to each worker. In Figure 5-1 (a), the total number of computing units is 60. HG uses a partition by column scheme as described in section 4.1. If the ratios of the capabilities of the four workers equals {4:3:2:1}, the four workers get the following number of computing units, namely, 24, 18, 12, and 6, respectively. Worker1 gets the 24 computing units on the left side of the red line labeled 1 in Figure 5-1 (a). Worker1 calls the 24 computing units as local computing units.

In the heat equation application, the neighbors of a point are immediate. However we cannot guarantee all the neighbors of local computing units are in the same worker with local computing units. For example, in Figure 5-1 (a), the first four columns of computing units are in worker1 and we define these computing units as local computing units for worker1. Some neighbors of computing units which are on the right boundary of worker1 are in worker2. We define these neighbors as remote neighbors. We call the collection of remote neighbors as the *ghost zone* which is colored as light gray in the right side of Figure 5-1 (b), and the collection of local computing units which the worker sends to other workers as *send zone* which is colored as light blue in Figure 5-1 (b). The exchange zone of a worker includes its ghost zone and send zone.



(a) Partitioning work zone



(b) The exchange zone of worker1

Figure 5-1. Partitioning work zone and exchange zone of a worker for heat transfer

For updating neighbors of computing units, there are three steps. Firstly, workers call the updating neighbors component, a runtime API implemented by the users to fulfill the requirements of their applications. Secondly, a worker finds its local computing units' remote neighbors and merges them together into the ghost zone. If the ghost zone is the same as at the last iteration, three is no need to send its ghost information to other workers. However If the ghost zone is changed, the worker has to communicate with other workers by sending computing units to update the exchange zone.

5.2. Architecture

The architecture of HG is shown in Figure 5-2. HG has two roles, master and worker. A worker is one GPU and a master is one of the workers. The system uses MPI (Forum 2009) as a communication tool that supports scalability very well and each MPI process manages one worker or GPU. HG has three phases, namely, initialization, computation, and finalization.

Initialization phase. The initialization phase consists of initializing the system and tasks. In the initializing system step, HG analyzes hardware information of workers and partitions tasks. In the initializing tasks step, HG reads data from source input and constructs the exchange zone. The exchange zone is defined in the next step. In the initializing system step, the master first gets the speed functions of all GPUs.

Partition. Using FPMC, the master decides the proportion of the workload to be assigned to each worker. The master reads the input workload size information such as the dimension of the job, a 2 or 3-D grid. If no workload size information is provided the system treats it as one dimension and the total workload size is the number of lines of each input file. The master uses a partition module which is controllable by the user or the system to provide a proportion partition algorithm as default to split the input workload into pieces and assign them to different workers. The proportion partition algorithm is a basic simple partitioning algorithm in this work. First, the master assigns an UID (unique ID) to each computing unit. Then the master can assign to each worker's its number of computing units on a proportional basis.

After receiving computing units a worker uses the initialization module to initialize each computing unit. The initialization module is an API that the programmer uses to initialize the computing unit's key, value and neighbors. Since computing units may need to communicate with their neighbors, local computing units should be extended to include their remote neighbors. We define a remote neighbor to be one which is not in the same worker. We call these neighbors as the exchange zone. The worker uses neighborhood information of each computing unit to create the exchange zone. Finally, the worker copies these data and information to GPU memory.

Computation phase. Following the initialization phase, the computation phase has three steps: updating the exchange zone, GPU computing, and checking the termination condition. At the first step, each worker will send keys and values of the exchange zone to other workers. Then according to the kernel computing module (part of the API) provided by the programmer, workers compute units' values in parallel. After each iteration, the system checks if the iteration number has converged to the termination condition. If so, the system merges all workers data, stores the output in local disk or memory and releases all resources, which is the finalization phase.

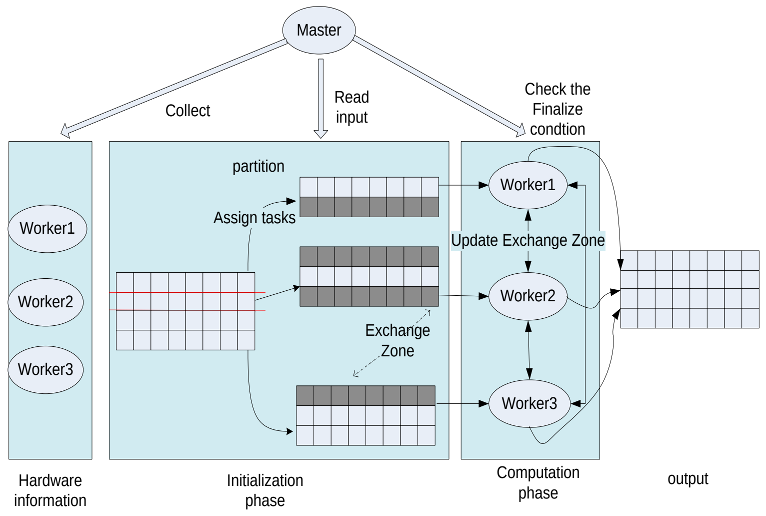


Figure 5-2. Dependent structured grids tasks model processed by HG

For example, in Figure 5-2 the HG system has one logic master and three workers. The capability of worker 3 at bottom of Figure 5-2 outperforms the others by 100%. Worker 1 and worker 2 have the same capability. Hence, when the workload size is 32 units as shown in Figure 5-2, HG will assign 16 units to worker 3 and 8 units to each of the other two workers. The different partition algorithms produce different results which can affect HG performance significantly. Using APIs provided by HG the programmer can implement the specific partition algorithm depending on the application. After receiving tasks each worker extends local units to cover their neighbors which are colored as darker grid points or computing units (the exchange zone) in Figure 5-2. Hence worker 1 and 3 each has 8 exchange units and worker 2 has 16 exchange units. As can be seen in Figure 5-2, the exchange zone for worker 2 is computing units from workers 1 and 3 (and similarly for workers 1 and 3). On completing the initialization phase HG enters the computation phase. When it meets the termination condition HG will merge all outputs and release all resources.

The HG system uses MPI as a communication tool which supports scalability and each MPI process manages a worker or GPU. HG has three phases, namely, initialization, communication, and computation. The work flow of HG as shown in Figure 5-3 consists of four parts. The initialization phase includes four sub-processes presented in the green box on the left side of Figure 5-3. The box of update Units neighbors means changing or updating the neighbors of computing units. In the communication phase workers update the values in their exchange zones. The dotted lines in Figure 5-3 represent two workers who use the internet to communicate with each other. The last part is the computation box in which HG calls the CUDA kernel functions and uses GPUs to process local computing units.

In the initialization phase, HG partitions the work zone based on each worker's capability. So our first concern is quantifying the capability of each worker. HG can use the Constant Performance Model, CPM (see chapter 3) where a positive number is used to express the speeds of GPUs or HG can use FPMC, Functional Performance Model (see chapter 4), as speed function. If the user wants to use CPM, he is required to set the α, β and δ in (3-1). Otherwise a table of speeds against workload sizes and communication cost should be provided as mentioned in section 4.2.

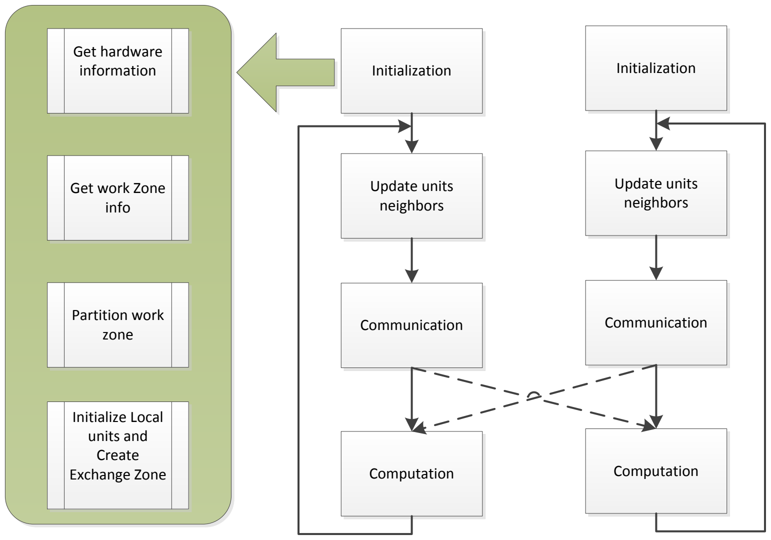


Figure 5-3. The work flow of HG

After collecting each worker's capability $C_{workerID}$, HG uses the partitioning module to get each worker's work zone. HG provides three partitioning schemes for constant performance model, sequential partitioning, row or column based partitioning and NPHP which are described in section 3.4 and a functional performance model with communication presented in section 4.3. The users are able to implement their own partition schemes using HG APIs. Then a worker uses the initialization module to initialize each computing unit. The initialization module is an API that the user uses to initialize the computing unit's key, value and neighbors. When all of the local computing units finish changing their neighbors, HG will reconstruct the exchange zone. This is optional. If the neighbors in the application are fixed, this step will be skipped. Finally, the worker copies the initialized data and information to GPU memory and processes computing units by the computation APIs. When this computation is finished, HG checks the finish condition, such as the iteration number. If the finish condition is not met, HG goes to the next iteration until the finish condition is satisfied.

5.3. System analysis

In this section, host denotes the CPU side and device represents the GPU side. On the host side, a worker needs to store all computing units' location information in the workers. Additionally, for communication, a worker uses the sending and receiving values lists to update the exchange zone. If neighbors of computing units are not fixed, a worker has to construct the sending and receiving indices lists every time.

On the device side, a worker first initializes the value offsets of local computing units. If the sizes of values are the same this step is not necessary. Then all computing units are stored on the device memory. The neighbors of local computing units are determined next. The set of neighbors is not needed when the neighbors' relationship is determined by a formula. A *G2LocalUnitID* is needed to convert the global computing unit ID to local computing unit ID when HG does not apply the sequence partitioning and columns or rows based partitioning. We easily know the range of local IDs under these two partitioning algorithms, because the local IDs are continuous in each worker. When we try to get the value of a computing unit with ID x, we just need to compare x to the first ID and the last ID of local computing units. If it is in the range, then we can convert global ID to local ID: $localID = x - firstID$; Otherwise this is a remote ID which is not in the local neighbor list and gets the value from the receiving value list. If the neighbors of all computing units are dynamic, the receiving and sending IDs lists are built between iterations. Otherwise they are fixed and built in the initialization step. However the receiving and sending values lists are updated every time. Firstly, the device gets the sending values and copies them to the host side. The host exchanges them and receives some values. Finally, the host transfers the receiving values to the device side.

