A Scheduling Framework for a Heterogeneous Parallel Architecture

Wei Zhang

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

November 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Wei Zhang

Entitled: A Scheduling Framework for a Heterogeneous Parallel Architecture

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____Chair
Dr. E. J. Doedel

_____ Examiner
Dr. T. Eavis

_____ Examiner
Dr. S. P. Mudur

_____ Supervisor
Dr. D. Goswami

Approved by _____
Chair of Department or Graduate Program Director

_____20_____ _____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

A Scheduling Framework for a Heterogeneous Parallel Architecture

Wei Zhang

Scheduling on heterogeneous parallel and distributed computing environment has been studied for decades. Based on different assumptions, researchers have proposed several algorithms and heuristics aiming to improve the performance of parallel applications. Most of these works focus on clusters of CPUs or grid-based environments where heterogeneity is created by processors and networks of varying speeds. However, in recent years, there has been wide spread use of another type of heterogeneous parallel computing environment, even on regular PCs and workstations, which comprise of multi-core CPUs and many-core GPGPUs (General Purpose Graphic Processor Units). Heterogeneity in this new generation of computers is even more pronounced due to the significant differences in architectures and programming models between CPUs and GPGPUs. The scheduling problem on a heterogeneous environment is known to be NP-Complete. Consequently, this research proposes several approximate strategies to solve this problem on a heterogeneous CPU-GPGPU environment. As a focus of this research, the strategies utilize the structural and behavioral characteristics of patterns in parallel programming to facilitate scheduling decisions. The parallel pattern extensively studied in this research is the farm pattern, which is used in a wide range of parallel applications. For the purposes of scheduling, the farm pattern is further classified into several categories and subsequently scheduling strategies for each of these categories are proposed. The similar strategies can be employed for the scheduling of some other

patterns, e.g., data flow and pipeline. Since characteristics of the patterns and the features of the CPU-GPGPUs environment are both considered in making scheduling decisions, the proposed strategies are found to deliver better performances as compared to other contemporary strategies.

A scheduling framework has been designed and implemented based on these strategies for the classified farm patterns. The framework not only intends to hide the complexity of parallel programming but also can automatically schedule tasks and balance loads among processors, relieving these burdens from the programmer. In addition, the framework serves as a test bed for newer scheduling heuristics on the target heterogeneous system, and also as an experimental verifier of the proposed hypothesis that use of patterns can facilitate in making better scheduling decisions.

Acknowledgments

This work would not have been possible without the technical and moral support of my supervisor Dr. D. Goswami. He opened a door for me in the research field and his enthusiastic supervision has been a continuous force driving my research in the right direction. I am so fortunate that I had his valuable guidance.

I am grateful to my parents for their both financial and moral support. I am very thankful to them for all they have given me in my life, for sharing ups and downs, and for reminding me, when necessary.

I am also thankful to Dr. Zunce Wei, who gave me many useful suggestions when I completed my thesis.

Finally, I am very thankful to everyone who has directly and indirectly helped me, in any way, to accomplish this research.

Table of Contents

List of Figures

List of Terms

GPGPU	General Purpose Computing on Graphics Processing Unit
CUDA	Compute Unified Device Architecture
TBB	Intel's Thread Building Blocks
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instructions Multiple Data
DLS	Divisible Load Scheduling
SP	Streaming Processor
SM	Streaming Multi-processor
HASS	Heterogeneous Architecture Scheduling Strategies
LTF	Largest Task First
STF	Shortest Task First
PD	Parallelism Degree
SPD	Simplex Parallelism Degree
MPD	Multiplex Parallelism Degree

Chapter 1 Introduction

Parallel computing has been widely explored for decades for solving problems fast and efficiently. Currently it has been widely employed in a variety of fields, from physics to biology, as well as from climatology to geology, to name a few.

With the advent of commodity parallel computing platforms like clusters, general purpose graphics processor units (GPGPUs) with hundreds of cores, and multi-core CPUs, parallel programming is becoming even more widespread and commonplace using regular PCs and workstations. However, compared with sequential programming, parallel programming has several additional issues.

Some of these issues are that parallel application development brings in some additional burdens such as effective partitioning of a given problem into sub-problems (i.e., tasks) that can be executed by processes/threads, process and thread creation and management, mapping processes to processors, performing efficient communication between different processes, scheduling and load balancing, etc. To address these issues, hundreds of parallel programming models and tools have been developed over the years. Some of the them targeted for current commodity platforms include PVM [1] and MPI [2] for clusters and network of workstations, TBB [3] and CUDA [4] for multi-core CPU and massive-core GPGPU [5] respectively, and OpenMP[6] for multiprocessors and multi-core CPUs.

One extremely important issue in parallel program execution is to obtain a high efficiency and a minimized *makespan*, i.e., the time interval between the start of the first process and the end of the last process. Efficiency will become high if any processor

idling due to load-imbalance or communication overhead is minimized. In order to achieve low makespan and high efficiency, it is very important to have proper mapping and scheduling of tasks to processor. The issues of scheduling and load-balancing during parallel program execution on a heterogeneous CPU-GPGPU environment are the focuses of this research.

Scheduling in the areas of parallel and distributed processing has been studied for decades, and several scheduling algorithms have been proposed by researchers on different parallel platforms. A basic goal of task scheduling is to map tasks to processors in such a way that the idling time of each processor is minimized, and all processors complete their computation at almost the same time. In a heterogeneous parallel computing platform with varying processor speeds and varying interconnection bandwidths, scheduling is a challenging problem and is known to be NP-complete [7][8]. Hence, many times precise scheduling may not be possible to be obtained in a reasonable computational cost, and hence heuristics need to be designed.

The issue of scheduling in the area of parallel programming can be application developer transparent, semi-transparent or fully developer assisted. In the first category, the parallel programming model or the development tool hides the details of task scheduling from the application developer. In the second category, the developer provides some suggestions to the underlying system. In the third category, the developer is responsible for everything. For example, in MPI and socket-based programming, the developer has to handle many of the issues of scheduling. TBB, on the other hand, has a scheduler that uses work stealing [10] strategy to balance loads among cores in a CPU. CUDA's scheduler uses a greedy scheduling strategy [11] to distribute and balance loads among

cores in a GPU. In OpenMP, a set of scheduling methods such as static, dynamic and guided [6] are provided as options for users; thus it will fall in the semi-transparent category.

1.1 Objectives of This Research

This research focuses on transparent to semi-transparent scheduling on a heterogeneous parallel computing environment comprising of multi-core CPU and many-core GPGPUs. Though scheduling on heterogeneous systems has been extensively researched in the past, many of those approaches deal with heterogeneity due to varying processor speeds and varying network bandwidths. In contrast, the heterogeneous system that we are dealing with has processors of not only varying speeds but also of different architectures (e.g., MIMD in CPU cores versus hybrid model in GPU cores), each supporting a different programming model (e.g., task-parallel and data-parallel are both equally supported in CPUs, while data-parallel are more likely to have a better performance than task-parallel in GPUs). Moreover, with increasingly fast interconnection links between CPU and GPUs with emerging technologies like PCI bus and its extensions [12], bandwidth is less a concern than in the previous works. All these issues render the previous heuristics for heterogeneous systems not to be directly applicable to a CPU-GPGPU system. All these issues are further elaborated in the following chapters of the thesis.

This research is on scheduling of parallel applications on a heterogeneous CPU-GPGPU system and it uses patterns in parallel programming to achieve its objectives. The research is based on the hypothesis, as proposed by us, that different patterns in parallel computing need different scheduling criteria. For example, scheduling strategies of a

farm pattern consisting of independent tasks could be different from a task-graph consisting of some forms of dependencies (i.e., control and/or data). Consequently, if the pattern(s) of an application is known in advance then better scheduling decisions can be applied as compared to ad hoc strategies that are usually employed. Some of the commonly used patterns in parallel programming are: task and data farm, master-worker, pipeline, divide and conquer, systolic array, to name a few [13]. The research makes further classifications of some of these patterns based on scheduling needs. New heuristics are developed which used some of the generic characteristics of the patterns. Finally, a scheduling framework is developed based on some of these patterns with the objectives of achieving load-balanced execution of parallel programs with reduced makespan, and also with an objective of verifying our proposed hypothesis.

Traditionally, design pattern concepts have been used towards the design and development of parallel programs. The term *skeleton* has been conventionally used to represent behavioral abstractions of patterns as pre-implemented code. The term *algorithmic skeleton* [15] has been traditionally used to represent algorithmic abstractions of patterns, which are best represented in functional forms and best implemented by functional programming languages. On the other hand, the term *architectural skeleton* [16] has been used to represent architectural abstractions of patterns. Architectural skeletons can be thought of as building blocks for parallel virtual machine on which an application developer builds the application. Architectural skeletons are behavior free and have been implemented using an object-oriented paradigm. The common objective of these skeleton-based approaches is to aid the programmer in the design and development phases of parallel programs, purely from the application and its algorithms perspective.

There are several skeleton-based approaches to parallel program design and development that can be found in the literature [14][15][16][17][18].

The aforementioned skeleton-based approaches to parallel programming concentrate solely on the algorithm-specific aspects of a parallel program, and do not employ patterns for other low-level system-specific uses such as scheduling, load-balancing, and fault-tolerance. This research is a step towards the application of patterns to scheduling and load-balancing of parallel programs. We use the term *parallel systems skeletons* to represent those skeletons (i.e. pre-implemented building blocks) that assist in patterns-specific scheduling and load balancing. The scheduling framework, by definition, is a collection of such skeletons.

In the literature, very limited amount of works can be found that uses patterns for the purposes of scheduling in the CPU-GPGPU heterogeneous system consisting of one CPU and multiple GPUs which are connected through the bus on motherboard. These works deal with heterogeneity in a cluster or grid environment arising from processors of varying speeds and interconnection links of varying bandwidths. In contrast, as discussed before, the requirements in this research are quite different due to a different type of heterogeneity.

The type of heterogeneous system of our interest, a commodity platform made of CPU and GPGPUs, is gaining commonplace in high-performance parallel computing due to its price-performance benefits. Multi-core CPUs, with two or more cores, are becoming commonplace, providing a (shared-memory) multiprocessor environment with regular PCs and workstations. Combined with GPGPUs, which are composed of hundreds of

cores designed for general purpose computing, it has provided a powerful parallel programming environment at minimal cost. GPU cores are organized into blocks, with each block providing a SIMD [21] style of architecture that supports data-parallelism [21]. On the other hand, CPU cores support MIMD [21] style of architecture that supports task-parallelism [21]. CPU and GPGPUs are connected in the motherboard via special high-bandwidth bus (e.g., PCI bus or its extensions [10]), which tries to overcome data transfer overhead. The heterogeneous architecture and its programming environment are further discussed in a following chapter.

1.2 Contributions of This Research

In this research, we propose the hypothesis that different patterns in parallel programming need different scheduling strategies for near optimal performance. We first investigate our hypothesis for the farm pattern. To achieve this goal, we need to redefine the farm pattern into two parts: *task (function) farm* and *data (load) farm*. We further sub-classify the task and data farm patterns for our purposes of scheduling. Subsequently, we discuss how the scheduling of the farm pattern can be applied in scheduling a *pipeline* and more generally a *data-flow* pattern.

As discussed before, this research focuses on a new type of heterogeneous parallel system, consisting of CPUs and GPGPUs, which is gaining immense interest these days as a commodity platform of choice for parallel programming due to its high performance-price benefits. Unlike the previous scheduling strategies developed for heterogeneous systems, these strategies are not straightway applicable to a CPU-GPGPU system due to differences in system constraints and programming models. As a result,

new scheduling heuristics have been developed for the farm pattern and its variants. To note that some of these new heuristics developed make use some of the previous heuristic(s).

Finally, a pattern-based scheduling framework has been proposed. The framework can serve as a test bed for newer scheduling heuristics on the target heterogeneous system, and also serve as an experimental verifier of the proposed hypothesis. These contributions are further elaborated in a following chapter.

1.3 Organization of the Thesis

The next chapter discusses some backgrounds and related works: it first discusses the characteristics of the CPU-GPUs architecture; followed by a discussion on some of the skeleton based approaches to scheduling and load-balancing. The details of conventional farm pattern, which is one of the principal focuses of this thesis, are presented and different approaches based on the parallel patterns are further elaborated. Some existing scheduling strategies relevant to this research are also revisited in this chapter. Chapter 3 introduces a further classification of farm pattern for the purpose of scheduling and some of the scheduling strategies of these classified patterns on the CPU-GPGPUs architecture are presented. It also discusses how the scheduling of the data-flow graph pattern can be mapped to the scheduling of the farm pattern. Chapter 4 discusses the experimental performances of the previous strategies and compares the results with some existing approaches. Chapter 5 discusses an implementation of the scheduling framework and illustrates its use in order to develop parallel applications on CPU-GPGPUs architecture.

Finally, chapter 6 concludes the thesis with a discussion on the existing problems and other issues that need to be resolved in the future.

Chapter 2 Background

2.1 GPGPU and CPU-GPUs Architectures

2.1.1 General Purpose GPU

GPU (Graphic Processing Unit) is a special microprocessor initially designed to process graphics applications and then output them to monitors in an efficient and rapid manner. Since graphics is comprised of independent vertices and fragments that can be processed in parallel, GPUs are naturally designed as massive-core processors for the purposes of manipulating these elements using stream processing. Here a stream is a set of data that require the same operations and stream processing is akin to SIMD (Single Instruction, Multiple Data) in parallel computing. The number of cores in modern GPUs can range from dozens to hundreds. An example of the architecture of nVidia's GPGPU is shown in Figure 1. In this architecture, SP (stream processor) is the processing unit performing computation. SM (stream multiprocessor) is a set of SPs that share a local memory and all the SPs in one SM perform operations in a SIMD fashion. One GPU contains multiple SMs. MT and IU are non-related components for general-purpose computation.