Table 5-1. The requirement of space for HG

Name	Space	Location	Description
UnitID2Worker	Number of global computing units	Host	In constructing the exchange zone steps, locate workers of the remote neighbors.
Computing Units	Number of local computing units.	Device	In computation steps, find the offsets of values of computing units. Optional if the sizes of values are the same.
Values of Computing Units	Sum of all sizes of values of local computing units.		The values of local computing units
Neighbors of Computing Units	Sum of all numbers of neighbors of local computing units		Optional if the neighbors relationship are conducted by a formula
G2InnerUnitID	Number of local computing units.		Convert the global computing unit ID to the local ID. Optional if HG applies the sequence partitioning and columns or rows based partitioning.
Receive IDs list	Number of receive computing units in an iteration	Host and Device	According to this list, a worker notifies the other workers which computing units it needs.
Send IDs list	Number of send computing units in an iteration		According to this list, a device gets the send values from the device and copies to send values list
Receive values list	Number of Receive values size in an iteration		Receive the values from other workers
Send values list	Number of send values size in an iteration		Send the values to other workers

In short, the space of HG is dependent on the applications. In these following formulae, n' is the number of local computing units and n is the number of global computing units. If the applications, PDE (partial deficiency equation) and heat transfer, have fixed sizes of values, fixed neighbor relationship, and continuous local IDs, the requirement of space on the host for a worker is $exchangeZoneSize$, where we ignore the space for the worker location of global computing units, because it's used only once in the initialization step to build the exchange zone. On the device the space is $valueSize \times n' + exchangeZoneSize$.

For the applications, matrix operations and Gaussian Elimination, which have fixed sizes of values and continuous local IDs, and the dynamic neighbor relationship can be formulated, the requirement of space on host for a worker is $n + 2 \times exchangeZoneSize$and on device it is: $valueSize \times n' + 2 \times exchangeZoneSize$.

The space on the host for the worst case is $n + 2 \times exchangeZoneSize$. On the device it is $\sum_{i=0}^{n'}(valueSize_i + neighborSize_i) + 2 \times n' + 2 \times exchangeZoneSize$, where $2 \times n'$ means we have to store the offsets of values and neighbors of local computing units.

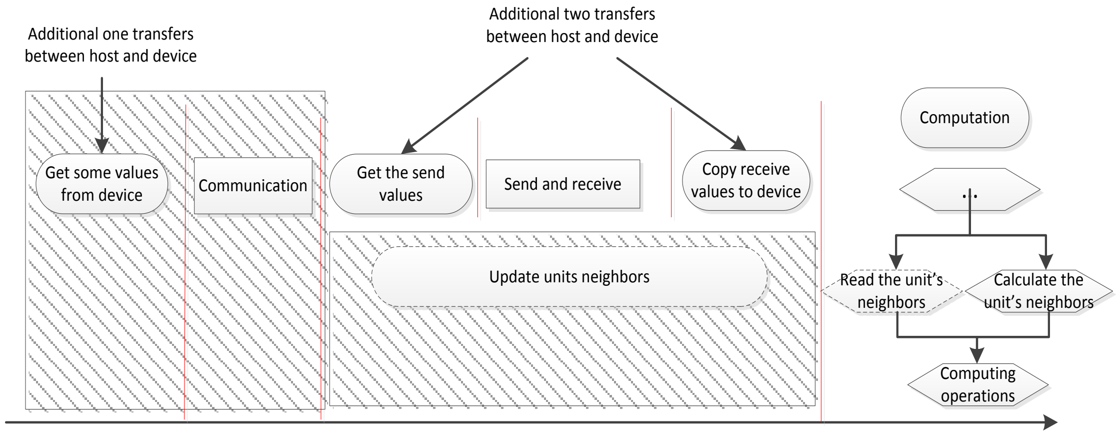

Additional one transfers between host and device
Additional two transfers between host and device
Get some values from device
Communication
Get the send values
Send and receive
Copy receive values to device
Computation
..
Update units neighbors
Read the unit's neighbors
Calculate the unit's neighbors
Computing operations

Figure 5-4. The *i*th iteration of HG

Furthermore, the operations in HG are also dependent on applications. The overview of operations at the *i*th iteration of a worker is shown in Figure 5-4. The axis denotes time and the vertical red lines represent the barriers where operations must wait for the previous operations. The slash zones represent the dynamic operations as in Figure 5-4 and the ellipses denote these operations working on the device; the rectangles denote these operations working on the host; the hexagons denote the sub-operations of the computation step. The hexagon and cylinder with dotted lines, read the unit's neighbor and update units neighbors, which mean they are optional. If the applications need a dynamic neighbor component they should use them. In the computation step, there are two methods to locate the unit's neighbors, formula based or reading from the neighbor set. If neighbors are able to be formulated it is better to use the formula, because it can save the usage of device memory and the time of memory accesses. The tradeoff is we need more computing operations to calculate neighbors. Our experiments show that using formula is better than neighbor set.

Compared to heat transfer running on CPU, HG requires additional two transfers between host and device to get the send values and copy the receive values to device. We use (5-1) to estimate the execution time of GPUs system and (5-2) for CPUs system. When compared to Gaussian Elimination implemented on CPU, HG adds additional two transfers between host and device.

$$T_{GPUs} = max\Big\{(networkBandwidth_i + 2DeviceBandwidth_i) \times exchangeSize_i + \frac{workload_i}{GPUprocessingSpeed_i}\Big\}$$

where T_{GPUs} represents the execution time of GPUs.

if

$$workload_i = \frac{workload \times GPUprocessingSpeed_i}{\sum_{i=0}^{m} GPUprocessingSpeed_i}$$

Then,

$T_{GPUs} =$

$$max\left\{(networkBandwidth_i + 2DeviceBandwidth_i) \times exchangeSize_i + \frac{workload}{\sum_{i=0}^{m} GPUprocessingSpeed_i}\right\} \quad (5\text{-}1)$$

And

$$T_{CPUs} = networkBandwidth \times \ exchangeSize + \frac{workload/n}{CPUprocessingSpeed} \quad (5\text{-}2)$$

Almost all applications implemented on GPUs keep data in GPU memory and avoid transferring them between CPU and GPU because the band width between CPU and GPU is dramatically less than the processing speed of GPU. For example, the bandwidth of PCI Express 2 is up to 8 GB/sec for an x16 device. However, it’s expected with the increasing bandwidth the bottleneck will be overcome. Right now the bandwidth of PCI Express 3 doubles the previous version which is up to 16 GB/sec. Hence, if the total processing speed of GPUs is larger than the total processing speed of CPUs, GPU can easily get tens or even hundreds speedup over CPU, when the exchange zone (the area where data needs to move betwwen GPU and CPU) sizes are not very large, the distributed GPUs system prevails over the distributed CPUs system in terms of speed.

We ignore the overhead of the network transfer part in (5-1) and (5-2) because almost all clusters run in a private cloud environment. They are connected by an infinite band where network bandwidth is significantly fast. Moreover the sizes of exchange zones of CPUs and heterogeneous GPUs are very close when the capabilities of GPUs are very close and the same partitioning algorithm is applied such as by rows or columns if the workload is a matrix.

The study of heterogeneous GPUs system is shown below. If the application is heat transfer and the workload is n×n matrices, network bandwidth is 40GB/sec, device bandwidth 16GB/sec, the size of exchange zone is n specified by a row. In (Cowboy 2013), the CPU is an Intel Xeon E5-2620 “Sandy Bridge” whose peak speed is 96 GLOPs. The processing speed of different types of

GPUs is listed as follows: 1 Fermi in Tesla M 2050: 1.03T GFLOPs;1 Fermi in Tesla M 2090: 1.33T GFLOPs; 1 Kepler GK 104s in Tesla K10: 2.29T GFLOPs; 1 Kepler GK 110 in Tesla K20x: 3.95T GFLOPs. The Fermi and Kepler are the serial types of GPUs.

In figures 5-5 and 5-6, the workload is a 2-D square matrix and its size is 256,000x256,000. The x-axis denotes the number of CPUs and the y-axis is the speedup of GPUs over CPUs which is calculated as the CPUs execution time divided by the GPUs execution time. Fox example in Figure 5-5, "CPU/16 M2050" represents the 16 Tesla M2050 over multiple CPUs and "CPU/32(K20+M2050)" means 32 Tesla K20x and M2050 over multiple CPUs. The red dash line in Figure 5-5 is the base line of "1". The GPUs have more GFLOPs than CPUs. A Tesla M2050 and a Tesla K20x are 10x and 40x faster than an Intel Xeon E5-2620 respectively. In the study, when the number of CPUs is up to 256 and the GPUs consist of 32 K20x and M2050, respectively, the heterogeneous GPUs system is able to get almost 5x speed up. With increasing the number of CPUs, CPUs are able to exceed GPUs system. However the more number of CPUs requires more energy consumption and expenditure. When increasing the number of GPUs and CPUs we simplify (5-1) and (5-2) to (5-3) and (5-4).

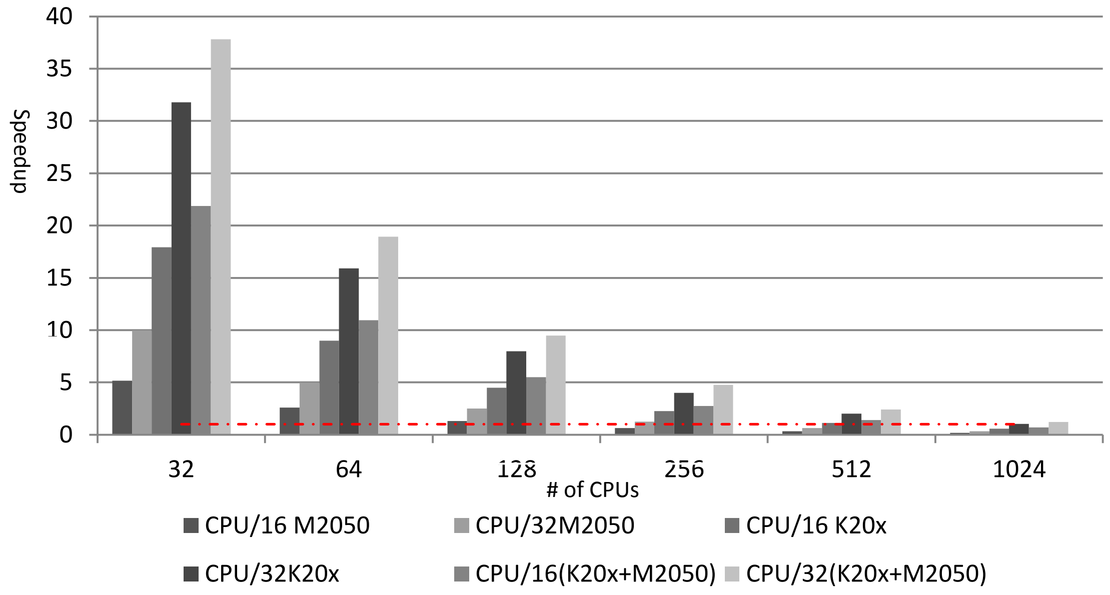


Figure 5-5. The expected speedup of GPUs over CPUs

$$lim_{m\to\infty} T_{GPUs} = max\{(networkBandwidth_i + 2DeviceBandwidth_i) \times exchangeSize_i\} \quad (5\text{-}3)$$

$$lim_{n\to\infty} T_{CPUs} = networkBandwidth \times exchangeSize \quad (5\text{-}4)$$

When the numbers of CPUs and GPUs are large, the execution time of CPUs is decided by the network communication time, and the execution time of CPUs is decided by the network communication time pluses the data movement time between CPU and GPU which is slower than network communication. With the improvement of PCI-E technology, this gap should be dramatically reduced in the future.

We compare different homogenous and heterogeneous GPUs system to show the improvement of the heterogeneous GPUs system. In Figure 5-6, “32xM2050” means the system consists of 32 Tesla M2050 GPUs which is considered as base line. The y-axis represents the speedup obtained which is the execution time of 32 Tesla M2050 GPUs divided by the execution time of GPUs system with different GPUs combinations. The x-axis represents the different sizes of the 2-D workload where one unit is a thousand.

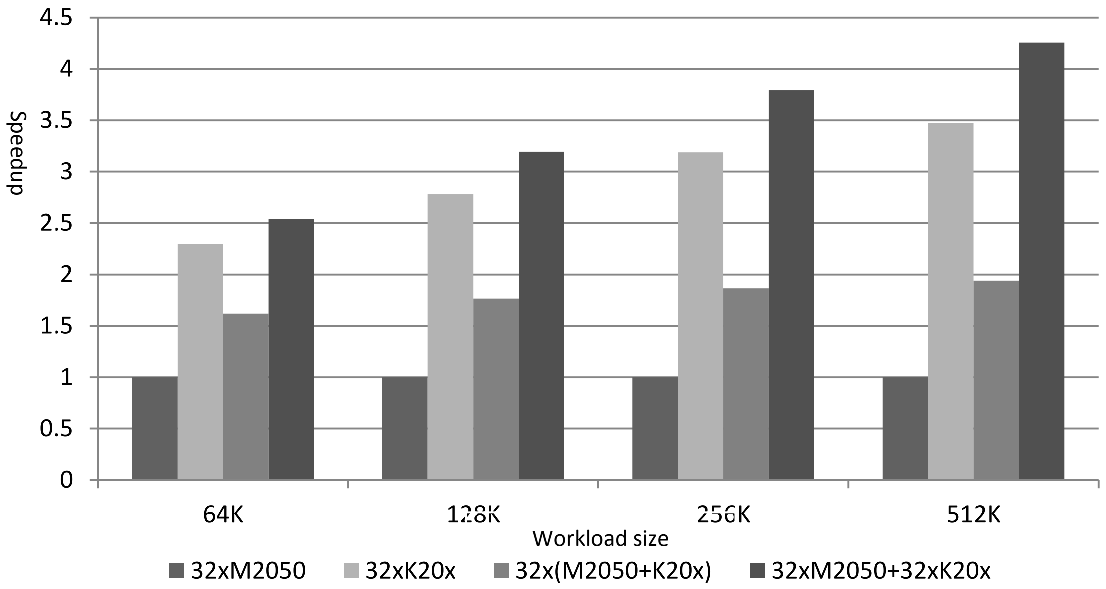


Figure 5-6. The speedup of GPUs over 32 Tesla M2050 GPUs

A Tesla K20x has almost 4 times of GFLOPs as a Tesla M 2050's. Therefore, the speedup of 32xK20x is up to 3.5 when the workload size is 512K. "32x(M2050+K20x)" means the system consists of 32 Tesla M2050 and 32 Tesla K20x, respectively. The evenly partitioning algorithm is applied and the workload in M2050 is the same as it is in K20x. The performance of 32x(M2050+K20x) is worse than 32xK20x because K20x should wait for M2050 to finish. From a theoretical view, the 32x(M2050+K20x) system equals to 64xM2050 system. The GPUs in "32xM2050+32xK20s" are the same as the "32x(M2050+K20s)" system, but it applies a different partitioning algorithm which can partition a workload depending on the computing capability of GPUs. Hence the partitioning algorithm results in the workload in K20x being more than it is in M2050. If we combine them together to make a heterogonous GPUs system, their performance is improved. From Figure 5-6, the heterogonous GPUs system with 32 Tesla K20x and M2050 has the best performance.

5.4. Configuration

HG provides parameters which can be specified by the user for different applications or performance tuning. Table 5-2 lists some of the major parameters. For example, the heat equation application can run in two-dimensional space in which case the parameter of *Job_Dimension* is set to 2. This parameter affects how neighbors are configured. When the job is 1 dimension, neighbors of a computing unit should be only left and right neighbors. If the *Job_Dimension* is 2, the neighbors should include top and bottom neighbors also. *Job_dimension* also impacts the set *Job_Size* which is the value of each dimension of a job. *Job_Size* can be described as x, y and z where *Job_dimension* is 3. If *Job_dimension* is set as 1, y and z are set as default value 1 and the default value of x is the number of source files. The only dimension of interest therefore is x.

Table 5-2.The configuration parameters

Parameter	Description	Default
Job_Dimension	Value of job physical dimension which is from 1 to 3.	1
Job_Size	Number of computing units in each dimension.	Lines of source file
Input_path	Direction of input file.	Project folder/input.in
Output_path	Direction of output file.	Project folder/output.ou
nBlock	Number of thread blocks.	512
nThread	Number of thread per block.	256
nNeighbor	Number of neighbors per unit.	4

Input_path is the path to a file or folder. In other words, the source folder can have multiple files. However *Output_path* just supports a single file, so all results must be stored in a single output file. Users can tune *nBlock* and *nThread* parameters for application performance. These two parameters are related to GPU programming and this is low level information which we want to hide. However these two basic parameters are very important to application performance and so users have the option to set them.

5.5. APIs

The system provides two types of APIs, user-defined and runtime APIs. Table 5-3 shows some of the major API functions. User-defined API functions should be minimal and simple to use. The system provides three user-defined functions which cover the three phases of the system, initialization, computing and finalizing. Furthermore, users can define their own partitioning functions for specific applications. Runtime API functions are more complex. Runtime APIs allow users to input data into the system and update the values and neighbors.

Table 5-3. APIs

Type	APIs	Required
User-defined	__device__ void initializeUnit(int unitID,WC_UNIT_T *unit, char* inputs) Initialize local unit using the input data in inputs array which are assigned to the worker.	Yes
	__device__ void ComputingKernel(int unitID, int runTime) Run on GPU and be the GPU kernel function.	Yes
	bool finishCon(Job job) Set the termination condition.	Yes
	int* partitionJob(float[] workerCapabilities, int workerID, WC_UNIT_T *units, char *inputs) Partition and assign tasks.	No
	__global__ void neighborDynamic(int unitID, int runTime) Device kernel to update neighbors of local computing units	No
Runtime	void finalize() Merge all the results and release allocated memory.	Yes
	void addInput(WC_UNIT_T *units, VALUE_T* value) Add a computing unit with a value to the computing unit structure in GPU memory.	Yes
	bool getValueFromDevice(int valueSize, int *IDlist, VALUE_T* values) Get values whose IDs are in the IDlist from device to host.	No
	bool copyValueToDevice(int valueSize, int *IDlist, VALUE_T* values) Copy values whose IDs are in the IDlist from host to device.	No
	__device__ int writeValue(int unitID, VALUE_T *value, int valueSize) Update value for the computing unit.	Yes
	__device__ int getValue(int unitID, VALUE_T *value) Find the value of unit.	Yes
	__device__ void setNeighbor(int unitID, int *neighbors, int neighborSize) set unit's neighbors	No

At a minimum, users are required to implement just two main functions: *initializeUnits* to initialize local units and *ComputingKernel* function to make GPU compute local units' values.

Algorithm 5-1 and 5-3 present the pseudo code of *initializeUnits* and *ComputingKernel* function for computing the average of neighbors in 2 dimensions.

Algorithm 5-1 Initializing computing units (*initializeUnit*)

```
Input:
        set of local units ID, unitIDs;
        set of input data, input;
        Job information,  jobInfo;
1:   for each i in unitIDs do
2:       Unit unit;
3:       unit.key = i;
4:       Set the location of the unit using jobInfo;
         //if the local computing units' neighbors cannot be located by formula they should use setNeighbor
function         to add their neighbor location to the set of neighbors. Otherwise omit setNeighbor function.
5:       setNeighbors(unit, up, down, left, right);
6:       Set unit value using input and get the value size;
7:       addInput(unit, value Size);
8:   end for;
```

We have two ways to initialize computing units based on how to locate neighbors. If the neighbors are able to be located by formula, it's not necessary to use additional space to store the neighbor relationship. So we can omit *setNeighbor* function to initialize the locations of computing units' neighbors. Otherwise we have to use *setNeighbor* function to store the neighbor relationship of all computing units. In algorithm 5-1, *initializeUnit*, the user sets the values of fields such as location, key and neighbors' ID. Then the user uses the *addInput* runtime API to add an initialized computing unit to the local computing units set.

Some types of applications need to change the neighbors of computing units when they are running. So we need a dynamic component to change the neighbors. For example, in Gaussian Elimination all the neighbors depend on the pivot. The dynamic component is used to select the pivot as in the pseudo code of algorithm 5-2.

Algorithm 5-2 Dynamic component in Gaussian Elimination (*ComputingKernel*)

```
1:   get the values of current column from device;
2:   merge them to a worker;
3:   sort to select the pivot;
4:   announce the current pivot;
5:   exchange pivot row;
6:   copy the pivot values to device;
```

In the *ComputingKernel* function, the user uses the *getValue* runtime API to get the value of a computing unit and uses *writeValue* runtime API to update the value. The pseudo codes of algorithm 5-3 and 5-4 are the *ComputingKernel* functions of the Gaussian Elimination application. The difference is the way to determine the neighbors. Algorithm 5-3 uses a formula and algorithm 5-4 uses a neighbor set. The tradeoff of algorithm 5-3 is we should get the location of the computing unit first and will need more computational operations to calculate the neighbors' locations. Its benefits are saving storage, and reducing memory accesses. Our experiments show that algorithm 5-3 performs better than algorithm 5-4.

Algorithm 5-3 Computing kernel of Gaussian Elimination using formula to get the neighbor

```
Input:
        Local unit ID, unitID;
        Current iteration, i;
        Job information, jobInfo;
1:   get the location information of unitID, x and y;
2:   result = getValue(unitID);
3:   calculate pivotID using $pivotID = i \times jobInfo.y + i$;
4:   pivot = getValue(pivotID);
5:   get the neighbor ID in the same row using $neighborID = i \times jobInfo.x + i$;
6:   rowValue = getValue($neighborID$);
7:   get the neighbor ID in the same column using $neighborID = i \times jobInfo.x + x$;
8:   colValue = getValue($neighborID$);
9:   result = result-colValue$\times \frac{rowValue}{pivot}$;
10:  writeValue(unitID, result);
```

Algorithm 5-4 Computing kernel of Gaussian Elimination using neighbor set to get the neighbor

```
Input:
        Local unit ID, unitID;
        Current iteration, i;
        Job information, jobInfo;
1:   result = getValue(unitID);
2:   get pivotID using pivotID =getNeighbor(unitID, 0);
3:   pivot = getValue(pivotID);
4:   get the neighbor ID in the same row using $neighborID$= getNeighbor(unitID, 1);
5:   rowValue = getValue($neighborID$);
6:   get the neighbor ID in the same column using $neighborID$= getNeighbor(unitID, 2);
7:   colValue = getValue($neighborID$);
8:   result = result-colValue$\times \frac{rowValue}{pivot}$;
9:   writeValue(unitID, result);
```

5.6. Data reuse

For the third challenge, we use a tiling technique proposed by (Kirk and Hwu 2010) where threads in a thread block use shared memories to reduce the accesses of global memory. CUDA utilizes four types of memory, global, constant, shared memory, and registers. The global memory of GPU features long access latencies, up to hundreds of clock cycles and finite access bandwidth. The long access latencies of global memory make it's impossible to use global memory directly on a device. However shared memory is very fast. In (Kirk and Hwu 2010), Kirk proposed a tiling technology to load the data from global memory to shared memory and the threads in the same block are able to access the shared memory instead of the global memory. In this technology most of the threads in the same block access the same location of global memory.