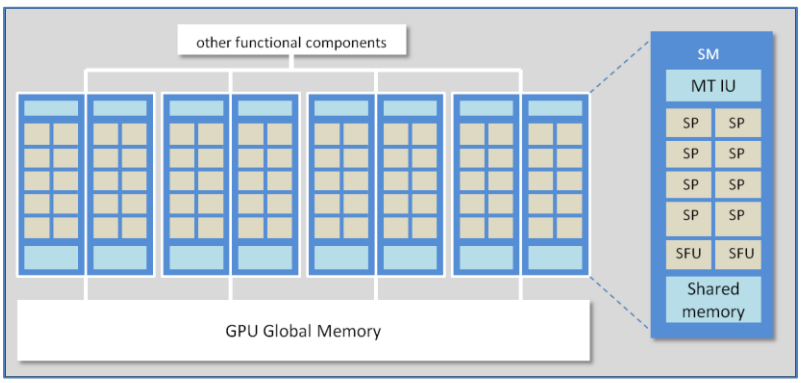

Figure 1: The architecture of nVidia's Tesla GPGPU

Recently, GPGPU (General-Purpose Graphics Processing Unit), which is introduced as a technique to leverage the excellent parallel computing power of GPUs to accelerate the solving of some general problems, has been studied by many researchers. With the help of certain auxiliary development tools such as CUDA (Compute Unified Device Architecture) from nVidia, programming on GPUs to perform general-purpose computation is becoming feasible.

Nearly all GPGPUs are present on video cards (also known as video adapter, graphics accelerator card, display adapter, or graphics card) which are normally connected to a CPU via the bus on motherboard of a PC or workstation. Another component that resides on a video card is video memory (also known as graphics memory). This memory is dedicated for GPU computing, which means that it is only accessible directly by GPUs; likewise, a GPGPU cannot access the data residing on the main memory of a PC or workstation directly. The data required by GPUs' computation must be copied and transferred from the main memory to the video memory and the results also have to be transferred back.

Nowadays, the GPGPU market is dominated by two manufacturers: nVidia and AMD. Since nVidia first released a series of GPUs supporting general purpose computing and provided corresponding programming tools and SDKs to users, its GPGPUs are currently more popular and are installed more widely. Therefore, in this thesis we will take nVidia's GPGPU and programming tools as our subjects of study.

2.1.2 The Characteristics of GPGPU Computing

(1) Since a GPU is traditionally connected to a CPU via the bus on the motherboard, the bandwidth between the GPU and the CPU is quite large, normally from several gigabytes per second to over ten gigabyte per second. With newer interconnection technologies like PCI bus and its extensions [10], this bandwidth is even getting higher. Therefore, the latency caused by data transferring between CPU and GPU is far less than that of transferring data over the network of general clusters or grids. In the network of a cluster or a grid, as the latency of data transfer is high, the overhead of *message start-up* (the process of packing the message as low-level network packets that can be transferred through the physical network and initialization cost of the data transfer environment) is often small enough to be neglected. However, in data transferring between a GPU and a CPU, since the data transfer time is reduced significantly due to the large bandwidth while the message start-up cost remains the same, the message start-up cost cannot be neglected any more. Therefore, the message start-up cost can even dominate the total overhead of passing a message between a CPU and a GPU when the message size is fairly small. To address such issues, we apply task bundling in our strategies to minimize the message start-up cost.

(2) Nowadays, the high-end GPUs for parallel computing have hundreds of cores and support hundreds of thousands of concurrent threads. A relatively small task that cannot take full advantage of these cores and threads would underutilize the computing capability of the GPUs. For example, if a task can only spawn 10 threads during its execution, computing it on a GPU that has 160 cores and supports 65535 concurrent threads would lead to 150 cores stay idle, which is a significant waste.

Furthermore, in our study, the GPUs used in our experiments do not support multiple concurrent kernels (i.e., the function running on a GPU thread), meaning that it is infeasible trying to assign multiple small tasks implemented by different kernels simultaneously to a GPU to reduce its underutilization. Therefore, without concerning about the video memory size constraints, increasing the size of a task (i.e., increasing the number of threads the task will spawn) assigned to a GPU is more likely to make the best of its computing power. The task bundling strategy mentioned above can also be applied to increase the size of a task and therefore reduce the occurrence of underutilization of GPU's computing capability.

(3) In GPGPU computing, the data transferring and kernel execution are difficult to be carried out concurrently. Taking CUDA as an example, although using pinned memory (i.e., the block of memory whose content will not be swapped out to secondary storage by operating systems; more details can be found in [32]) makes the concurrency of data transferring and kernel execution possible [33], it is still not a satisfactory approach because the size of pinned memory is limited; otherwise the performance of the whole system would be compromised. Therefore, overlapping of data transfer and computation, as many researchers did in the grid or cluster environments to improve performance, is not a practically feasible option in the CPU-GPGPU architecture of our study.

(4) GPU is actually a peripheral device for the CPU; the latter has to interact with the former via a device driver. A function call on this device (i.e., the kernel function call) through the device driver is not as cheap as compared to a function call on CPU. Take CUDA architecture as an example: the overhead of one GPU function (kernel

function) call, which is named as *kernel start-up* overhead, is approximately 9 μs. This time is quite significant compared with the average 3 *ns* overhead of a CPU function call [36]. The task bundling aforementioned can also be applied in here to reduce the kernel start-up latency.

(5) Considering that the original intention of introducing GPUs is for graphics rendering, and both the programming paradigm and the execution model of GPUs are for this purpose, GPUs are designed as a massive core processor suitable for stream processing. Hence the ideal GPGPU applications [33] should have large data sets, high parallelism, and minimal dependency between data elements. Although GPGPU is ideal for data-parallel computation, task-parallelism can also be achieved in this architecture using code branches in a kernel. In GPGPU, all the SPs (streaming processors) in one multiprocessor perform their execution in a SIMD fashion, while all multiprocessors are independent of each other. Therefore theoretically, these multiprocessors can concurrently go through different control flows and therefore exhibit the capability to execute in a task-parallel way. This characteristic will be considered when we develop the scheduling strategies for certain sub-categories of farm pattern, which will be elaborated in the following chapter.

2.1.3 CUDA

CUDA (Compute Unified Device Architecture) is developed by nVidia as a parallel computing architecture allowing users to program and carry out general-purpose parallel computing effortlessly on its GPGPU. CUDA includes C-extension programming syntax, runtime environment, complier, API and SDK. The syntax is simple and compatible with

C. The following is an example code showing how to design a kernel function (the function called by the CPU and running on a SP of the GPU) and call it on a GPU using CUDA.

```
const int N = ...; // the size of the array for the computation

//definition a kernel function called by the CPU and executing on the GPU
__global__ void mykernel(int* a, int* b, int* c, int N)
{
	int i = blockDim.x*blockIdx.x + threadIdx.x;
	if(i<N)
		c[i] = a[i] + b[i];
}


//host code
int main(void)
{
	int size = N*sizeof(int);

	//Allocate memory for vector a and b on the CPU's memory
	int* h_a = (int) malloc(size);
	int* h_b = (int) malloc(size);
	int* h_c = (int) malloc(size);

	//Initialize vector h_a and h_b
	...

	//Allocate memory for vector a and b on the GPU's memory
	int* d_a, d_b, d_c;
	cudaMalloc(&d_a, size);
	cudaMalloc(&d_b, size);
	cudaMalloc(&d_c, size);

	//Copy vector h_a and h_b to the GPU's memory
	cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
	cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);

	//Indicate the number of blocks and the number of thread in each block
	int blockNum = ...;
	int threadNumPerBlock = ...;
```

```
    //call kernel
    mykernel<<<GridDim, BlockDim>>>(d_a, d_b, d_c);

    //Copy the result back to the CPU's memory
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    //Release the GPU's memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

A kernel is a function executing on the GPU and it is defined using __ global__ declaration specifier. The symbol <<<..., ... >>> is the execution configuration syntax for indicating the number of CUDA threads that will be spawned to execute the kernel. GridDim indicates the number of thread blocks that will be yielded and BlockDim indicates the number of threads in one block. For example:

```
mykernel<<<4, 256>>>(d_a, d_b, d_c);
```

In above code, the number of blocks is 4 and the number of threads in each block is 256. So the total number of threads to be spawn by calling this function is 1024. In the mykernel function, each thread is mapped to a different element in the input arrays, as shown in the above code, by using the thread ids as index.

In CUDA's execution model, the thread distribution mechanism is based on *blocks* and "*warp*" [33]. In CUDA, multiple CUDA threads are organized as a block and multiple blocks are organized as a *grid*. So when calling a kernel, users need to specify the number of threads in one block and the number of blocks in a grid. CUDA runtime environment maintains a "block pool" on a GPU and assigns a block in the pool to one multiprocessor

(i.e., SM) once it becomes idle. When a multiprocessor is running, it is dedicated to its assigned block. Concurrent blocks assigned to one multiprocessor are not supported. Once a block is assigned to a multiprocessor, it will be partitioned into small pieces called “warps”. The multiprocessor will pick up one warp piece and execute it in a SIMD fashion on its SPs. Since the number of SPs in one multiprocessor is 8 for most nVidia GPUs, if the size of the block is too small (e.g., there is only one thread in a block), only a part of the SPs in the multiprocessor would be used and all other SPs would stay idle. This leads to a severe underutilization of GPU's power.

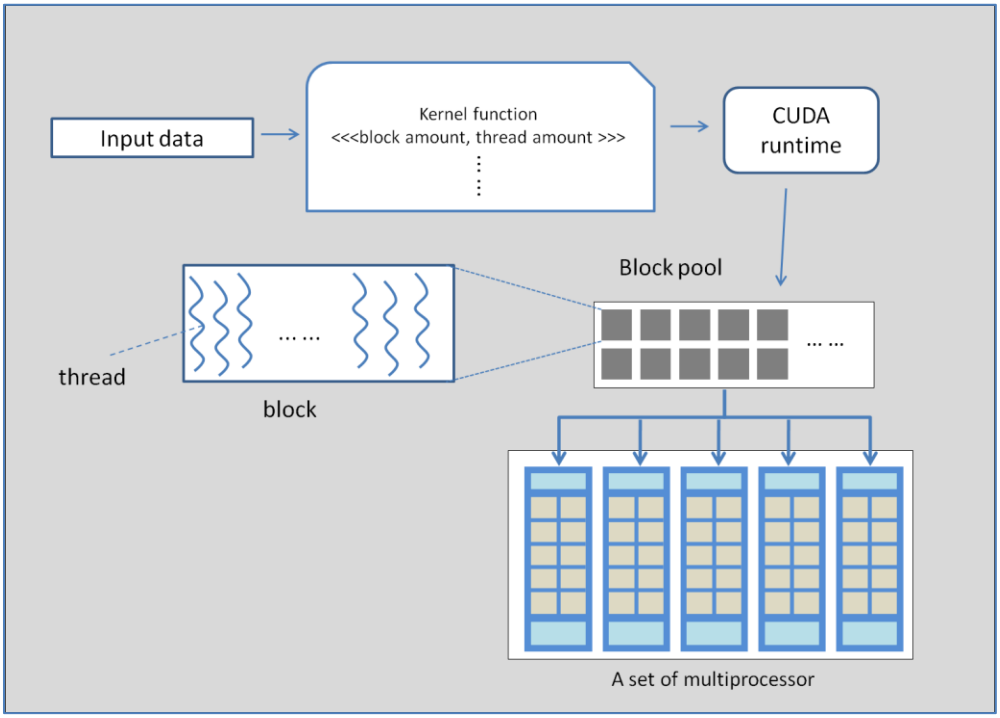

Figure 2: The model of thread distribution in CUDA

In practice, users are allowed to spawn much more CUDA threads than the number of SPs in a GPU in order to avoid I/O waste. It means while some threads wait for I/O, the processors still have other threads to execute. It implies that multiple thread warps in

CUDA run on processors concurrently. The CUDA runtime environment will map these threads to SPs automatically and users do not have to be concerned about the task distribution and load balancing among the SPs.

The programming model of CUDA is akin to SPMD (Single Program Multiple Data). In this model, users write one program—a kernel function, and specify the amount of threads to execute this program and each thread works on different data. This model is ideal for data parallel architectures, especially for GPGPU, since a GPU is a stream processor as aforementioned. On the other hand, although task parallelism is as well possible to be achieved in this model using program branches (e.g., if/else statement), too much extra programming effort would fall on users, not to mention extra branches in this parallel environment making programs more error-prone. Therefore, task parallelism is not encouraged for CUDA programming model. Nevertheless, under certain situations, this kind of parallelism can be exploited as an alternative to reduce, to some extent, the severe underutilization of GPU power. We will elaborate this in the following chapter.

2.1.4 Multi-core CPU

In the last several years, as the manufacture technology of microchips keep improving, the architecture of mainstream CPUs for PCs and workstations has shifted from single-core to multi-core. Nowadays, increasing the number of cores of CPUs has become a trend, from the initial dual-core to today's sixteen-core of certain high-end CPUs. Figure 3 is the example of Intel Core i7 processor's architecture [34]. This multiple cores structure is suitable for parallel computation: all the cores can perform computation concurrently.

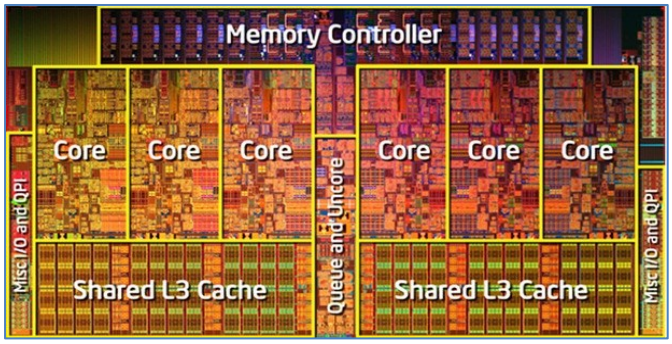

Figure 3: The architecture of the Intel's i7-970 multi-core CPU

For CPUs with parallel architecture, certain programming tools such as Intel TBB (Thread Building Blocks) are released in order to help users to leverage its computing power conveniently and efficiently without concerning data distribution and load balancing. TBB is a C++ template library containing several data structures and algorithms that relieve programmer's burden to manually deal with many issues arising from multi-thread programming, such as synchronization, creating and terminating threads, and load balancing among different threads, by using some native threading libraries like Windows threads and POSIX threads.

Current mainstream multi-core CPUs normally have no more than 16 cores, which is a quite small quantity compared with a mainstream GPGPU that could have hundreds of cores. But regarding each core's speed, normally a CPU will surpass a GPU. Therefore, CPUs and GPUs are suitable for different types of tasks. Certain type of tasks would have good performance on CPUs while others may have a less execution time on GPUs. Generally speaking, for a high-parallel task with a large input data set, GPGPUs are more likely to have a better performance than CPUs. On the other hand, if a task is

single-threaded, or with a small number of threads, assigning it to a CPU would get a less execution time than doing it on a GPU.

2.1.5 CPU-GPGPUs Architecture

The combination of CPU and GPUs conforms to a message-passing model, which means that they communicate with each other by passing messages via the bus on the motherboard of a machine, while both CPU and GPUs are also shared-memory architectures per se. This combination of message-passing and shared-memory parallel architectures provide a two-level parallelism to applications: coarse grain parallelism among CPU and GPUs and fine grain parallelism in CPU and GPUs. In addition, the CPU cores are independently clocked and hence they don't work under tight synchronization unlike the SPs inside a GPU's SM. Consequently, the CPU with its cores can be thought of as a multicomputer that supports task-parallelism; on the contrary each SM inside the GPU is a SIMD multiprocessor that supports tightly synchronous data-parallel computation, while these SMs are independent to each other. More characteristics about this architecture will be explored in next chapter while discussing the approaches for solving the scheduling problems on this platform.

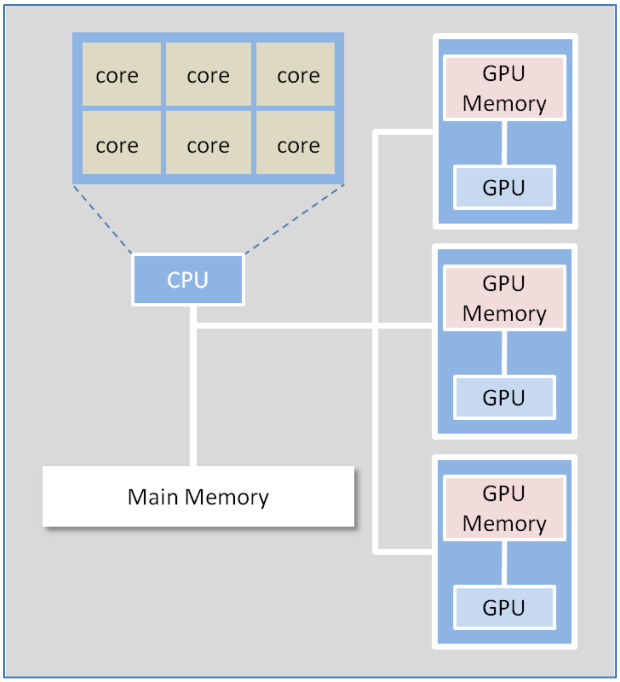

Figure 4: An example of CPU-GPUs parallel architecture with one CPU and three GPUs

The topology of CPU-GPUs architecture is a typical star-network. In this network, CPU is the center node and all GPUs are connected to it directly, and there is no direct connection among all the GPUs. The network connecting these processors is the bus on the motherboard of a PC or workstation.

It must be noticed that in our assumption, the architecture contains one CPU and a limited number of GPUs (Single-CPU-Multi-GPUs architecture), which is the common configuration of today's PCs and workstations. The number of GPUs in these machines is normally less than 3 and these GPUs communicate with the CPU through the same bus. Since the number of GPU is small and the bandwidth is quite high, the bus contention is not an issue. However, as the number of GPUs increases, the bus contention and the

shared bandwidth must be taken into consideration while designing scheduling strategies for this architecture.

Regarding Multi-CPUs-Multi-GPUs architecture, the today's operating system can hide the complexity of the multiple CPUs architecture and handle load balancing among these CPUs automatically, so from user's perspective the system contains only one CPU. Therefore this architecture does not have difference from Single-CPU-Multiple-GPUs architecture, so the scheduling strategies developed for the latter can also be applied to the former.

2.2 Framework Embedded with Scheduling Strategies

ASPARA (Adaptive Structured Parallelism) [20], as the succeeding work of eSkel [17], was developed at the University of Edinburgh in 2010. ASPARA is not a specific algorithmic skeleton, but a generic methodology to optimize the performance of skeletons in a grid. It proposes a way to embed scheduling strategies with algorithmic skeletons [15] and therefore it creates a scheduling-strategies-embedded (SSE) framework for parallel computing on heterogeneous distributed systems. The following example shows the application interfaces of a farm pattern and a pipeline pattern written in ASPARA. As shown in the example, the skeletons are built on MPI, which means that it is designed for message-passing parallel platform such as clusters and grids. It should also be noticed that, the skeleton for the farm pattern has a parameter that indicates which scheduling strategy will be applied as a suggestion from the application developer.

```
void pfarm(void(*worker)(), void *in_data, int in_length, MPI_Datatype in_type,
     void *out_data, int out_length, MPI_Datatype out_type, MPI_Comm comm,
     enum scheduling sched);
```
(a)

```
void pipeline(stage_t *stages, int no_stages, MPI_Datatype in_data[],
      MPI_Comm comm);
```
(b)

"ASPARA comprises a set of rules to be embedded with skeletons, where every rule essentially defines an application scheduling scheme which is parameterized in terms of the existing system resources" [20]. Since, as aforementioned, the behaviour of an application's pattern is known a priori to its execution, this behavioural information can be exploited by ASPARA to guide the scheduling of tasks in the application. Application development using ASPARA contains four steps: programming, compilation, calibration and execution.

In [20], the researchers make use of the methodology defined by ASPARA to address the scheduling problems of farm and pipeline applications. In the farm pattern, they assume that the workers are all identical and the tasks can be arbitrarily divided, which is akin to Divisible Load Scheduling problems [22] discussed later in this chapter. ASPARA uses a property called *Fitness* (*F)* to evaluate each processor. For a processor, a greater *F* value means more tasks will be assigned to it. Suppose the size of the input task is *a*, the number of available processor is *N* and the size of tasks assigned to processor *i* is a_i. Then,

$$a = \sum_{i=1}^{N} a_i$$

Furthermore, a_i can be calculated using *F* as: $a_i = a \times F_i$.

The value of F can be determined by several methods. The simplest choice is to use the execution speed of each processor.

Based on the F value, ASPARA presents two ways to handle a farm application's scheduling: single-round scheduling and multi-round scheduling. In the single round scheduling, F is computed and the input task is divided in terms of F (size of tasks assigned to processor i is $a \times F_i$ where a is the size of the input task) and scheduled in one time. On the other hand, in multi-round scheduling, tasks are divided and assigned in multiple times. Since ASPARA is built on a heterogeneous distributed environment and resources in this environment is non-dedicated, the availability and performance of the processors do not remain constant, which means that the value of F may change from round to round. Therefore, in this multi-round strategy, F is recomputed in every round, and hence the sizes of tasks assigned to processors will also be recomputed.

For the pipeline pattern, ASPARA employs a similar strategy: use Fitness F to evaluate each processor which acts as a pipeline stage. Once the value of F of a processor is no longer acceptable, it will drain the pipeline and hence replace the processor with another one with a better F to resume the data flow.

APSARA is built for grid and cluster computing platforms consisting of CPUs, where the value of the fitness function F can be easily computed in terms of the execution time of a CPU. On the other hand, in the CPU-GPGPU architecture of our interest, the GPUs and their execution behaviours are different from that of the CPUs. Furthermore, in ASPARA, the cost model of the worker processes is linear, which means that more loads lead to more execution time of a worker. On the contrary, for GPUs the cost model is sometimes

nonlinear. e.g., it can have the same execution time for tasks of different sizes in certain situations. For example, addition of two vectors of 10 dimensions will take approximately the same time as the addition of two vectors of 100 dimensions on a GPGPU with 128 cores. This difference arises due to difference in the programming models of CPUs and GPGPUs.

Furthermore, the multi-round scheduling strategy of one divisible load in ASPARA would cause too many kernel calls and too many message transfers, both of which have high start-up overheads as discussed before. Consequently, the strategies employed in APSARA are not straightaway applicable in our work.

2.3 Task Farm Pattern

The farm pattern is widely used in parallel programming. In this thesis, we study scheduling of the farm pattern and its variants on the heterogeneous system discussed before. Further classifications of the farm pattern for the purpose of our work are discussed in the next section. Furthermore, scheduling techniques for the farm pattern can be applied to the scheduling of some other patterns, e.g. data-flow and its specific form—pipeline.

According to the definition by Berna L. [13], a farm pattern consists of the following key components: a master (also known as farmer or task generator), multiple workers and a collection of independent tasks. In our work, we add another component—a scheduler to this pattern. The functionality of each component is as follows:

- **Task Generator**: generating tasks and putting them into a “pool”;

- **Worker**: accepting tasks assigned by scheduler, performing the indicated computations, returning the results;
- **Scheduler**: assigning tasks in the task pool to workers based on certain scheduling policies. The goal is to minimize the makespan. Scheduler is an application independent module and its behaviour is irrelevant to masters and workers.

The task generator, master and worker are defined by users, therefore these components are application-specific. There are additional application-specific plug-ins to the scheduler (e.g., partitioner), which will be elaborated in the following chapters.

Another key component of a farm pattern is a collection of independent tasks which are held by the task pool. However, a formal and precise definition for these "tasks" has not been given so far. M. Danelutto et al [35] indicated in their work that a "task" in a farm pattern is a data item in an input stream. In this case, the scenario is that all the workers carry out the same computation on different data items, which is akin to the SPMD (Single Program Multiple Data) programming model. This scenario yields a data-parallel farm. One example of this type of farm is searching through a database: each task has a sub-seachspace and all worker processes can concurrently perform the same searching strategy on different sub-searchspaces assigned to them by the scheduler. However, confining the type of tasks to data stream restricts the applicability of the farm pattern. Moreover, "task" in traditional computing terminology usually means some computational control flow (for instance, a task carried out by a function) that operates on input data. In practice, the farm pattern should cover a wild variety of problems, as long

as they contain a collection of independent tasks, no matter if the tasks are different data items or different control paths. We will elaborate this in the following chapter.

2.4 Existing Scheduling Strategies Applicable to the Farm Pattern

Tasks in a farm pattern based application, by definition, are independent of each other. In some farm pattern applications, the independent tasks can be further divided arbitrarily. Therefore, the scheduling in a farm pattern can be regarded as either *divisible load scheduling* (DLS) [22] problem or an *independent task scheduling* [23] problem. These two categories of problems have been studied for quite a time and many scheduling approaches have been put forward.

2.4.1 Divisible Load Scheduling (DLS)

A definition of divisible load scheduling problem is as follows: divide an input load into small chunks and assign them to processes in a way which can minimize the total idle time of the processes and hence minimize the makespan. In the case of a divisible load, the load can be further divided arbitrarily without violating any dependency constraints. An example of DLS problem is processing a large image file in a parallel environment. The large image file consists of a set of pixels which can be processed independently. Therefore, this image can be divided into smaller blocks and assigned to different processors for the purpose of speeding up the processing. The divisible load scheduling strategies focus on what is the near-optimal way to divide this image in order to minimize the makespan. In a farm pattern based application, when the size of a load is quite large and it is divisible, dividing it into smaller chunks and distributing the chunks to worker

processes in a load-balanced way is the basic idea of scheduling. One example SSE framework that employs this divisible load scheduling model is ASPARA [20].

As far as we know, most works in the field of divisible load scheduling are based on the network-based distributed architectures such as cluster and grid. Several approaches [24][25][26] are proposed for solving the DLS problems on grid. These approaches can be broadly classified into two categories: single-round divisible load algorithms and multi-round algorithms. Single-round algorithms divide the input load into multiple blocks whose amount is exactly as same as the number of processes. For example, if there are k processes available for the computation, single-round algorithms will divide the input load into k blocks. Therefore, each process will be assigned one block. On the contrary, multiple-round algorithm will divide the in a way where the number of blocks is more that the number of processes. Every process is assigned exactly one block in one round, and therefore it requires multiple rounds to complete this assigning, so each process will be assigned more than one block eventually. In practice, multi-round algorithm assigns more than one block to each process in a pipelined fashion, in order to overlap the data transfer time and computation time.

In [25], O. Beaumont et al proposed a single-round algorithm for star networks, which is also the topology of CPU-GPUs architecture (CPU is P_0 and GPUs are $P_i, i > 0$).

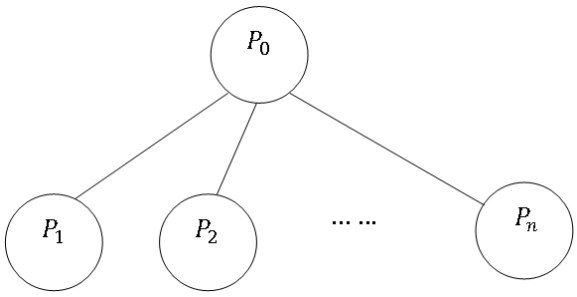

Figure 5: A star network

It employs a linear programming approach to solve the problem. The following lists the constraints of their linear programming.

$$
\begin{aligned}
&\text{MINIMIZE } T_f, \\
&\text{SUBJECT TO} \\
&\begin{cases}
(1)\ \alpha_i \geq 0 & 1 \leq i \leq p \\
(2)\ \sum_{i=1}^{p} \alpha_i = W_{\text{total}} & \\
(3)\ \alpha_1 g_1 + \alpha_1 w_1 \leq T_f & \text{(first communication)} \\
(4)\ \sum_{j=1}^{i} \alpha_j g_j + \alpha_i w_i \leq T_f & (i\text{-th communication})
\end{cases}
\end{aligned}
$$

Where α_i is the number of units of load sent to worker P_i , and p is the number of worker processes, hence the number of divided blocks is the same as the number of worker processes, which makes this method the single-round algorithm. W_{total} is the whole input data, T_f is the total execution time (i.e., makespan). g_i and $\omega_i$are linear cost model: it takes $X \times \omega_i$ time units to execute X units of load on worker P_i and similarly it takes $X \times g_i$time units to send X units of load from P_0 to P_i . In the constraints, the first communication means assigning the block to worker process 1, while the *i*-th communication means assigning the block to worker process *i.* However, as previously discussed, the GPU does not have such a linear cost model ω_i when the size of tasks is small.

Regarding multi-round algorithms for DLS problems, the main purpose of these approaches is to overlap the data transferring time and the computation time, which is a direct consequence of the assumption that data transferring and computation can be carried out concurrently in their models. However, in the current CPU-GPUs architectures as discussed before, the data transferring and kernel execution in a GPU can hardly be carried out concurrently. Therefore, these approaches are not applicable to contemporary CPU-GPU architecture of our interest.

For CPU-GPUs architectures, to the best of our knowledge, Qilin [30] is the only approach that focuses on divisible load scheduling. It employs an adaptive mapping scheme that introduces a training phase. The training phase interpolates a system of linear equations based on empirical results from adaptive mappings, whose solution provides the best partitioning among the CPU and the GPUs. To be more specific, they firstly build linear equations: $t_c = a_c \times s + b_c$ for CPU and $t_g = a_g \times s + b_g$ for GPU, where t is the execution time and s is the size of the input data. Then they use the following graph to find the near-optimal partition.

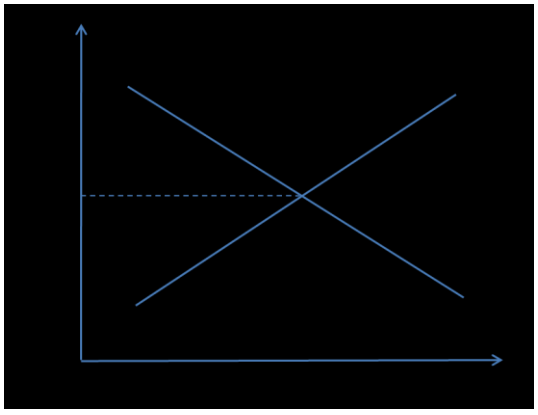

Figure 6: Task distribution curve in Qilin

In Figure 6, time means the execution time and β is the proportion of loads assigned to CPU, so it ranges from 0 to 1. The β value at the point of intersection of the two curves can yield the near-optimal partition.

However, as analyzed before, the linear model is applicable for GPU when the load size is fairly small. Besides, considering that they exploit the curve intersection approach, the current Qilin is only applicable for a single-CPU-single-GPU environment. Its extension to single-CPU-multiple-GPU environment may not be straightforward and could not be found in available literatures.

2.4.2 Independent Task Scheduling

Independent task scheduling problem focuses on generating a schedule for a set of independent tasks on a set of processing units, for the purpose of minimizing the total execution time. The tasks are independent, meaning that there are no communication or precedence constraints among them.

If the loads in a farm application are comparatively small and hence need not be further partitioned, then scheduling of a set of these small loads can be regarded as an independent load scheduling problem, which is akin to independent task scheduling as the terms load and task have been intermixed in several of these works. Note that we will further elaborate on the terms “load” and “task” in the next chapter. There is a considerable amount of work towards independent load scheduling in the last few decades, and the proposed approaches are applicable in cluster and grid environments. A survey of these approaches can be found in [27][28], out of which min-min heuristic is found to perform as the best [27]. In min-min, the next minimum load is always picked up for scheduling as compared to max-min in

which the maximum load is scheduled next. These approaches are designed for independent tasks scheduling.