For example, in *ComputingKernel* of heat transfer, computing units load their own values from global memory first and then load their neighbors values from global memory again before starting computation. Actually, there is excess global memory traffic because each value is loaded five times from global memory by different threads. If the computing units C_1 and C_2 are adjacent, C_1 and C_2 load their values from global memory separately. Then C_1 loads its neighbors' values including C_2 from global memory. For C_2, there are two global memory accesses. The idea of data reuse is firstly, computing units load their own values from global memory to shared memory. Then they load their neighbors values from shared memory instead of global memory.

5.7. Conclusion

In this chapter, we have proposed a novel platform for processing dependent tasks on heterogeneous GPUs system based on neighborhood grid points. It supports change of neighborhood when the system is running. Also we implement some optimizations to improve its performance. We integrate our previous partitioning algorithms to ensure load balance for heterogeneous GPUs as well as reducing their exchange zones. We use the data reuse in computing kernel to utilize shared memory to reduce global memory accesses.

CHAPTER VI

EVALUATION

Using the heat transfer and Gaussian Elimination application, we show the partitioning algorithms and optimizations to improve the performance of HG system. Then we evaluate HG compared to homogenous, heterogeneous GPUs systems, and CPU-based counterparts.

6.1. Experimental setup

Our test-bed consists of two computing nodes. In each node, the CPU is an Intel Xeon 2.4 GHz and 4 cores with 6GB memory. Additionally, in each node we have a Tesla C2050 GPU card and a GeForce GTX650 GPU card. The memory of Tesla C2050 is 2687MB with 384 bits memory bus width, and GTX650 is 1024MB with 128 bits memory bus width. Their numbers of CUDA cores are 448 and 384, with 1.15GHz and 1.05GHz GPU clock speed, respectively. Obviously, C2050 is more powerful than GTX650. So we use the four GPU cards to make a heterogeneous distributed system. We connect the two nodes with a 100Mbps router and the average latency is around 0.2 ms which is much higher than a standard supercomputing center.

To test the effect and efficacy of our system, we use the 2-D heat transfer and 2-D Gaussian Elimination application with different workload sizes. For each experiment we record the time of each phase of HG, such as initializing, communicating, and computing. We also compare the total execution time with the MPI programs running on the CPU. For the heat equation experiment, we set the time steps to 1,000 and change the number of computing units from 100 thousand to 41 million. And for the Gaussian Elimination application, the range of workload size is from 1 million to 41 million.

We first compare the performance of the systems with and without optimizations including partitioning algorithms and data reuse. We run the heat transfer application to show the speedup of the system with the partitioning optimization and show the performance improvement within data reuse in the heat transfer application and the Gaussian Elimination application. In the following figures, Comp. represents computing time, Comm. is communication time and init. is initializing time.

6.2. Result of NPHP partitioning algorithm

In this experiment, we run the heat transfer application applying different partitioning algorithms on a heterogeneous GPUs system which includes two C2050s and two GT650s. The capability of the GPU is measured by FPM which is the experimental results presented in section 4.2. Thus, a C2050 card is assigned around two and a half times of computing units as a GTX650 card. Then according to their capabilities, partitioning by row, PHP, and NPHP partitioning algorithms as described in sections 3.3 and 3.4 are used. NPHP constructs a smaller exchange zone than partitioning by rows does. Hence, NPHP partitioning algorithm reduces the communication time of the applications where computing units' neighbors are immediate. In Figure 6-1 the by rows partitioning algorithm brings the worst performance and PHP is better. For all workload sizes, NPHP achieves the shortest communication time. For example, when the workload size is 6400x6400 computing units, NPHP partitioning algorithm improves to reduce the communication time up to 3.5 times when compared to by rows partitioning and 10% when compared to the PHP partitioning algorithm.

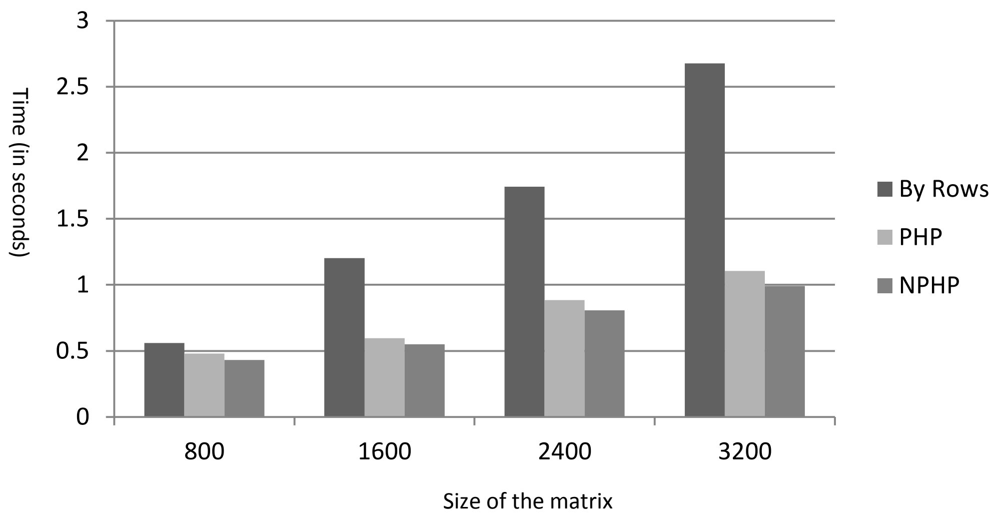


Figure 6-1. The communication time of Heat transfer applications using different partitioning

6.3. Result of FPMC partitioning algorithm

We compare the proposed algorithm FPMC with evenly partitioning and previously proposed FPM (P E Crandall and M J Quinn 1994) in this section. In the following figures, comp.-GTX650 represents computing time of GTX650, comp.-C2050 denotes computing time of C2050, comm. is communication time and init. is initializing time. In the x-axis, the system applies different partitioning algorithms with different workload sizes. For example, 800-Even means the workload is a 800x800 square matrix and partitioned evenly. So each GPU gets a 400x400 square matrix. 1600-FPM denotes the workload is a 1600x1600 square matrix and it uses the FPM partitioning algorithm. 3200-FPMC means the workload is a 3200x3200 matrix and it uses the FPMC partitioning algorithm.

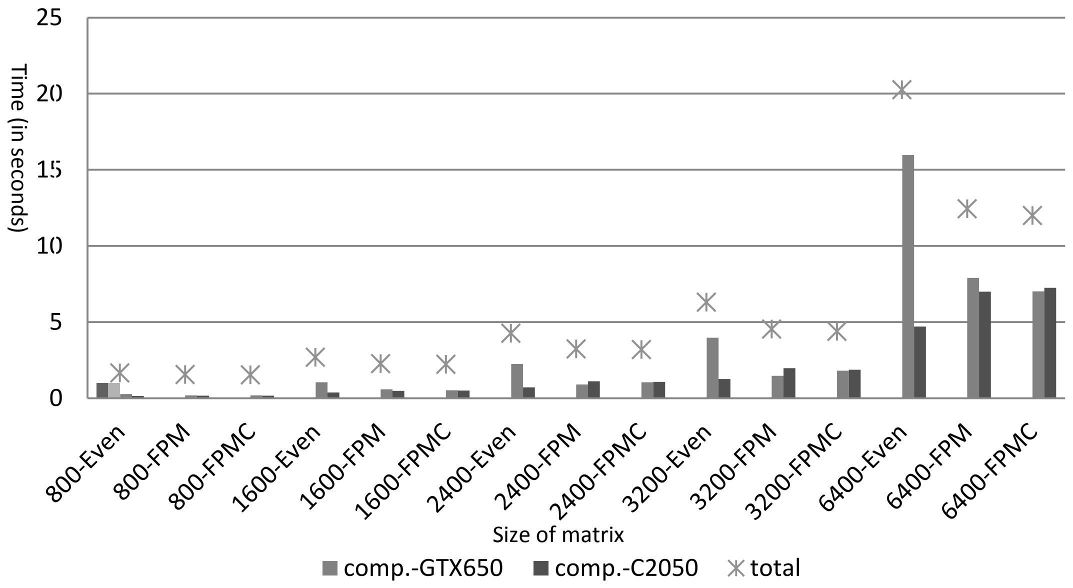


Figure 6-2. 2-D heat transfer using different partitioning algorithms

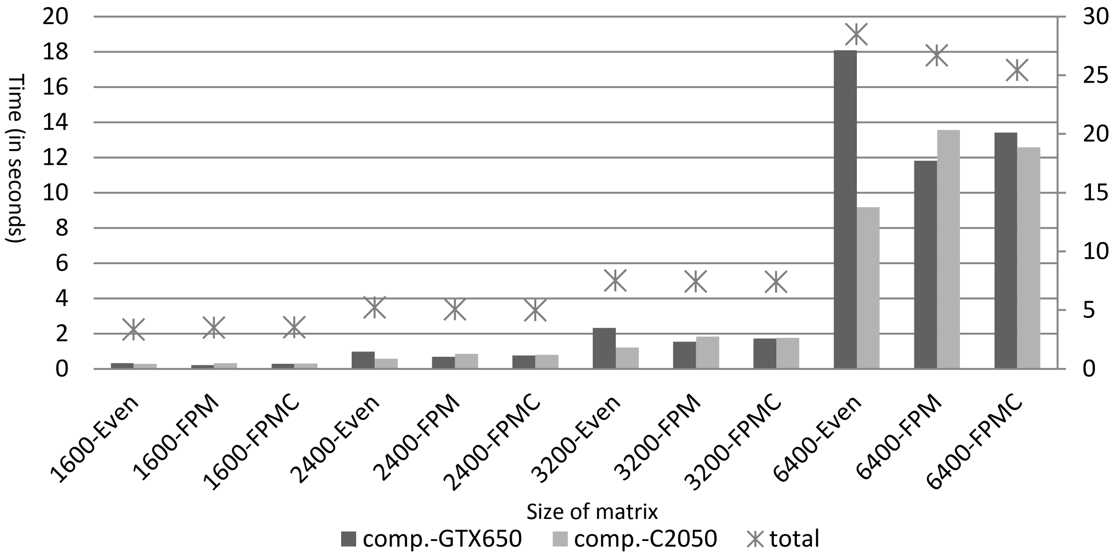


Figure 6-3. 2-D Gaussian Elimination using different partitioning algorithms

In Figure 6-2 and Figure 6-3, the evenly partitioning algorithms causes severe load imbalance. Especially, when the workload size is a 6400x6400 square matrix the execution time of GTX650 is more than 3 time of C2050's in Figure 6-2 and 2 time in figure 6-3. From the total execution time view, FPMC brings the best result because FPMC gets better load balancing.