```
(1)  for all tasks t_i in meta-task M_v (in an arbitrary order)
(2)       for all machines m_j (in a fixed arbitrary order)
(3)            c_ij = e_ij + r_j
(4)  do until all tasks in M_v are mapped
(5)       for each task in M_v find the earliest completion
               time and the machine that obtains it
(6)       find the task t_k with the minimum earliest
               completion time
(7)       assign task t_k to the machine m_l that gives the
(8)            earliest completion time
(9)       delete task t_k from M_v
(10)      update r_l
(11)      update c_il for all i
(12) enddo
```

Figure 7: The min-min heuristic

The Min-min heuristic shown in Figure 7 is presented in [27]. The input of the algorithm is a set of independent tasks (i.e., meta-task).

"In the figure, r_j is used to denote the expected time that machine m_j will become ready to execute a task after finishing the execution of all tasks assigned to it at that point in time, and e_{ij} is used to denote the execution time of task t_i on machine m_j. First the c is the matrix of completion time and c_{ij} entries are computed using the $e_{ij}$and r_j values. For each task t_i, the machine that gives the earliest expected completion time is determined by scanning the *i*th row of the *c* matrix (composed of the c_{ij} values). The task t_k that has the minimum earliest expected completion time is determined and then assigned to the corresponding machine. The matrix *c* and vector *r* are updated, and the above process is repeated for tasks that have not yet been assigned a machine" [27].

However, the performance of directly applying these approaches to loads scheduling in CPU-GPUs environment would not be acceptable due to several reasons. Firstly, these heuristics assign loads to processors one by one, which would cause too many message start-up calls and kernel calls and the overhead of these calls are non-trivial. Also, considering that the size of loads is quite small in some cases and the number of cores in GPUs is large, assigning the small loads one by one would lead to severe underutilization of the computing capability of GPUs. Moreover, most of these heuristics are static. Though dynamic versions for some heuristics have been designed [28], they are more applicable for a cluster and a grid environment where constraints are different. Lastly, most of these heuristics are applicable when the execution time is proportional to the load (data) size, which may not be always the case as discussed in the next chapter.

Chapter 3 Scheduling for Farm Pattern Based Applications

In this chapter, we propose several approaches to solve the task scheduling problems of the farm pattern based applications on a CPU-GPGPUs platform. These approaches make use of the underlying infrastructure and characteristics of the farm pattern when assigning tasks to processors. Similar approaches can be applied to the scheduling of pipeline and data-flow graph patterns and these are discussed at the end of this chapter. The next chapter elaborates the experimental results comparing the new approaches with some of the traditional approaches. As mentioned in the previous chapter, the characteristics of the tasks will affect the scheduling policy. In this chapter we first classify the farm pattern into several sub-categories based on the features of the tasks. Then we propose a set of scheduling strategies called HASS (Heterogeneous Architecture Scheduling Strategies) for these sub-patterns.

3.1 The Architecture

In general, the software architecture of a task farm pattern comprises of the following components: one or more task generators, a task pool where tasks are deposited by the generators, worker processes/threads that execute tasks from the task pool, a result pool where results are deposited, and result collector(s) (Figure 8). It must be noticed that workers not only consume tasks from task pool, they also generate tasks and deposit them to the task pool, i.e., workers also can act as task generators. This gives more flexibility in the use of the pattern.

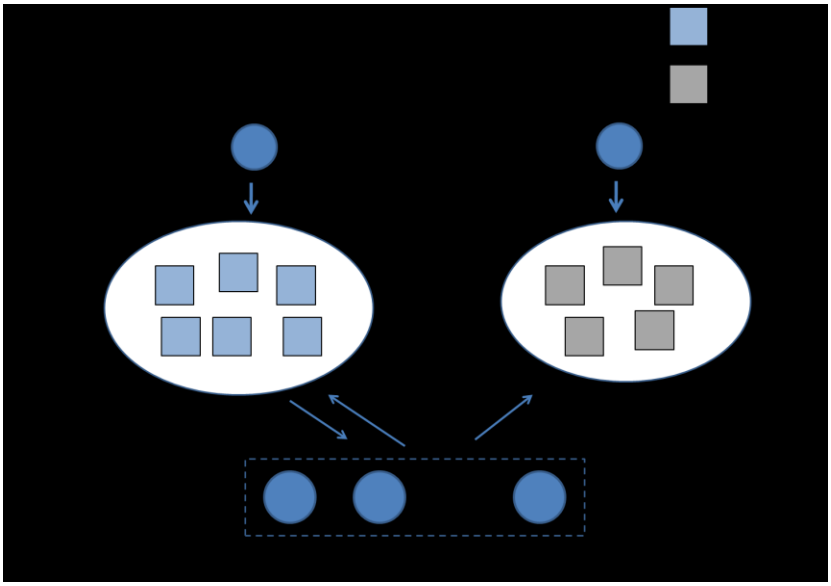

Figure 8: A task farm architecture

In the traditional task farm architecture, an idle worker pulls work from the task pool. However, since the GPU being a subordinate processor of the CPU, its work load needs to be pushed. The CPU still pulls its workload. Parts of these functionalities are performed by the scheduler.

The scheduler is an intermediate layer that sits between the task pool and the workers. It is responsible for ideal scheduling of tasks to workers so that all loads are well balanced and the total processing time is minimized. In our discussion, the scheduler is called HASS (Heterogeneous Architecture Scheduling Strategy) and is elaborated in the following sections.

Figure 9 illustrates the general configuration of a single-CPU-multiple-GPGPU architecture. The CPU is the predominant processor and each GPU is a subordinate processor under the control of the CPU. The CPU has multiple cores (e.g., 2 to 8) which can execute different codes, thus resulting in task-parallelism. In comparison, the GPU

has many more cores (e.g., from dozens to hundreds). The GPU cores are grouped into physical units, called streaming multiprocessors (SMs). Each SM is comprised of streaming processors (SPs) which communicate via shared memory and its execution model is SIMD, i.e., the SPs inside a SM synchronously execute the same code (also known as kernel code) but on different data. Hence the corresponding programming model inside an SM is of data-parallelism. Each SM can run different kernel codes, thus resulting in task-parallelism. In general, the GPU is more suited for data-parallel computation which results in less idling of its cores and thus more utilization of its resources.

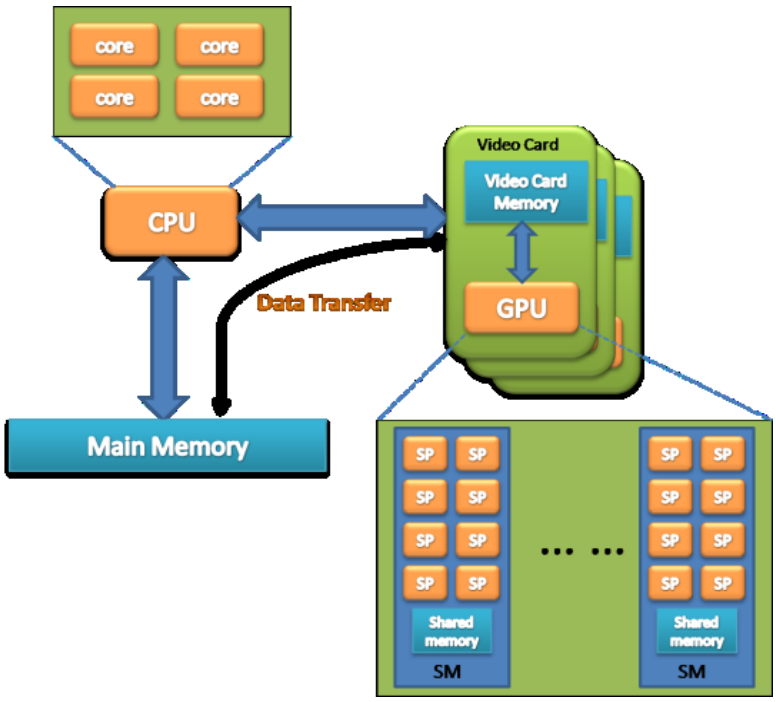

Figure 9: The CPU-GPGPU architecture

The following are some of the features of a contemporary CPU-GPGPU architecture which need to be considered while designing a scheduling strategy for such architectures. Firstly, in contemporary architectures the interconnection bandwidth between CPU and GPU is

fairly high, of the order of 10 gigabytes per second or even more with newer interconnection technologies like PCI bus and its extensions [12]. Considering this high bandwidth, data transfer time between the CPU and the GPUs is much smaller as compared to a cluster or a grid environment. Sometimes this message start-up cost becomes the predominant factor in the total data transferring cost between the CPU and the GPUs, especially when the size of data to be transferred is small.

Secondly, the architecture of an SM is SIMD and hence the GPU is more suited for data-parallel model of computation. All cores (SPs) inside an SM execute identical code, but on different parts of the input data. Consequently the bigger the size of the load, (whether divisible or indivisible), the more of the GPU cores can be kept busy, thus better utilizing the available cores. A smaller load could keep some of the cores idle, e.g., if the SM has 64 cores and the load can be utilized by only 20 cores then the remaining 44 cores will be idling. Though multiple SMs could run different kernel code, thus resulting in task-parallelism, this may not be achieved in a straightforward way in many existing GPGPUs.

Thirdly, in the contemporary GPU computing, it is difficult to overlap data transferring with concurrent kernel execution in a GPU. Take CUDA as an example [4], where this type of overlapping can be achieved only with great difficulty, e.g., using pinned memory which is limited in size and its use might negatively affect the performance. As a result, commonly used performance measures like overlapping of computation with communication as in a traditional cluster or grid environment may not be feasible in the CPU-GPGPU architecture.

Lastly, a GPU is a peripheral device for the CPU. Consequently, any kernel function call has to be done via device driver and thus has a much higher start up latency as compared to a function call in the CPU. For example, in CUDA, a GPU kernel call can have start-up latency about 3000 times higher than a CPU function call.

Since the CPU is the dominant processor and each GPU works under the control of the CPU, in our design the task pool and the result pool of the farm reside in the main memory of the CPU. Loads are subsequently distributed to the memory of the graphics card as will be discussed in the following strategies. The worker threads run on the CPU and the GPU: they will be called CPU workers and GPU workers respectively. Note that the CPU and GPU workers perform identical works; however their codes are different due to the difference in their architectures and programming models.

3.2 A Classification of the Farm Pattern

In this section we propose a new way to classify the farm pattern based on the type of the tasks in this pattern. For each category, we propose a corresponding heuristics to solve the scheduling problem on the CPU-GPGPUs platform.

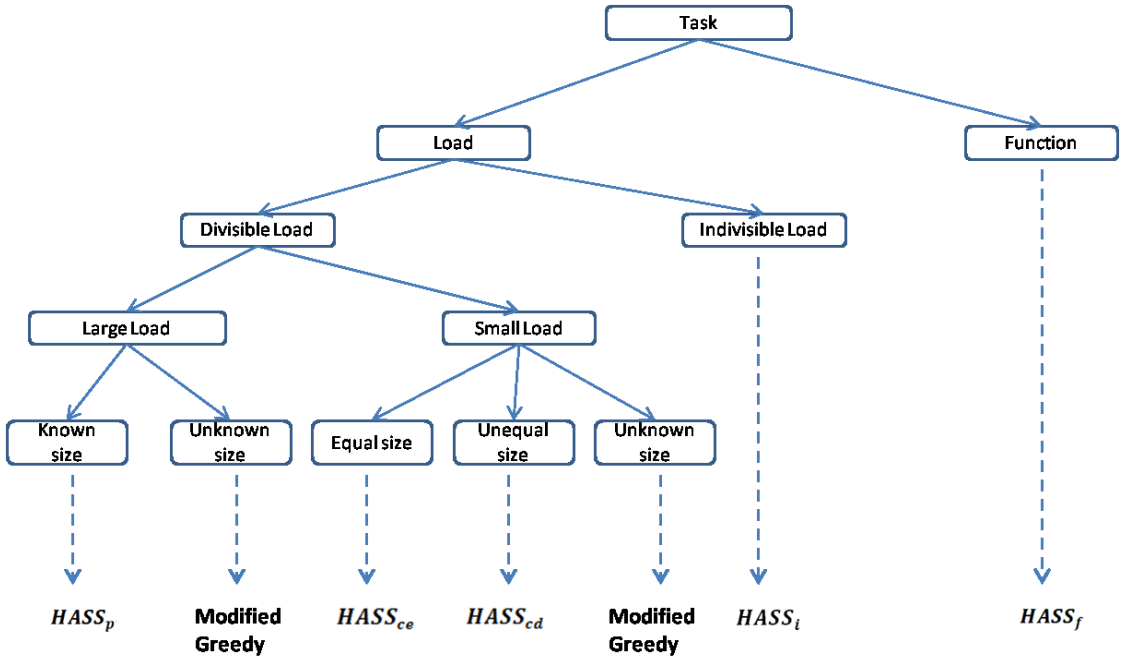

Figure 10: A taxonomy of farm pattern in terms of the type of the tasks and the corresponding heuristics

Figure 10 presents an entire classification of the farm pattern and the heuristics, which will be elaborated in the following. It should be noted that for the farm pattern, all tasks are independent irrespective of which category it falls into.

In the literature, the tasks in the task pool of a farm pattern are regarded as data items. In [35], the author explicitly defines the tasks in a farm pattern application as data items and the task pool as a data stream. In [20], a task is defined as a divisible load which is a synonym of data. This assumption implies that the worker processes are all identical and the only difference is that they take different data as input. Hence, this task farm pattern is essentially a data-parallel model of computation.

However, to be more flexible, the types of tasks in the farm pattern need not be restricted to only data items; they can be functions as well. Therefore, we classify a farm pattern into

two sub-categories: data (or load) farm and function farm (Figure 9). In the data farm, the task pool is essentially a data pool, and a load in the pool is a collection of data elements where each data element is a portion of the load on which a worker operates. . For example: in a matrix-matrix multiplication, a load comprises of the two matrices to be multiplied and if a worker is responsible for producing an entry of the result matrix then the corresponding (input) data element for the worker is a row of the first matrix and a row of the second matrix. In the case of GPGPU computing, each data element is usually mapped to one worker. Therefore, the number of workers in GPGPU computing depends on the number of data elements of a load. Many of such workers can in turn be mapped to a GPGPU core (Figure 11). On the contrary, for CPU computing, multiple data elements could be mapped to each worker, i.e., it is a many-to-one mapping. For example, each worker is in charge of one or several rows or columns of the result matrix. As in a GPGPU, more than one worker can be mapped to a CPU core (Figure 11).

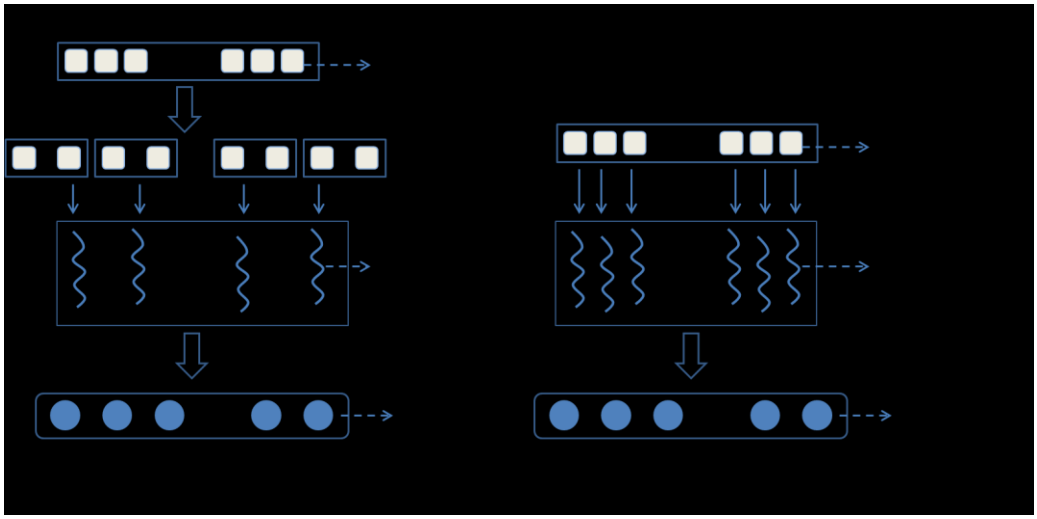

Figure 11: (a) the mapping of data and worker in CPU; (b) the mapping in GPGPU

On the other hand, in the function farm each task is a function together with its input data. Each function is mapped to a worker and thus the workers could carry out different computations by receiving different tasks (Figure 12). Hence, the computational model in this case is of task-parallel. As discussed previously, task-parallel computations are best carried out in the CPU due to its MIMD architecture. However, considering the GPU's potential to execute task-parallel applications as discussed before, function farm applications can as well be implemented on GPGPUs. By designing the farm's scheduling strategies carefully, we can have a function farm with good performance. Even though sometimes these task-parallel applications can cause severe wastage by idling the GPU cores, a well-designed scheduler is able to reduce this waste as much as possible and thus make the performance more acceptable. Some examples of the task farm pattern are the ready queue in DAG scheduling and the PCB queue in an operating system.

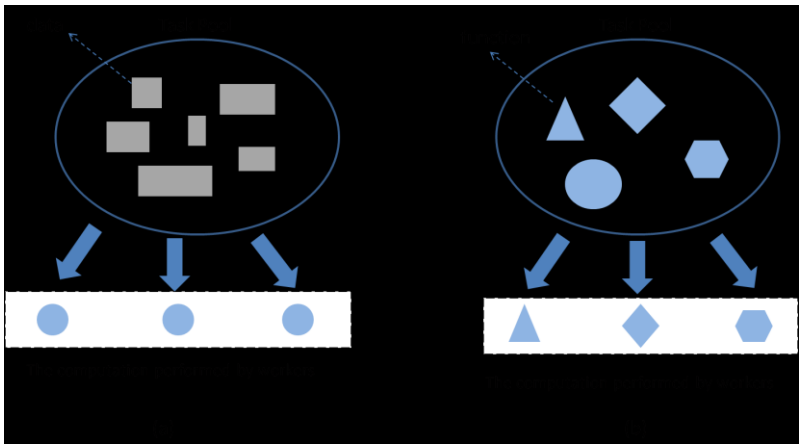

Figure 12: Comparison of data farm and function farm

3.3 Further Classifications of the Data Farm Pattern

This section presents a further classification for the load (data) farm pattern based on the divisibility of the loads: the farm with divisible loads and the farm with indivisible loads,

which are called as divisible and indivisible load farm respectively in the following discussion.

In the divisible load farm, each load can be arbitrarily partitioned into independent smaller loads. On the other hand, in the indivisible load farm, the workers operating on different elements of a load have inherent dependencies and hence the load cannot be arbitrarily divided.

Besides the classification mentioned above, the scheduling of divisible load farm can have two broad classifications: large load scheduling versus small load scheduling (Figure 10). In the large load scheduling case, the size of each input load is fairly large, based on certain parameters of the underlying architecture, e.g., number of cores in a GPU block. Such a load must be partitioned into multiple smaller loads (Figure 13). Therefore, this situation is called the *partition case*. For this case, it can also be further be classified into two sub-categories: known size task case and unknown size task case. The known size task means the execution time of the tasks which take the loads as input is proportional to the size of the loads, while the unknown size task means the execution time is non-proportional to the size of the loads. An example of the known size task case is image processing, where each pixel needs to be processed. On the other hand, an example of the unknown size task case is searching through a text space to find out the first words with a certain pattern. The known size task case is akin to a divisible load scheduling (DLS) problem. In this case, the goal of our strategy is the same as that of Qilin[30] system: trying to find an optimum partition of the input load with the intention of minimizing the makespan. Compared to the Qilin approach, the improvement of our approach is that it is applicable to the single-CPU-multiple-GPU environment, whereas the approach in Qilin

system only works in the single-CPU-single-GPU environment. $HASS_p$ (p means partition case) is the proposed heuristic in this category. On the other hand, if the task size is unknown then we propose a modified greedy strategy (Figure 10).

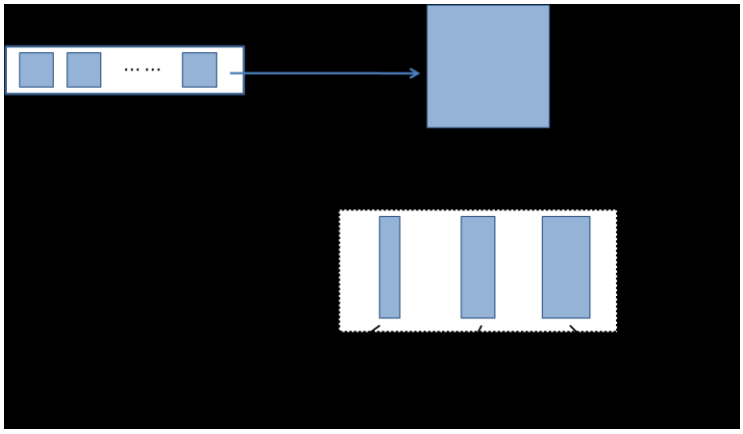

Figure 13: The execution model of partition case

In the small load scheduling case, the size of each load is comparatively small based on similar parameters of the underlying architecture. Therefore, it is unnecessary to further partition a load, and scheduling of these independent small loads boils down to the independent task scheduling problem. Instead of being partitioned, loads assigned to the GPUs need to be bundled for the purpose of reducing their execution time (Figure 14). The need for load bundling is further discussed in section 3.4.1. Hence this scenario is also called the *load combination* case. In case can also be further classified into several categories: known size task case and unknown size task case, and the known size task case also has two situations: all tasks have equal size or different. For this scenario, we propose two different strategies: $HASS_{ce}$ (c represents combination or bundling and e represents equal sized tasks respectively) for the farm pattern with equal task sizes;

$HASS_{cd}$ (d stands for distinct task sizes) for the farm pattern with distinct task sizes. A modified greedy scheduling strategy is proposed for the unknown size task case.

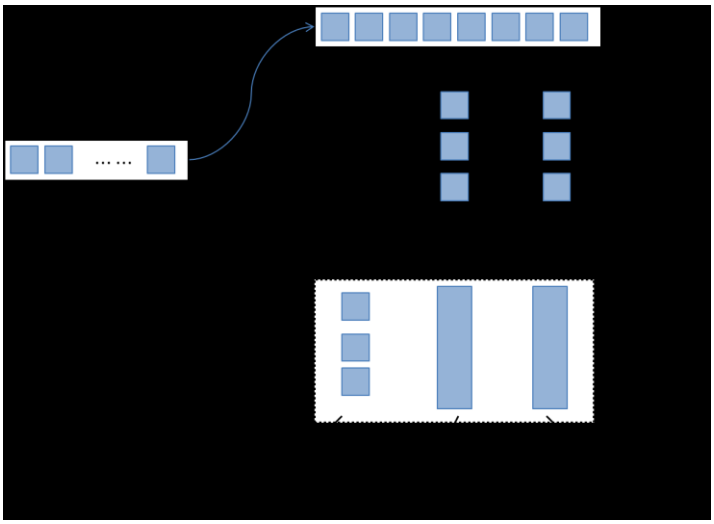

Figure 14: The execution model of combination case

Although we have this large load case and small load case classification, in practice, it is difficult to determine whether a load is large or small in an abstract way, since it is dependent on the configuration of the underlying infrastructure and so far a standard rule which can differentiate these two cases in terms of the configuration of a system has not yet been proposed. Therefore, in this research, we only broadly put forward these two cases so that for each case we can design certain scheduling strategies.

All the aforementioned approaches are for the divisible load farm. Regarding the indivisible load farm, the strategy $HASS_i$ is proposed. These strategies are elaborated in the following sections.

3.4 New Approaches for the Divisible Load Farm

3.4.1 The Impact of Load Bundling in GPU Performance

Load bundling means to bundle multiple small loads into a single large load and assign this one large load to the GPU. Therefore, in this case, the requirement is that all small loads must be able to be bundled. For a divisible load, since it can be arbitrarily partitioned into smaller loads, so these smaller loads can also be bundled arbitrarily.

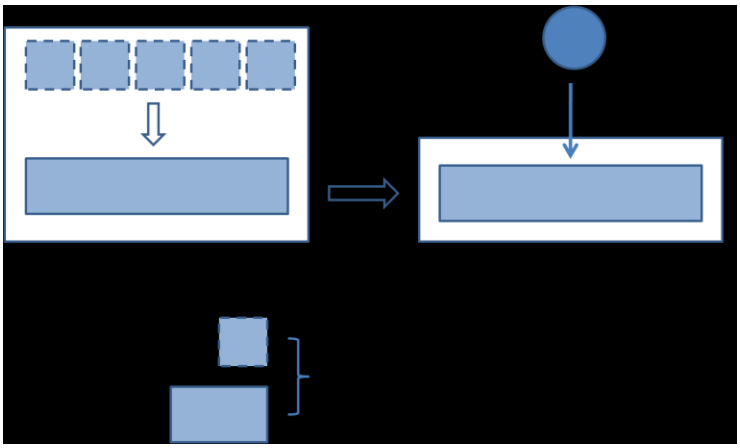

Figure 15: An example of load bundling for a GPU

The advantages of one single large load versus several small loads are that the message start-up latency, associated with each message between CPU and GPU, and kernel function start-up latency, associated with each GPU kernel call, are significantly reduced. Furthermore, considering that in CUDA programming, each data element in a load is mapped to one thread, i.e., worker, and one or more threads are mapped to a GPU core, if a load is too small, i.e., the number of its data elements is small, not enough threads can be spawned to take advantage of all the cores in a GPU. By applying load bundling, the bundled large load can relieve this underutilization by augmenting the amount of data

elements and consequently increasing the number of concurrent threads on a GPU. Based on the above discussion, it is preferred that the GPUs are assigned large size loads, as determined by the number of cores inside the GPU. Figure 16 gives an example of the influence of load bundling on the GPU's execution time.

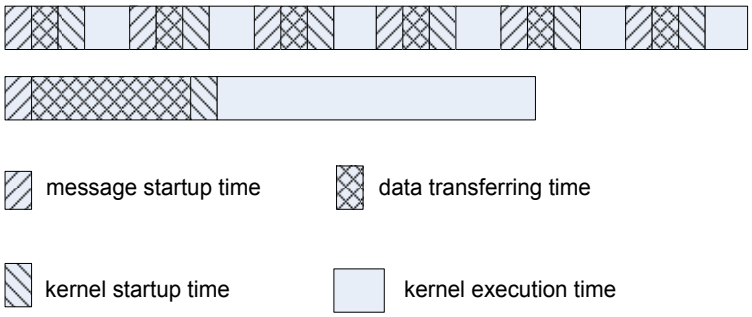

Figure 16: Impact of load bundling

Figure 17 is an example of searching through an array of 40 elements using a GPU with 80 cores. As shown in the figure, half of the cores are unused and therefore the GPU is severely underutilized.

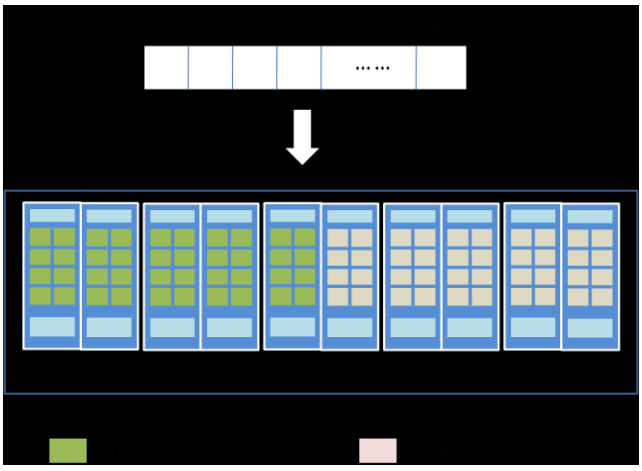

Figure 17: An example of underutilization of a GPU.

By bundling many small loads into a large load, we can reduce the occurrence of this underutilization problem. The following figure is an example of traversing arrays. In this example, the two small arrays are combined into one large array and then be assigned to a GPU. All the cores of the GPU are involved in the computation and the underutilization problem is eliminated.

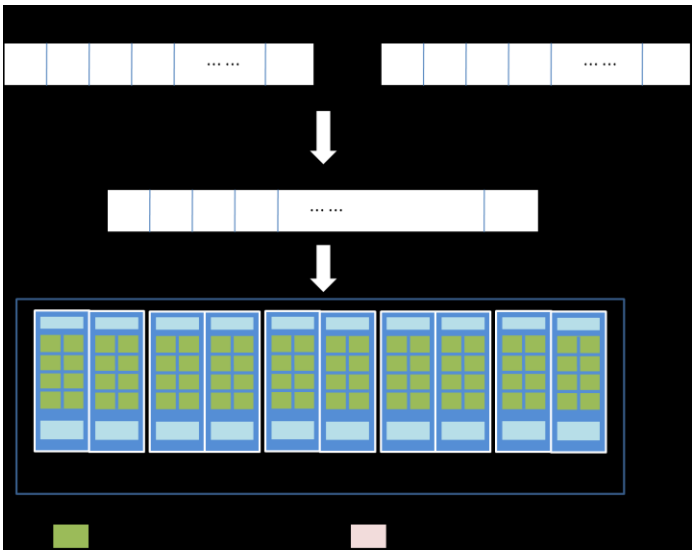

Figure 18: An example of full utilization of a GPU

3.4.2 The Heuristics for Divisible Load Scheduling

The following heuristics comprise of the following: (i) independent load scheduling, (ii) work stealing, (iii) load bundling, and (iv) learning and adaptation. As discussed before, a divisible load is first partitioned into independent smaller loads. An effective independent load scheduling strategy (e.g., Min-min) is first applied to do an initial distribution of the (partitioned) independent loads to the processors. This initial distribution may not be the best and hence it might need to be adjusted in subsequent assignments. Work stealing is a way to adjust the loads, where the idling GPUs steal work from CPU and vice versa. This

leads to a learning and adaptation phase, during which load distribution to the processors is adjusted for near-optimal performance. The discussion assumes a single-CPU-multiple-GPU architecture. The CPU and each GPU has its own buffer (in main memory) where loads are assigned. Subsequently the assigned loads are transmitted to the GPU with or without bundling, depending on the strategy employed.

We start with basic HASS, which is used inside some of the other heuristics presented here. In basic HASS, the input is a set of independent loads. Basic HASS is a two-phase approach: the first phase is the *mapping* phase. In this phase, certain traditional independent task scheduling heuristic (e.g., Min-min) is used to generate an initial mapping from loads to processors. The loads mapped to each processor are stored in the processor's buffer (in main memory). In *Execute* phase, loads are transmitted from processor's buffer to the processor for execution. In transmitting to a GPU, the loads inside the GPU's buffer are bundled together and assigned to the GPU as a whole for the purpose of removing the influence of underutilization of GPU and the overhead caused by message and kernel start-up latency, as discussed before. Therefore, a load bundler is needed by the scheduler (Figure 19). In addition, basic HASS rebalances the load between the CPU and the GPUs by using a work stealing strategy: Once a GPU becomes idle, it will request loads from the CPU's buffer. This load rebalancing strategy can ensure the CPU and the GPU complete their execution at almost the same time with minimized idling.

It should be noted that Min-min requires the estimated execution time of a task. In basic HASS, the size of a load is used to estimate the execution time of the associated task. It is assumed that the execution time is linear to the size. However, as was discussed before, when the size of a load is small (i.e., the number of data elements is less than the number of

cores in a GPGPU), its execution time is not linear to the size, and it can cause an incorrect time estimation of Min-min when applied to GPGPU scheduling. When this situation happens, the computing power of GPGPU would be underestimated. For example, suppose the linear model of size and execution time is $t = 2 \times s$, where t is the execution time and s is the size, then a load with size 10 would have execution time 20 units and a load with size 20 would have execution time of 40 units. However, in a GPGPU with more than 40 cores, these two loads will have the same execution time. Hence estimation for Min-min will not be accurate. Therefore, in the following heuristics, min-min is used for initial mapping and subsequently load-bundling and/or work stealing are applied to handle any inaccuracies of Min-min.

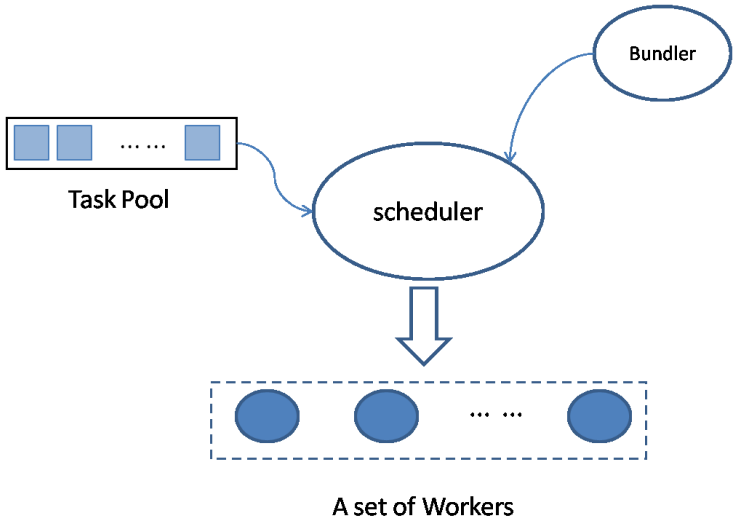

Figure 19: The scheduling model of divisible load

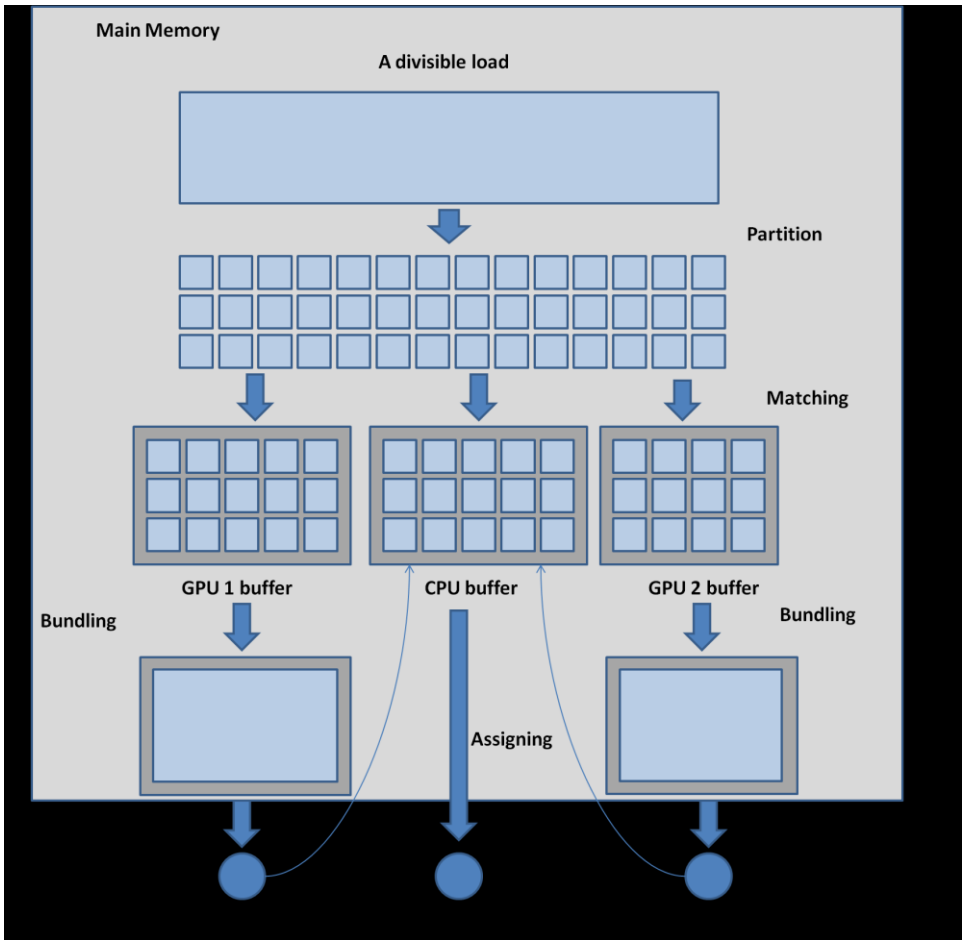

Figure 20: An example of using basic HASS on a CPU-GPUs system

Basic HASS is presented in the following:

Algorithm 1: HASS
Input: A set of independent loads D = {d_1, d_2,...,d_m}.
/*The loads are to be executed on processors P = {p_c, $p_{g,1}$, $p_{g,2}$, ..., $p_{g,n}$}; p_c is the CPU and $p_{g,i}$ is a GPU*/
Output: Loads executed with minimized makespan.
Begin
{Mapping phase}
1. Apply min-min heuristic to map loads to processors.
Loads are first assigned to processor's buffers residing on
main memory: b_c is the CPU buffer and $b_{g,i}$ is a GPU buffer.
2. After mapping is complete, go to the execute phase
/*step 3 below*/.
3. {Execute phase}