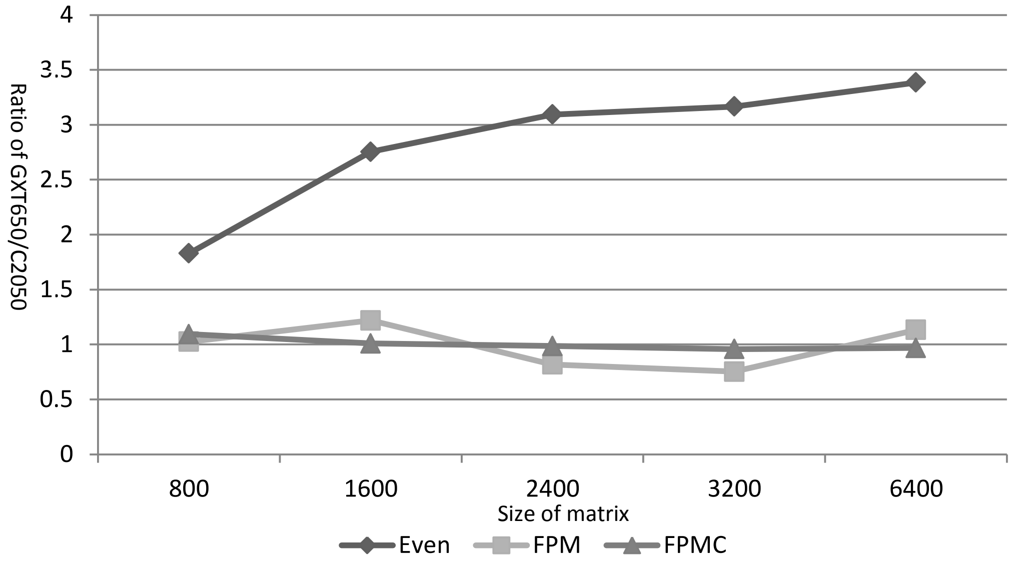


Figure 6-4. Load balance of 2-D heat transfer

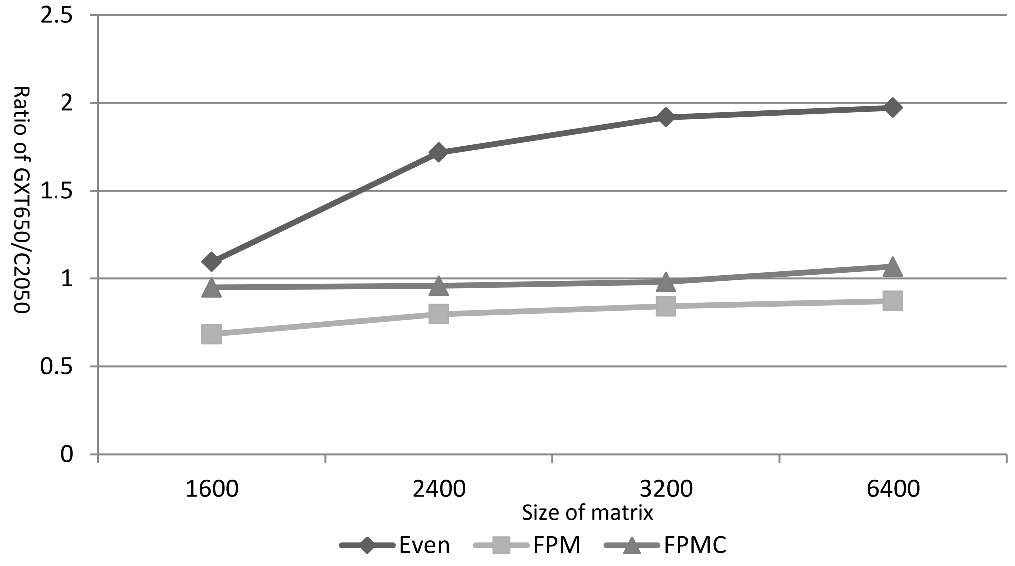


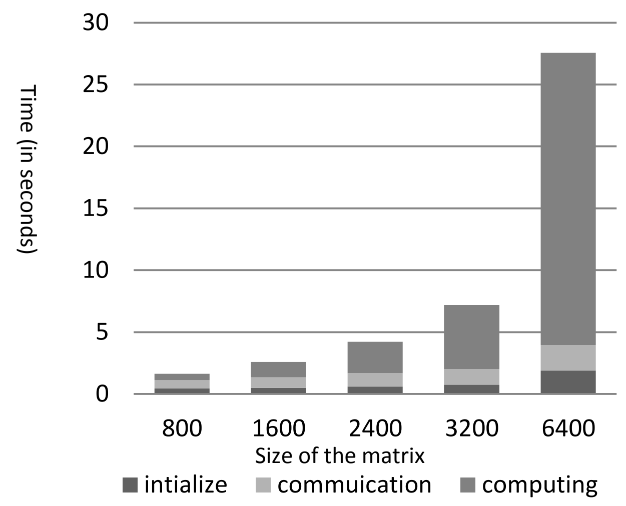
Figure 6-5. Load balance of 2-D Gaussian Elimination

Figure 6-4 and Figure 6-5 show how partitioning algorithms affect the load balance. The y-axis denotes the ratio of execution time of GXT650 and C2050. If the ratio is very close to 1 it means the load is balanced. With increasing workload size, even partition brings more load unbalancing. However FPM and FPMC are very close to 1 and FPMC is closer to 1 than FPM which means

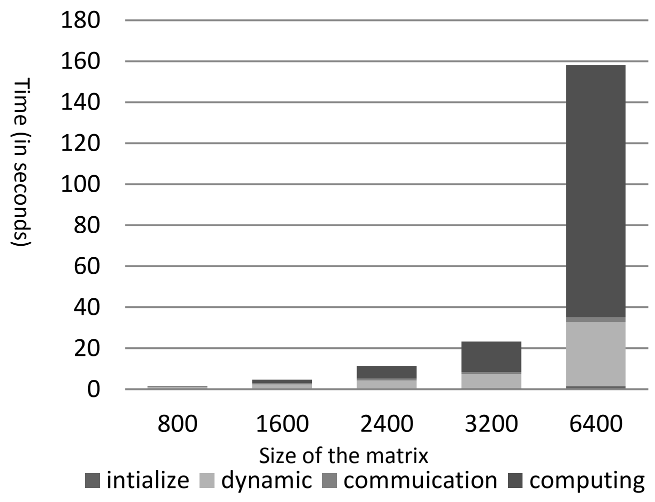
FPMC provides the best load balance. In Figure 6-5 for the 2-D Gaussian Elimination application, when the workload is very small like as in 1600x1600, evenly partition outperformances FPM. This is because of the large communication overhead which affects the performance. In this situation, FPMC can estimate the speed of GPUs more accurately than FPM to achieve better load balance.

6.4. Optimization of data reuse

In this experiment, HG system consists of two C2050s and runs the heat transfer and the Gaussian Elimination applications. The range of workload size ranges from 800 to 6400 square matrix. In Figure 6-6(a) and (b), we measure the execution time at each step of HG system without optimizations and analyze which step is the most time-consuming and how to improve its performance. In Figure 6-6(a) and (b), HG system spends the most of the time in the computing step. In Figure 6-6(b) Gaussian Elimination application, the second most time-consuming step is the dynamic step where HG gets the pivot and updates the computing units' neighbors. In figure 6-6 (a) heat transfer application, when the workload size is up to a 6400x6400 matrix, the computing step takes up to 85.6% of total execution time. When the HG system processes a 6400x6400 matrix for the Gaussian Elimination in Figure 6-6 (b), the computing and dynamic steps take up to 77.7% and 19.8% of execution time, respectively. In the computing step, the computing kernel has a vast number of global memory accesses which are very slow in GPU. Usually, the ratio of float-point calculation to the global memory access operation (RFM) in heat transfer application is 1 to 5, because each computing unit of the heat transfer application has 4 neighbors. For the Gaussian Elimination application is 1 to 4, or 0.25 which means each computing operation needs four global memory accesses. However, if we use the neighbor set to locate computing units' neighbors the RFM increases to 1/7. This is because we need 3 more global memory access operations to get the neighbors locations. We can use data reuse optimization to improve the performance of the computing step.



(a) Heat transfer application



(b) Gaussian Elimination application

Figure 6-6. Results of applications without the data reuse optimization

In Figure 6-7, the data reuse optimization is able to improve the computing performance up to 2.5 times in heat transfer application which is lower than the expected 5x. The reason is we add some logic decisions in the computing kernel section to process the computing units on the boundary. We divide the local computing units to tiles, that is, a tile consists of a number of computing units. Each tile is processed by a block of threads in GPU where these threads can share values through the shared memory in GPU. Each computing unit in a tile firstly loads its own value to shared memory, and depending on its location, the boundary value is loaded. If it is on the grid boundary we should load its remote neighbor to the tile. If it is on the tile boundary we should load its local neighbor to the tile. For the Gaussian Elimination application, the data reuse optimization improves the performance up to almost 7 times. The computing units in a tile share their neighbors through the shared memory of GPU. All of the computing units have three neighbors. All of them have the same pivot. The computing units in the same column share the same neighbor and those in the same row share another neighbor. Thus, there are few decisions to be made in the Gaussian Elimination computing kernel. Additionally, we use a formula to calculate the locations of computing units' neighbors. Hence we reduce the global memory access to only 1 compared to the original version's 7 which is without data reuse optimization and reads neighbors locations from global memory.

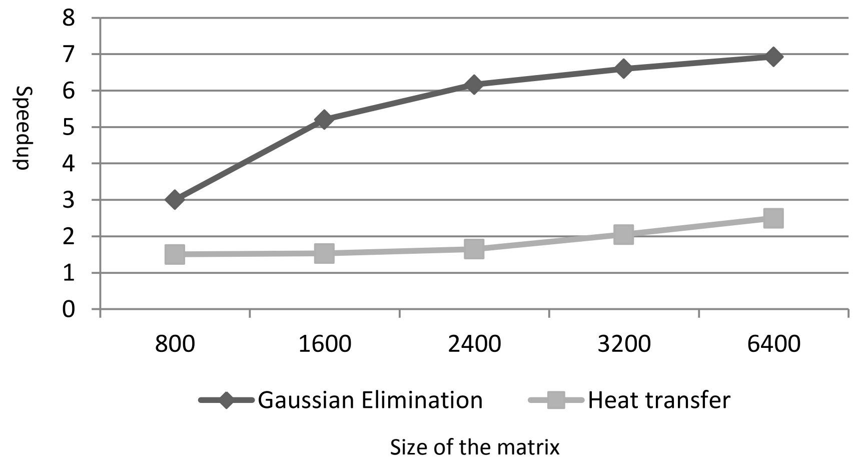


Figure 6-7. Speedup of data reuse optimization

6.5. Heterogeneous and homogenous GPUs system.

In these experiments, we compare the performance of HG system in different GPU combinations. In figure 6-8 and 6-9, "800-2xC2050" means the HG system consists of two C2050 cards and the workload size is an 800x800 matrix. So this is a homogenous GPUs system. "1600-hetero" presents the HG system consists of two C2050 and two GTX650 cards and the workload size is a 1600x1600 matrix. We use the experimental results in section 4.2 to quantify the capability of each GPU card which means a C2050 is assigned approximately 2.5 times more computing units as a GTX650. "2400-4xEven" denotes the HG system consists of two C2050 and two GTX650 cards and the workload size is a 2400x2400 matrix. However we assign the same number of computing units to all of them. "initi." means initiation time and "dyna." is dynamic time. In the same way, "comm." is short for communication time and "comp.-GTX650/C2050" means computing time of GTX650 and C2050, respectively. The asterisk mark "total" means the total execution time.

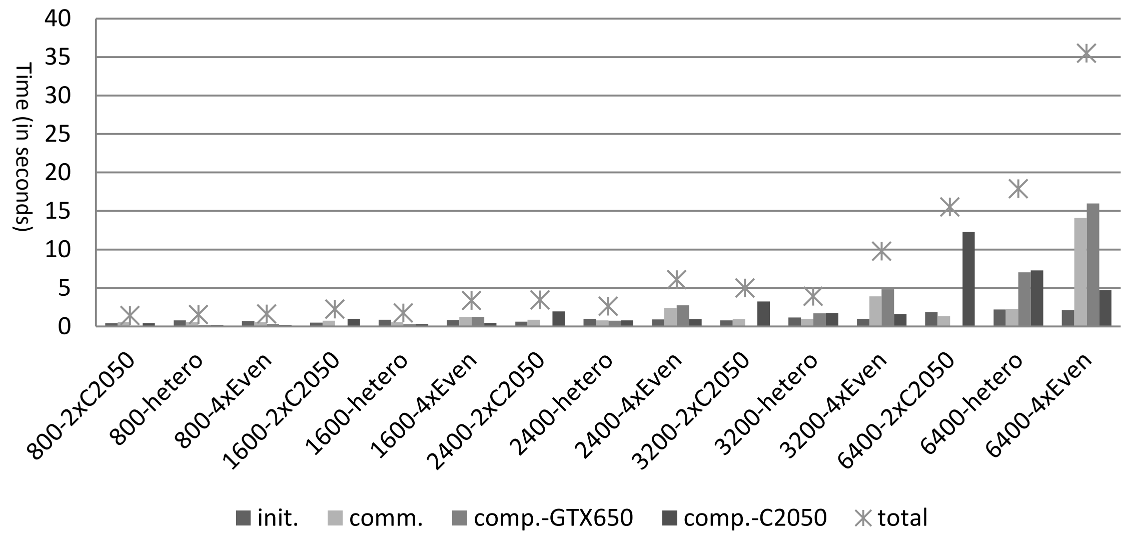


Figure 6-8. 2-D heat transfer in different GPUs combinations

It is not true that more GPUs mean better performance. In figure 6-8, in most situations, the four GPUs system with same workload size needs more time to execute than two C2050s GPUs system. Because we assign the same workload size to each GPU, the slow GTX650 becomes the bottleneck. Additionally, more GPUs increase the communication time. If we assigned appropriate workload size to GPUs based on their capabilities, the heterogeneous system is expected to improve the performance. In figure 6-8, although the communication time of heterogeneous system is a little longer than the two C2050s system, the computing time of the heterogeneous GPUs system is much shorter than the two C2050s homogenous system. We get the same conclusion when we run Gaussian Elimination. In Figure 6-9, the best performance is created by the heterogeneous GPUs system. The second beat performance is delivered by two C2050s homogeneous GPUs system. The worst performance is given by the four heterogeneous GPUs system where each GPU is assigned the same workload.

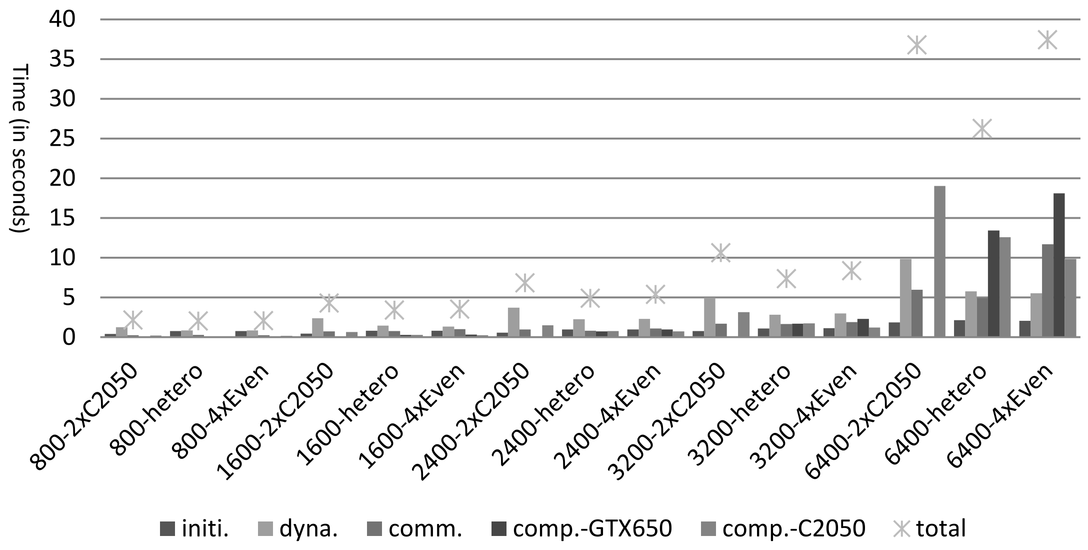


Figure 6-9. 2-D Gaussian Elimination in different GPUs combinations

We use a parameter, improvement percentage which is calculated by (6-1) to estimate the performance improvement by the heterogeneous GPUs system over the homogenous system.

$$improvement\ percentage = \frac{T_{homo} - T_{heter}}{T_{homo}} \tag{6-1}$$

where T_{homo} is the execution time of the homogenous system and T_{heter} is the execution time of the heterogeneous system.

In Figure 6-10, the heterogeneous GPUs system consists of two C2050s and two GTX650 respectively and the homogeneous GPUs system includes two C2050s. If we add two cheaper GPUs into the original homogeneous GPUs system we see an improvement of 14% in the heat transfer application and even up to 30% performance improvement in the Gaussian Elimination application. The price of GTX650 is just around one hundred USD comparing to the price of

C2050 which is almost two thousand USD. In other words, we add 5% of resources but we can get 14%~30% improvement in performance. This is achieved even though our tests are not running on a perfect supercomputing environment such as connected by the InfiniBand. The two nodes are connected by a normal router. Thus, if we deploy this HG system in a real supercomputing environment, the communication time can be trivial. So the HG heterogeneous system is able to provide even better percentage improvement.

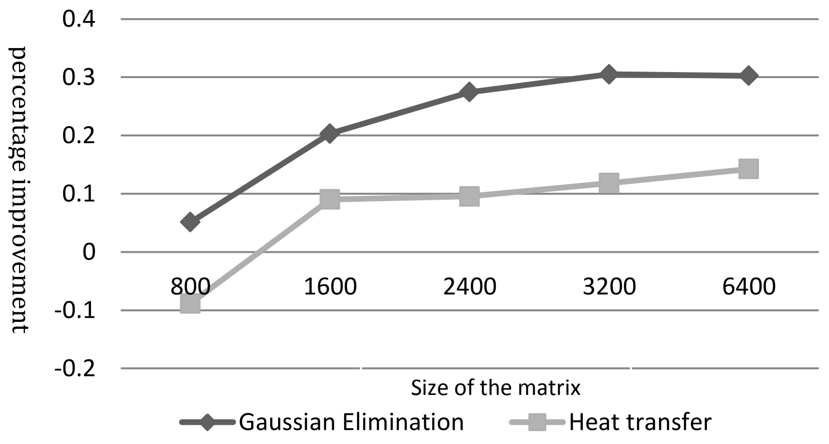


Figure 6-10. The percentage of improvement of heterogeneous to homogenous GPUs

6.6. Results of heat transfer application

In this experiment, we use an MPI based program running on the CPU called Cmpi. In figure 6-11, the left y-axis captures time in second and the right y-axis is measured by million updates of computing units per second which is calculated as the workload size divided by the sum of Comm. and Comp. time. The x- axis denotes workload sizes of different systems, Cmpi and HG. For example, 100k-Cmpi means the workload size is 100 thousand computing units running on Cmpi system which is the CPUs based program. 1M-HG means the workload size is one million computing units running on HG system which is based on 4 GPUs, namely, 2 GTX650s and 2 Tesla C2050s. An update of computing units means a computing unit calculates the value at time *t+1* at time *t*.

In Figure 6-11, HG outperforms Cmpi system because the HG system speeds less time in initiation and computation. HG initializes and computes computing units on GPU side which dramatically increases performance. However HG system needs more time in the communication step because HG needs to copy values from GPU to CPU and then copy them back. This disadvantage does not affect the performance significantly.

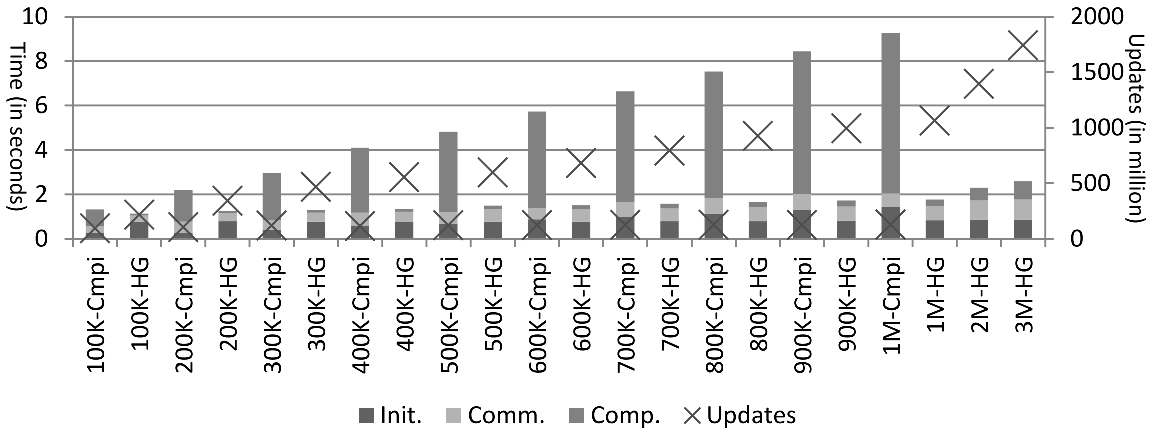


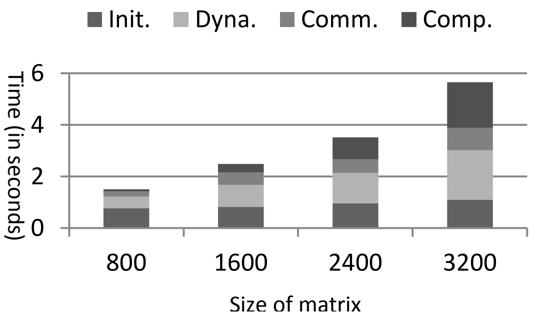
Figure 6-11. 2-D heat transfer in HG and CPU based system

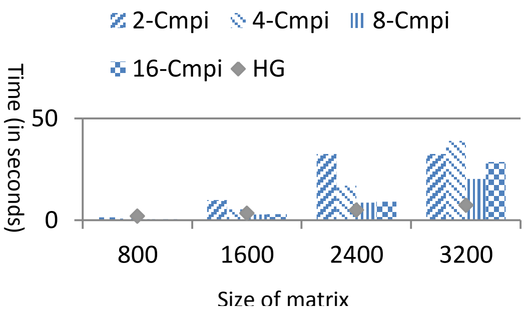
The time for updates of computing units includes the time of communication and computation steps. Hence the performance of the system is measured more accurately as million updates of computing units per second which is marked as × in figure 6-11. When the workload size is 1 million, the updates of Cmpi is around 127 million comparing to 1064 million of HG. Thus, HG system is more than 8 time faster than Cmpi. When the workload size is larger than 1.5 million Cmpi cannot run. This is because Cmpi uses a 3 dimensional array in the main function. The system cannot allocate enough memory in stack for such a requirement. However even at these high loads, HG can run smoothly because HG uses dynamic memory allocation.