```
    for i = 1 to n do
    {
        Bundle all loads in b_{g,i} into a single load D_{g,i}.
        Assign D_{g,i} to p_{g,i}.
    }
    while b_c is not empty do
    {
        if p_c is idle then remove a load d_i from b_c and assign to p_c.
        for each GPU p_{g,k}
            if p_{g,k} is idle then
                /*Steal work from CPU's buffer*/
                remove a load d_m from b_c and assign it to p_{g,k}
    }
end
```

Basic HASS employs work stealing to re-balance loads among the CPU and the GPUs. The next heuristic $HASS_{ce}$, which is applicable for equal sized independent loads, employs learning and adaptation strategy in multiple rounds to re-balance the loads. $HASS_{ce}$ employs basic HASS during the first round of scheduling. The scheduling information (i.e., the ratio of loads executed by each processor over the total loads) is recorded, and this record is used to guide the scheduling in the following round. In the next round, all loads from the task pool are retrieved and mapped to the processors based on the recorded information from the previous round. Any load re-balancing is done if necessary and any change in scheduling ratio is recorded for the next round.

Algorithm 3: $HASS_{ce}$

Input: A set of loads is generated by task generator and deposited to the task pool continuously.

/*The loads are to be executed on processors $P = \{p_c, p_{g,1}, p_{g,2}, ..., p_{g,n}\}$; p_c is the CPU and $p_{g,i}$ is a GPU*/

Output: Loads executed with minimized makespan.

Begin

1. Initialize variables i and j_k to 0;
   /*Here i is the proportion of the work that has to go to the CPU and j_k is the proportion of the work that has to go to GPU k*/
2. Extract all loads from the task buffer and use L_q to denote the loads
3. Apply HASS to schedule loads in L_q.