6.7. Result of Gaussian Elimination application

Gaussian Elimination application is different from the heat transfer application. We add a dynamic step in the Gaussian elimination application where the system can decide the pivot. The control system is implemented by different number of MPI processes running on CPUs.

Figure 6-12 (a) shows the execution time of each step of the Gaussian Elimination application. The time consuming part is the dynamic step because it gets the values of the current row from device and collects all of them to select the pivot. This is then broadcast to all workers. Finally each worker updates the pivot row on device. So the HG system in Gaussian Elimination application includes four data transfers between device and host and four data transfers between workers. Even with these complicated time consuming steps in the HG system, it still outperforms the MPI system. For these two computing nodes, 8 MPI processes get the best performance of the MPI system. However the HG system gets 2.75 times speed up.





(a) 2-D Gaussian Elimination application (b) 2-D Gaussian Elimination application and MPI processes

Figure 6-12. 2-D Gaussian Elimination in HG and CPU based system

6.8. Conclusion

In this chapter, the experimental results show these optimizations improve the performance of the HG system. For example, the NPHP algorithm minimizes the inter-processor communications. The FPMC algorithm ensures the load balance of heterogeneous GPUs. The data reuse technology significantly reduces the computation time. The proposed heterogeneous GPUs system outperforms the MPI system based on CPU and the homogeneous GPUs system. The price for a GTX650 GPU card is around 100 USD and the price for a Tesla T2050 GPU card is almost 2,000 USD. The ratio of the price of a GTX650 and a Tesla T2050 is 5%. From the results of heat transfer and Gaussian Elimination, we add 5% resource measured by money spent which means we add two GTX650 GPU cards, but we can get 14%~30% performance improvement.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

7.1 Conclusion

The dependent tasks problem in a grid running on heterogeneous processors including CPU and GPU faces some challenges which include load balancing and reducing communication cost. The goal of load balancing is to assign the workload sizes to be proportional of the speeds of processors. So it is vital to estimate the speeds of processors accurately. However it is very hard to use a positive number to estimate the speed of a processor, especially when considering communication cost. One feasible approach is to use a function of absolute speed of a processor against the workload size and communication cost to estimate their speeds accurately. Existing function approaches use experimental measurements to build the speed function or table for each processor. We have proposed in this thesis a new approach called FMPC that utilizes NPHP to minimize communication cost. FMPC aims to ensure load balancing of heterogeneous processors and minimize their communication cost. Our experiments show that NPHP reduces communication cost by at least 10% and FMPC improves the load balance by 10% on average.

We have proposed in this thesis a novel approach for processing dependent tasks on heterogeneous GPUs system based on neighborhood grid points. Previous works have investigated only process in dependent tasks on homogeneous GPUs. Our approach supports change of neighborhood when the system is running. Hence the system can support more types of applications. Also we implement some optimizations to improve its performance, such as NPHP reduces at least 10% communication cost, FMPC improves the load balance by 10% average, and the data reuse technology in computing kernel to utilize shared memory to reduce the global memory accesses, brings 7 times speedup.

In chapter 1, we define the computation scenario: the dependent tasks on grid problems and we do a survey of how to utilize heterogeneous processors to process the dependent task. Also, we give the reason for using use general purpose graphic processing units to deal with computing intensive problems. In chapter 2, we list two ways to partition data on heterogeneous processors: constant performance model and functional performance model. Secondly, we briefly talk about how to program in CUDA and the memory architecture of GPU. Finally, we introduce a model to process dependent tasks in GPU. In chapter 3, we propose a novel partial homogenous partitioning algorithm (NPHP) to reduce the inter-processor communication cost. In chapter 4, we propose a partitioning algorithm of functional performance model that considers communication cost to ensure load balance. In chapter 5, we give the architecture of the heterogeneous GPUs system (HG) and analyze the system. In chapter 6, we run two types of applications on the system: heat transfer and Gaussian Elimination. The heat transfer is a static neighbor application and the Gaussian Elimination is dynamic neighbor application. For the static neighbor application, HG is at least 8 times faster than MPI program running on CPU. For the dynamic neighbor application, HG can get 2.75 times speedup.

7.2 Contributions

This work addresses the main challenges in executing scientific applications on GPUs:

1. A novel approach to reducing internet communication between GPUs based on grid sharp (chapter 3),
2. A new technique to achieve load balancing to maximize utilization of heterogeneous GPUs resources based on the partitioning algorithm of functional performance model that considers inter-processor communication cost (chapter 4),
3. The implementation of the data reuse technology to improving CUDA device memory accesses (chapter 5.6).
4. The design and implementation of a general platform to support fixed and dynamic task dependent applications (chapter 5).

In this contribution we propose a novel platform, Heterogeneous GPUs (HG) that leverages heterogeneous GPU resources to process task dependent applications, such as heat transfer and Gaussian elimination. This platform utilizes heterogeneous GPUs power and partitions jobs according to their capabilities. In this work we have implemented and provide a set of APIs. Researchers can use these APIs to implement their own task dependent applications.

7.3 Future Work

For future work, one area that requires further study is using InfiniBand to reduce the bottleneck in the distributed GPUs system. By improving inter-processors communication, the performance of the system will be improved as well. A second area for future work is to propose an adaptive method for efficiently building the performance benchmark tables for GPUs. This method can be used to construct performance benchmark tables for GPUs. Finally, a third area of investigation is the use of distributed GPUs system to process big data, especially GPUs integrated into HBase which is NoSQL database based on hadoop.

7.3.1 Using InfiniBand to improve performance

In the section 5.3, we analyzed heterogeneous GPUs system and found that the performance bottleneck is in the data transferred between GPUs. The inter-GPUs communication requires two transfers by bus plus a transfer by network. First, the data in the source GPU memory is transferred into local main memory by bus. Then, it is transferred from the source node to the target node through the network. Finally, the target node receives the data and copies it into target GPU memory. Form the results of Gaussian Emulation application, the communication time takes up more than 50% of the time. In this work, we use the PCI Express 2.0 and a 100 Mbps network which is a normal configuration for a commercial computer. J. Huang etc. (Huang et al. 2012) used the InfiniBand to connect all HBase region servers and they show that this can improve the throughput more than 3 times. Hence, if we use a much faster bus such as PCI Express 3.0 which speed is 2 times than the speed of PCI Express 2.0, and use InfiniBand to connect all GPUs, we can improve performance greatly.

7.3.2 Constructing benchmark tables of GPUs

From the experiment results in section 6.5, we observed that if the performance of GPU is estimated more accurately, the system gets better load balancing. The FPMC technique proposed in this work brings better quality of problem partitioning for high performance computing on common, heterogeneous networks than the previous work (FPM) (Alexey Lastovetsky and Ravi Reddy 2010), because FPMC considers communication cost. So the performance functions of GPUs in FPMC are not only related to work load sizes but also to communication cost. The performance benchmark tables of GPUs in FPMC are surfaces as shown in Figure 4-3. In short, in the dependent tasks environment, to get good quality of problem partitioning for dependent tasks, it is critical to accurately build performance benchmark tables of GPUs. Our goal is to build performance benchmark tables of majority GPUs and for different types of applications.

However it is very expensive to build the performance table. Firstly, there are many parallel computing applications. It’s impossible to build performance benchmark tables for all of them. Secondly, even for a specific application the experimental time is very large. Because we need to run the large range of work sizes and for each work size we need to run a large range of communication costs. We will use optimizations to efficiently build the performance table for GPUs and for different applications. For example, the expected maximal workload size of a GPU (w_{gpu}) is 100 million grid points and the interval for the workload size (I_w) may be set as 1000 grid points. If the grid is a 1 million by 1 million grid points matrix, the maximal communication for a GPU (C_{gpu}) is up to the perimeter of the matrix which is 4 million and the interval for the communication (I_c) is set as 100. The experimental time is 4 billion and is calculated by equation 7-1.

$$\text{Experimental Time} = \frac{w_{gpu}}{I_w} \times \frac{C_{gpu}}{I_c} \tag{7-1}$$

In future, we plan to use the following methods to minimize experimental times.

- Classify applications by the number of memory accesses and arithmetic operations.

The operations of kernel computation in a GPU thread usually constitutes of memory access and arithmetic operations. The speeds are different, and hence the number of memory accesses and arithmetic operations dominate the execution time. Some classic applications which have different numbers of memory accesses and arithmetic operations can be used to build the benchmark performance tables. For a new application, we just need to find the benchmark performance table of a classic application which has the same or approximately the same numbers of memory accesses and arithmetic operations.

- Use adaptive method to select the intervals of work load sizes and communication cost.

If small intervals are selected, the accuracy is improved but the experimental time increases. However, if large intervals are selected, the accuracy is decreased but the experimental time is reduced. So the crucial step to minimize experimental times is determining how to set intervals. Alexey et cl (Alexey Lastovetsky et al. 2006) proposed a procedure for building a piecewise linear function approximation of the speed band of a processor. Their work can be extended using an adaptive method to choose the appropriate intervals instead of fixed intervals as shown in Figure 7-1. *S(x)* is the GPU performance function; x is the workload size and y is the absolute speed of the GPU. First we set the workload interval I_w to d and select a threshold h. We run the application with workload size x_1 and x_2 respectively and we get the absolute speeds, y_1 and y_2. If $l = \|y_1 - y_2\| > h$, then $I_w = \frac{I_w}{2}, x_3 = x_2 - I_w$; otherwise $I_w = d$, $x_3 = x_2 + I_w$. Hence, we get the next test workload size x_3 and run the application to get the absolute speed y_3. This process keeps going until the expected maximal workload size is reached. In the same way, the benchmark performance tables of communication cost can be constructed.