```
4. Update i and j_k for each p_{g,k} based on the scheduling in 3 above.
5. Go to the adaptation phase /*step 6 below*/.
6. {Adaptation phase}
  while task generator is still generating loads
  {
    Extract all loads from the task buffer and use L_q to denote the loads
    for k =1 to n do
    {
        put q × j_k loads into GPU buffer b_{g,k}.
        bundle all loads in b_{g,k} and assign the single bundled load to p_{g,k}.
        put q × i loads into CPU's buffer b_c.
    }
    while b_c is not empty do
    {
        if p_c is idle then remove a load d_i from b_c and assign to p_c.
        for each GPU p_{g,k}
        {
            if p_{g,k} is idle then
            /*Steal work from CPU's buffer*/
                remove a load d_m from b_c and assign it to p_{g,k.}
         }
          update i.
          for each GPU p_{g,k}
             update j_k.
        }
    }
  }
end
```

$HASS_{cd}$ is developed for the case when the loads are of different size for the independent task scheduling. The difference of $HASS_{cd}$ from $HASS_{ce}$ is that, since the size of each load is different in every round, there is no useful scheduling information from the previous round to guide this round. Therefore, in each round, traditional scheduling heuristics must be used and the GPU workers have to steal loads from the CPU worker. However, instead of asking the GPU workers to steal one load each time as $HASS_{ce}$ does, the GPU workers in here steal $\frac{M}{(n+1)}$ tasks from CPU, where *M* is the number of tasks in CPU worker's buffer, and *n* is the number of processors. The reason to replace the

"*single-stealing*" with this "*multi-stealing*" is that stealing one task per time would cause both probable GPU underutilization and a large number of work stealing, which would consequently lead to much many message transferring and kernel calling. This multi-stealing strategy may cause an over-stealing (i.e., GPU workers stealing too many tasks and leaving the CPU has no tasks), but the time reduced by this multi-stealing can amortize the loss caused by over-stealing. It should be noticed that this multi-stealing strategy is not suitable to $HASS_{ce}$**,** in which case the scheduling information from the last round is exploited to guide the scheduling of the next round. Over-stealing would generate incorrect scheduling information that is useless for the next round scheduling. The following is the algorithm of $HASS_{ce}$, where **K** is used to denote all the tasks in task pool.

Algorithm 3: $HASS_{cd}$

Input: A set of loads is generated by task generator and deposited to the task pool continuously.

/*The loads are to be executed on processors $P = \{p_c, p_{g,1}, p_{g,2}, ..., p_{g,n}\}$; p_c is the CPU and $p_{g,i}$ is a GPU*/

Output: Loads executed with minimized makespan.

Begin

1. Extract all loads from the task buffer and use L_q to denote the loads
2. Apply basic HASS to schedule loads in L_q.

while loads are still generated by the task generator do

{

Extract all loads from the task buffer and use L_q to denote the loads

for k =1 to n do

{

put $q \times j_k$ loads into GPU buffer $b_{g,k}$.

bundle all loads in $b_{g,k}$ and assign the single bundled load to $p_{g,k}$.

put $q \times i$ loads into CPU's buffer b_c.

}

while b_c is not empty do

{

if p_c is idle then remove a load d_i from b_c and assign it to p_c.

for each GPU $p_{g,k}$

{