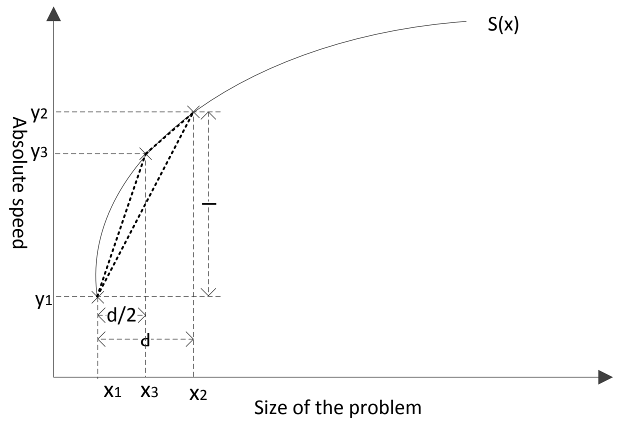


Figure 7-1. Adaptive method to build the performance benchmark table of a GPU

7.3.3 *Using distributed GPUs to process big data (NoSQL database, HBase)*

It is essential to parallel process the queries of NoSQL databases storing big data. GPUs have the essential advantages to process small tasks in parallel. Hence utilization of GPU in NoSQL database is attracting a lot of interest in both academia and industry. IBM filed a patent about a GPU-Accelerated Database (Child 2012) which can execute a parallelized query on a GPU kernel executable or process the particular stored procedure on one or more GPU devices. B. He et al. designed and implemented and in-memory relational query processing system on GPU (He et al. 2009). The result shows the performance using GPU is 2 to 7 times faster than CPU. However it needs to load all the data to GPU memory first. W. Fang et al. proposed a way to compress the data in a GPU database (W. Fang, He, and Q. Luo 2010). It is still impossible to use their GPU database to process big data because of the limitation of GPU memory. Using GPU, the feasible ways to improve the throughout and response time of NoSQL, for example HBase, include: a) Sort row_key in GPU; b) Split a query into several queries and use GPU to compute index or hash values.

REFERENCES

Apache. 2013. “hadoop.” Retrieved March 24, 2013 (http://hadoop.apache.org/).

Asanovic, Krste, Bryan Christopher Catanzaro, David A Patterson, and Katherine A Yelick. 2006. “The Landscape of Parallel Computing Research : A View from Berkeley.” *EECS Department University of California Berkeley Tech Rep UCBEECS2006183* 18(UCB/EECS-2006-183):19. Retrieved (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.8705&amp;rep=rep1&amp;type=pdf).

Asanovic, Krste, John D Kubiatowicz, Edward A Lee, David Wessel, and Katherine A Yelick. 2008. “The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View.” *Parallel Computing* 52(UCB/EECS-2008-23):23. Retrieved (http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-23.html).

Barney, Blaise. 2012. “Modeling Heat Transfer in Parallel.” Retrieved March 25, 2012 (http://www.cas.usf.edu/~cconnor/parallel/2dheat/2dheat.html).

Beaumont, O, V Boudet, F Rastello, and Y Robert. 2000. *Matrix-matrix multiplication on heterogeneous platforms*. IEEE Comput. Soc Retrieved (http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=963416).

Belkhale, Krishna, and Prithviraj Banerjee. 1990. “recursive partitions on multiprocessors.” Pp. 930–938 in *Processdings of the Fifth Distributed Memory Computing Confereence*.

Berger, M, and P Colella. 1989. “Local adaptive mesh refinement for shock hydrodynamics.” *Journal of Computational Physics* 82(1):64–84. Retrieved (http://linkinghub.elsevier.com/retrieve/pii/0021999189900351).

Berger, Marsha J, and Shahid H Bokhari. 1987. “A partitioning strategy for nonuniform problems on multiprocessors.” *IEEE Transactions on Computers* C-36(5):570–580. Retrieved (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1676942).

Bowen, N S, C N Nikolaou, and A Ghafoor. 1992. *On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems*.

Child, Timothy. 2012. “Database acceleration using gpu and multicore cpu systems and methods.” United States patent US 20120259843

Cowboy. 2013. “Cowboy.” Retrieved February 11, 2013 (http://hpcwiki.it.okstate.edu/index.php/Cowboy).

Crandall, P E, and M J Quinn. 1993. “Block data decomposition for data-parallel programming on a heterogeneous workstation network.” in *1993 Proceedings The 2nd International Symposium on High Performance Distributed Computing*.

Crandall, P E, and M J Quinn. 1994. *Block data decomposition for partial-homogeneous parallel networks*. IEEE Comput. Soc. Press.

Crandall, Phyllis E., and Michael J. Quinn. 1993. *Problem Decomposition in Parallel Networks*. Corvallis,OR.

Dovolnov, E, A Kalinov, and S Klimov. 2003. “Natural block data decomposition for heterogeneous clusters.” in *Proceedings International Parallel and Distributed Processing Symposium*.

Fang, Wenbin, Bingsheng He, and Qiong Luo. 2010. “Database compression on graphics processors.” *Proceedings of the VLDB Endowment* 3(1-2):670–680. Retrieved (http://portal.acm.org/citation.cfm?id=1920927).

Farber, R. 2011. *CUDA Application Design and Development*.

Farivar, Reza, Abhishek Verma, Ellick M. Chan, and Roy H. Campbell. 2009. “MITHRA: Multiple data independent tasks on a heterogeneous resource architecture.” Pp. 1–10 in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE Retrieved November 1, 2012 (http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5289201).

Forum, Message Passing Interface. 2009. “MPI: A Message-Passing Interface Standard, Version 2.2.” *International Journal of Supercomputer Applications*. Retrieved (http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf).

Govindaraju, Naga K, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. “GPUTeraSort: high performance graphics co-processor sorting for large database management.” *Memory* 325–336. Retrieved (http://portal.acm.org/citation.cfm?id=1142511).

He, Bingsheng et al. 2009. “Relational query coprocessing on graphics processors.” *ACM Transactions on Database Systems* 34(4):1–39. Retrieved (http://portal.acm.org/citation.cfm?doid=1620585.1620588).

He, Bingsheng, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. “Mars: a MapReduce framework on graphics processors.” P. 260 in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT ’08*. New York, New York, USA: ACM Press Retrieved November 1, 2012 (http://portal.acm.org/citation.cfm?doid=1454115.1454152).

Hong, Chuntao, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. 2010. "MapCG : Writing Parallel Program Portable between CPU and GPU." Pp. 217–226 in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques PACT 10*. ACM Retrieved (http://doi.acm.org/10.1145/1854273.1854303).

Huang, J et al. 2012. "High-performance design of hbase with RDMA over InfiniBand." Pp. 774–785 in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium IPDPS 2012*. Retrieved (http://www.scopus.com/inward/record.url?eid=2-s2.0-84866848525&partnerID=40&md5=4ac06f5112e4318131ebac67bb24696e).

Kalinov, Alexey, and Alexey Lastovetsky. 2001. "Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers." *Journal of Parallel and Distributed Computing* 61(4):520–535. Retrieved (http://linkinghub.elsevier.com/retrieve/pii/S0743731500916861).

Khronos. 2012. "OpenCL." Retrieved (http://www.khronos.org/opencl/).

Kirk, David B, and Wen-mei W Hwu. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Retrieved (http://www.amazon.co.uk/Programming-Massively-Parallel-Processors-Hands-/dp/0123814723).

Lastovetsky, A, and R Reddy. 2004. "Data partitioning with a realistic performance model of networks of heterogeneous computers." in *18th International Parallel and Distributed Processing Symposium 2004 Proceedings*.

Lastovetsky, Alexey, and Ravi Reddy. 2007. "Data Partitioning with a Functional Performance Model of Heterogeneous Processor." *International Journal of High Performance Computing Applications* 21(1):76–90.

Lastovetsky, Alexey, and Ravi Reddy. 2010. "Distributed Data Partitioning for Heterogeneous Processors Based on Partial Estimation of Their Functional Performance Models." Pp. 91–101 in *EuroParHeteroPar 2009*, vol. 6043, edited by Hai-Xiang Lin et al. Springer Retrieved (http://dx.doi.org/10.1007/978-3-642-14122-5_13).

Lastovetsky, Alexey, Ravi Reddy, and Robert Higgins. 2006. "Building the Functional Performance Model of a Processor." Pp. 746–753 in *21st Annual ACM Symposium on Applied Computing*. New York, New York, USA: ACM Press.

M. Ekman, F. Warg, J. Nilsson. 2005. "An in-depth look at computer performance growth." *ACM SIGARCH Compuer Architecture News* 33:144–147.

Maruyama, Naoya, Kento Sato, Tatsuo Nomura, and Satoshi Matsuoka. 2011. *Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers*. IEEE Retrieved (http://dl.acm.org/citation.cfm?id=2063384.2063398).

Nedeljkovic, N, and M J Quinn. 1992. *Data-parallel programming on a network of heterogeneous workstations*. edited by Geoffrey C Fox And Luc Moreau. John Wiley & Sons, Ltd.

Nicol, D.M. 1994. “Rectilinear partitioning of irregular data parallel computations.” *Journal of Parallel and Distributed Computing* 23(2):119–134.

Nvidia. 2012a. “CUDA.” Retrieved March 24, 2012 (http://www.nvidia.com/object/cuda_home_new.html).

Nvidia. 2012b. “NVIDIA’s Next Generation CUDA™ Compute Architecture:Fermi™.” Retrieved (http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf).

Nvidia. 2012c. “TESLA™ C2050 / C2070 GPU ComPUTinG ProCESSor.” Retrieved (http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf).

Nvidia. 2013a. “CUDA Community Showcase.” Retrieved (CUDA Community Showcase).

Nvidia. 2013b. “Developer Zone.” Retrieved June 24, 2013 (https://developer.nvidia.com/cuda-action-research-apps).

Nvidia. 2013c. “What is CUDA.” Retrieved (https://developer.nvidia.com/what-cuda).

Nvidia, C. 2011. “NVIDIA CUDA C Programming Guide.” *Changes* (350):173. Retrieved (http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf).

Parashar, M. 2012. “GrACE.” Retrieved March 25, 2012 (http://nsfcac.rutgers.edu/TASSL/Projects/GrACE/).

Sanders, Jason, and Edward Kandrot. 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Retrieved (http://wwwzb.fz-juelich.de/contentenrichment/inhaltsverzeichnisse/2010/9780131387683.pdf).

Xue, M., K. K. Droegemeier, and V. Wong. 2000. “The Advanced Regional Prediction System (ARPS) - A multi-scale nonhydrostatic atmospheric simulation and prediction model. Part I: Model dynamics and verification.” *Meteorology and Atmospheric Physics* 75(3-4):161–193. Retrieved November 1, 2012 (http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/s007030070003).

Yang, Rui, and Johnson Thomas. 2012. “Processing dependent tasks on a Heterogeneous GPU resource architecture.” Pp. 627 –632 in *Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on*.

VITA

Rui Yang

Candidate for the Degree of

Doctor of Philosophy

Thesis: PROCESSING DEPENDENT TASKS ON A HETEROGENEOUS GPU RESOURCE ARCHITECTURE

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Doctor of Philosophy in Computer Science at Oklahoma State University, Stillwater, Oklahoma in July, 2013.

Completed the requirements for the Master of Science in Information Security at University of Electronic science and Technology of China, Chengdu, China in 2009.

Completed the requirements for the Bachelor of Science in Computer Science at University of Electronic science and Technology of China, Chengdu, China in 2005.