```
                    if p_{g,k} is idle then
                    {
                        /*Steal work from CPU's buffer*/
                        the amount of tasks in bc is M.
                        remove M/(n+1) loads from bc, bundle them and assign the bundled
                        load to p_{g,k}.
                    }
                }
            }
        }
end
```

The previous three algorithms are for divisible loads which do not need any further partitioning. However, if a load needs to be further partitioned into smaller loads then the next algorithm, $HASS_p$ is proposed. In this algorithm, the scheduler needs a *partitioner*, as a plug-in module, to partition each input load into smaller loads (Figure 21).

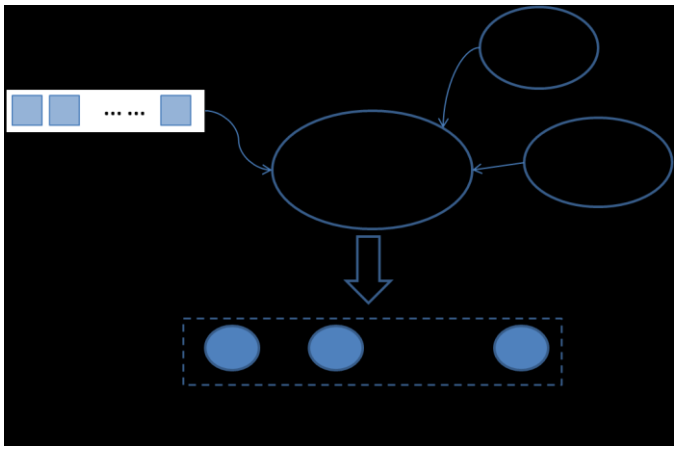

Figure 21: The scheduling model of divisible load

In $HASS_p$, the size of the input load is large and hence it needs to be divided into smaller loads prior to assigning to processors. The purpose of this algorithm is to find a nearly optimal way to divide the input so as to balance the load among the processors. The HASS algorithm is also the basis for $HASS_p$ (p here stands for partitioning). The $HASS_p$ algorithm works on multiple rounds: in round i, a divisible load is partitioned into

chunks of smaller loads of equal size using the partitioner and then HASS is employed to schedule those independent loads. The near-optimal scheduling obtained by HASS (after work stealing) in round i is recorded, and this information is used in round i+1. Thus, $HASS_p$ is an adaptive algorithm similar to $HASS_{ce}$ discussed before, i.e., in each round, it applies the partitioning information from the previous round(s) and as a result the balancing of loads improves from round to round. Figure 22 is the flowchart of the $HASS_p$ algorithm:

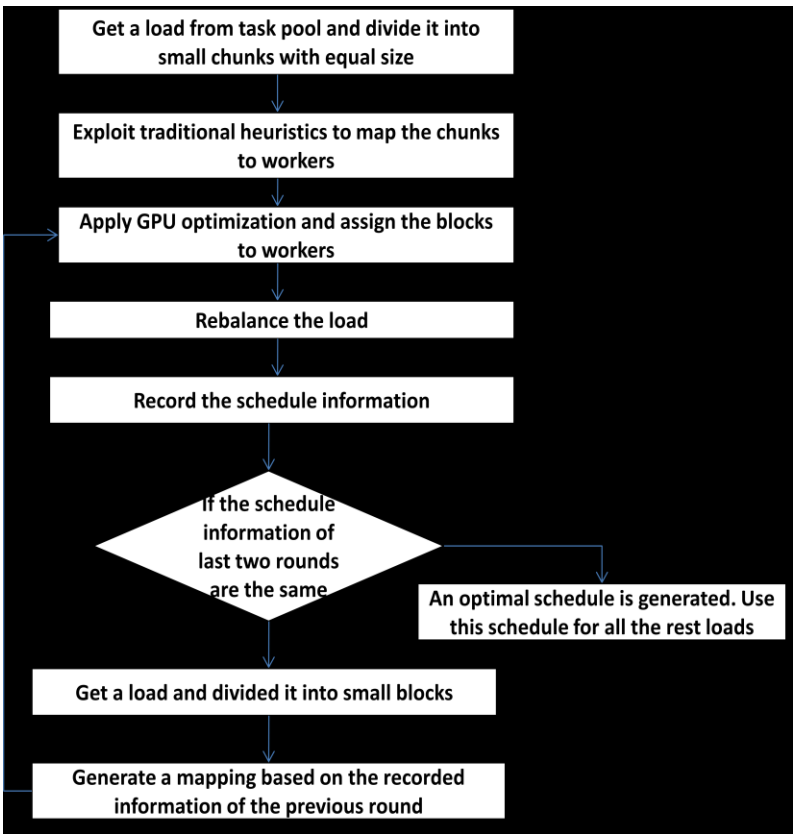

Figure 22: The flow chart of $HASS_p$

Algorithm 4: $HASS_p$
Input: A set of loads is generated by task generator and deposited to the task pool continuously.

/*The loads are to be executed on processors P = {p_c, $p_{g,1}$, $p_{g,2}$, ..., $p_{g,n}$}; p_c is the CPU and $p_{g,i}$ is a GPU*/

Output: Loads executed with minimized makespan.

Begin

1. Initialize variables i and j_k to 0;
   /*Here i is the proportion of the work that has to go to the CPU and j_k is the proportion of the work that has to go to GPU k*/
2. Extract a load from the task buffer, partition it into smaller blocks with size e and use L_q to denote these blocks
   /* $e = max\{sp_1, sp_2, \dots, sp_n\}$ where sp_i is the number of cores in GPU_i*/
3. Apply basic HASS to schedule loads in L_q.
4. Update i and j_k for each $p_{g,k}$ based on the scheduling in 3 above.
5. Go to the adaptation phase /*step 6 below*/.
6. {Adaptation phase}

while task generator is still generating loads

{

Extract a loads from the task buffer, and suppose its size is S.

if(i is unchanged compared with its previous value) /* An near-optimal schedule is generated*/

{

Assign a block with size S×i to p_c.

for all the GPU $p_{g,k}$

assign a block with size S× j_k to $p_{g,k}$

}

else

{

Partition the load into smaller blocks with size e and use L_q to denote the loads

for k =1 to n do

{

put q × j_k loads into GPU buffer $b_{g,k}$.

bundle all loads in $b_{g,k}$ and assign the single bundled load to $p_{g,k}$.

put q × i loads into CPU's buffer b_c.

}

while b_c is not empty do

{

if p_c is idle then remove a load d_i from b_c and assign to p_c.

for each GPU $p_{g,k}$

{

if $p_{g,k}$ is idle then

/*Steal work from CPU's buffer*/

remove a load d_m from b_c and assign it to $p_{g,k}$.

```
                    }
                    update i.
                    for each GPU p_{g,k}
                    update j_k.
                }
            }
        }
}
end
```

The *e* value in the algorithm can guarantee that all cores of each GPUs will be utilized when compute the smaller blocks. Therefore, the possibility of overestimating execution time of tasks by Min-min, which is described in the case of $\mathrm{HASS_{ce}}$, can be eliminated since the size of the smaller blocks are not "small"(i.e., the size of the blocks are greater than the amount of the cores of the GPUs)

All the algorithms presented before are for the cases when the estimated task size associated with a load is known a priori. If the size is unknown, we propose a modified greedy scheduling approach based on naive greedy scheduling. For naive greedy scheduling, once a processor becomes idle, it will request a task from the task pool. In contrast, in the modified greedy, we take advantage of task bundling to improve the performance: if a CPU worker is idling, it requests one load from the load pool; if the GPU workers are idling, they request multiple loads from the load pool. The reason why the GPU requests multiple loads is same as the "multi-stealing" in $\mathrm{HASS_{cd}}$.

3.5 A Different Approach for the Indivisible Load Farm

Compared with a divisible load which can be arbitrarily divided into independent smaller loads, an indivisible load has internal dependences among the tasks processing it and hence it cannot be partitioned arbitrarily. These internal dependencies in processing of an

indivisible load restrict its scheduling on contemporary GPUs that have a special thread distribution model.

3.5.1 The Thread Distribution of GPU

As discussed before, when programming with CUDA on GPUs, users need to map each data element of an input load to one CUDA thread, and the CUDA runtime environment will distribute the threads among the GPU cores. To be specific, the threads are first organized as blocks and the way to organize the threads must be indicated when the kernel function is called using two *specifiers*: the number of threads in one block, and the total number of blocks (therefore the total number of threads is the number of threads in one block multiplied by the total number of blocks).

During execution, the CUDA runtime will organize the threads into blocks as indicated and put them into a block pool. Then these blocks will be assigned to multiprocessors (i.e., SMs) in a greedy fashion: once a multiprocessor becomes idle, it will be assigned a block if the block pool is not empty. These multiprocessors are independent of each other and there is no synchronization mechanism among them. Hence the thread blocks are also independent of each other. This thread distribution model is ideal for divisible load scheduling because such loads can be arbitrarily divided and the blocks are totally independent. However in the case of indivisible loads, the threads might need to interact with one another (via shared memory) due to inherent dependencies and hence organizing all threads of an indivisible load into a single block is a necessity.

When employing the one-load-one-block strategy, it must be noticed that, no matter how many threads are contained in a block, it can only be executed by one multiprocessor while

all other multiprocessors remain idle. This is a huge wastage of the GPU's power. Therefore, instead of asking a GPU to compute one indivisible load at a time, *N* loads can be assigned to the GPU as a whole (*N* is the number of multiprocessors in the GPU), and each block is mapped to one multiprocessor. This can be implemented using code branches:

```
__global__ void kernel ()
{
    if(blockIdx.x = = 0)
        compute one indivisible load;

    else if (blockIdx.x = = 1)
        compute another indivisible load;
}
```

Suppose block 0 is assigned to multiprocessor *A* and block 1 is assigned to multiprocessor *B*, then *A* and *B* will compute two different indivisible loads concurrently. In such a case, if one of them finishes earlier, it must wait for the completion of other multiprocessors. If one multiprocessor is assigned a very large indivisible load and hence performs the computation for a very long time, all other multiprocessors which complete their executions earlier have to remain idle until this one finishes its computation, before another kernel can be invoked.

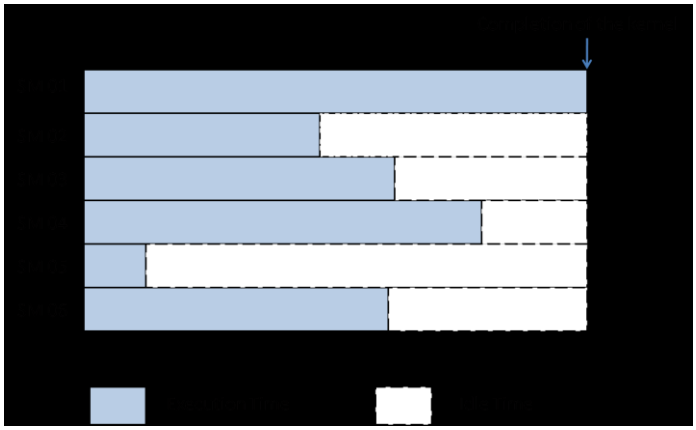

Figure 23: A Gantt Chart of the execution of multiprocessors on loads of different sizes

The previous discussion on the thread distribution model of contemporary GPUs is the basis for the following strategy on indivisible load scheduling.

3.5.2 $HASS_i$: A Strategy for Indivisible Load Farm Scheduling

For indivisible load scheduling in the CPU-GPUs environment, a greedy style strategy is proposed: once a processor becomes idle, it will be assigned loads if the load pool is not empty. While assigning loads to a GPU, *N* different indivisible loads with similar sizes are extracted from the pool and mapped to *N* CUDA blocks, where *N* is the number of multiprocessors in that GPU. This policy can guarantee that every multiprocessor can have a load to compute. The reason why the loads must have similar sizes is that they can assure the multiprocessors will finish computation almost simultaneously so that none of them has to wait. In order to achieve this, the loads need to be first sorted according to their sizes.

Furthermore, when picking up loads from the pool, the largest loads are chosen first for the GPUs. We call it the *Largest Load First (LLF) rule for GPU*. The reason behind this rule is

that during a time interval, the amount of small tasks computed by a GPU is more than the amount where the size is large.

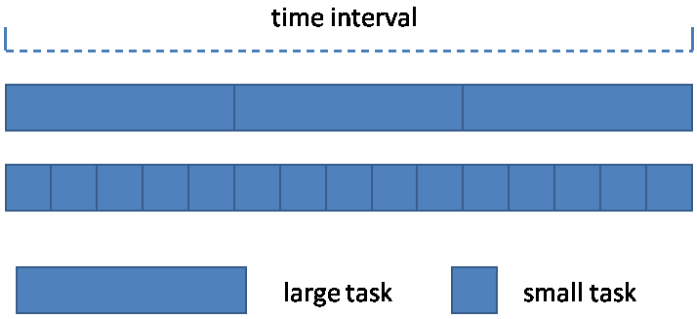

Figure 24: Comparison of the amount of large tasks computed by a GPU and the amount of small tasks during a same time interval

However, every task is associated with one message start-up latency and one kernel start-up latency and these overheads are quite significant, as discussed before. Compared with one large task computing, computing many small tasks will lead to more idle time of cores as the message start-up and kernel start-up will take more time, while at the same time, the smallest loads can be left for the CPU to compute, which is called the *Shortest Task First (STF) rule for CPU*.

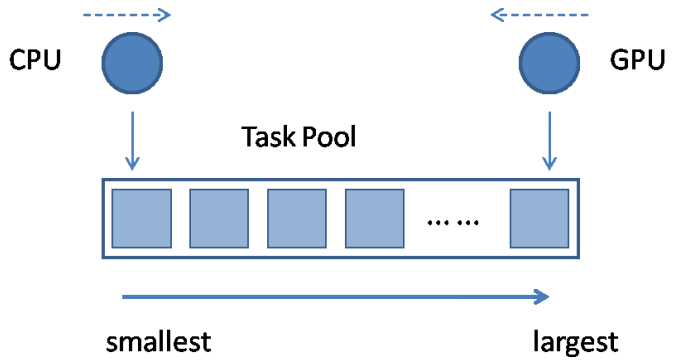

Figure 25: The LTF rule for GPU and STF rule for CPU

3.6 Function Farm

For function farm patterns, the tasks in the task pool are objects that consist of a function and an input. Workers are assigned the objects and they execute the functions using the inputs. The parallelism in this kind of farm pattern is essentially task-parallel. Unlike the data farm pattern where tasks are data items, tasks in function farm cannot be bundled or divided. Therefore, the strategies exploited for scheduling loads in the data farm pattern are not applicable to this pattern.

3.6.1 Parallelism of a Function in the Function Farm

The parallelism of a function is its potential to be computed in parallel by *a multi-core processor* so as to yield output in a shorter time. Usually the type of this parallelism is data-parallel. For example, as shown in the following, the function paraFunc has such a parallelism.

```
void paraFunc(int* aInt, unsigned int size)
{
    int a = 10;
    for(int i = 0; i< size; i++)
    {
        aInt[i] += a;
    }
}
```

In paraFunc, the inputs are an array of integers and the array's size. By calling this function, each element of the array is added by an integer value 10. It is obvious that this function can be computed in parallel, by dividing the array into blocks and assigning the blocks to the cores of a parallel processor. We use the term parallel function to refer to the function that can be computed in parallel. The reason why we focus on the parallelism

of a function is that, the objects in the task pool of the function farm pattern may contain parallel functions, and such functions are likely to have shorter execution times on the GPUs than on the CPU in the CPU-GPGPUs environment, since the GPU has much more cores than the CPU. Considering the importance of parallelism of a function for scheduling in CPU-GPGPUs environment, we attach an attribute: *parallelism degree (PD)* to each object in the task pool of the task farm pattern. PD reflects the maximum number of threads (workers) needed to execute the function. Take `paraFunc` as an example: assuming that the value of input size is 1000, so it can be computed by at most 1000 threads, each of which is in charge of one element of the array. Therefore, the value of PD for this function based on this input is 1000.

3.6.2 Revisiting the GPGPU Computing

As shown in Figure 2, a GPGPU in our model is comprised of a set of streaming multiprocessor (SM) and a streaming multiprocessor consists of multiple streaming processors (SP). The multiprocessors carry out executions independently of each other, while the SPs in one multiprocessor perform computation in a SIMD fashion. It is possible to ask each multiprocessor to run different control flows concurrently using certain techniques such as code branches. It should be noticed that in our system model, a GPGPU does not support concurrent kernels. Therefore, the approaches invoking several kernels and assigning different kernels to multiple multiprocessors is not feasible in our work. Moreover, it is also impossible to ask the SPs in one multiprocessor to run different control flow concurrently using code branches. This is because in the SIMD execution mode of the SPs, the execution of code branches will be serialized: when one SP executes

one code branch, the SPs which are assigned other branches must stay idling until that one finishes.

3.6.3 An Approach for Scheduling Tasks in the Function Farm Pattern

In this section we propose a heuristic $HASS_f$ to schedule tasks in function farm on the CPU-GPGPUs environment. For the function farm pattern, the tasks are distinguished in terms of their PDs. In the task pool, tasks with a PD value equal to one (i.e., to execute the task, only one thread will be spawn) are placed into one set, and the tasks with PD value greater than one (i.e., to execute the task, multiple threads will be spawn) are put into another set. For the convenience of description, we refer the tasks whose PD values are equal to one as *SPD (Simplex PD) tasks* and the tasks whose PD values are greater than one as *MPD (Multiplex PD) tasks*, and also name the set of SPD tasks as *SPD set* and the set of MPD tasks as *MPD set.*

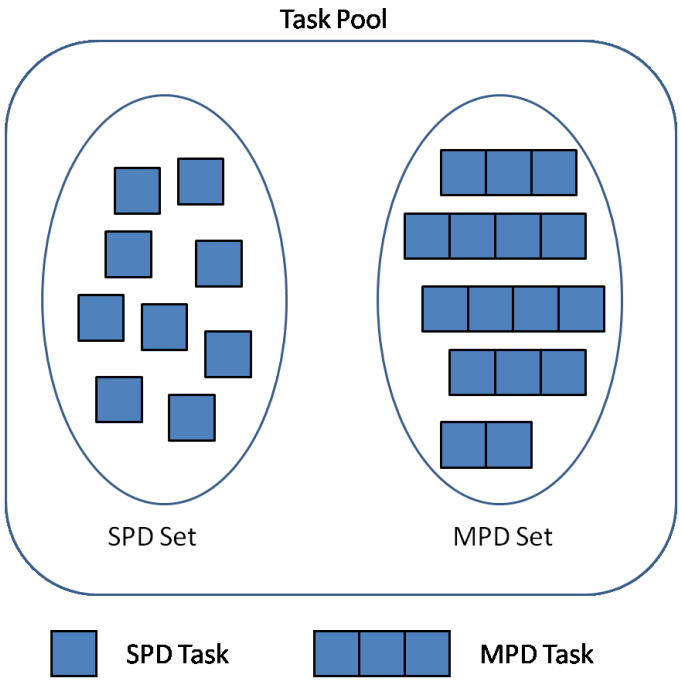

Figure 26: The classification of tasks in the task pool of the function farm

As mentioned in chapter 2, the GPUs are suitable for data-parallel tasks. In our assumption, the execution mode of the GPU is dedicated, which means that one task will monopolize a GPU while others must wait even if some cores of the GPU are idle due to underutilization caused by the task. Therefore, the tasks in the MPD set are more suitable for GPUs to compute than those in the SPD set, considering that the former will invoke multiple GPU threads while the latter can only use one thread, which leads to a severe underutilization of GPU's power.

The scheduling of this pattern exploits the greedy strategy: once a processor becomes idle it will request tasks from the task pool. Several rules are set to regulate the scheduling for the purpose of minimizing the total execution time.

To assign tasks to the GPU workers, the MPD set is first examined. If the set is not empty, a task is selected based on the *Largest Task First (LTF)* rule (refer to section 3.5.2) and assigned to the requesting processor. The reason is the same as in the scheduling of indivisible loads.

If the MPD set is empty, the GPU workers will select tasks from the SPD set. Suppose the number of multiprocessors of a GPU worker is *M*. Then *M* tasks with similar execution times are chosen, and are assigned to the processor in a way that one task is mapped to one multiprocessor by using code branches as discussed before. Since the tasks have similar execution times, all the multiprocessors have minimum idling times based on the previous discussion. When selecting tasks from the SPD set for the GPU workers, the *LTF* rule is employed again for the same reason.

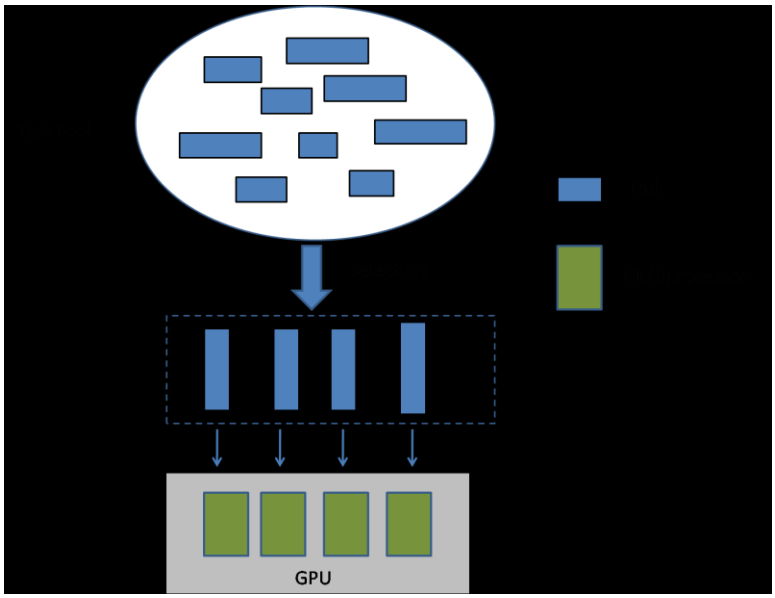

Figure 27: The selection of tasks for a GPU in the function farm

When assigning tasks to the CPU, the SPD set is first examined. If the SPD set is not empty, then a task is chosen and assigned to the CPU. To choose a task, the *Shortest Task*

First (STF) rule (section 3.5.2) is applied for the same reason. If the SPD set is empty and the MPD set is not empty, then a task is chosen from the MPD set by following the *STF* rule and is then assigned to the CPU.

3.6.4 An Implementation of the Dataflow Pattern as a Function Farm Pattern

As discussed below, a dataflow pattern can be implemented as a function farm and hence the same scheduling strategies of a function farm pattern can be applied to a dataflow pattern.

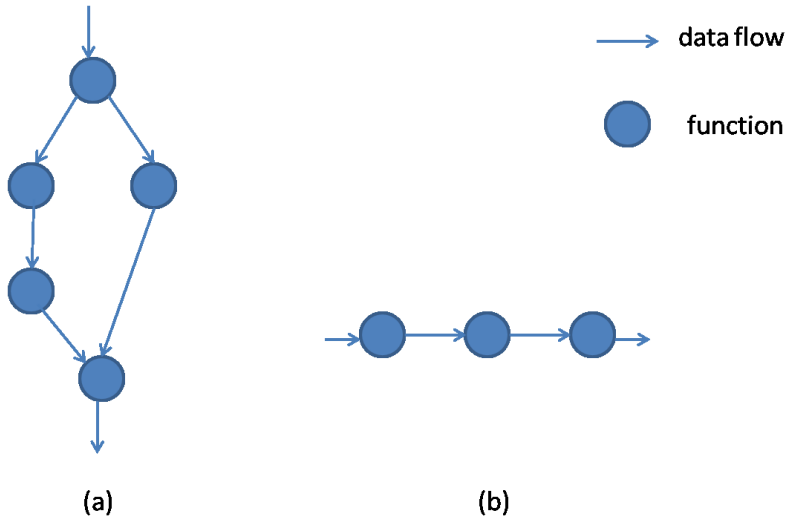

Figure 28: Two examples of dataflow pattern applications

In the data flow pattern, the participants are a set of functions with precedence constraints. Each function receives input dataflows generated by its predecessors, performs computation on these data, and sends the results to its successors.

For a dataflow application, when a function node generates a result and sends it to its successor function, then this successor can start the computation based on this input. This conduct can be viewed as the predecessor function "generating" the successor function,

since the successor function can be called once the predecessor functions finish. All of these functions can be regarded as tasks in the task pool of the function farm pattern. Therefore, the behaviour of this dataflow pattern can be represented from the perspective of function farm: the tasks in the task pool are the functions that have obtained input and are ready to be called. Once a function is executed by a worker, its successor function will be generated and put back to the task pool. Since the dataflow pattern can be represented by the function farm pattern, the scheduling strategy designed for function farm can be implicitly applied to the dataflow pattern.

A pipeline (Figure 28(b)) is a special type of the linear data flow pattern that is widely used in parallel applications. The same strategies can be employed to implement a dynamically scheduled pipeline, i.e., a pipeline whose stages are dynamically scheduled to processors.

Chapter 4 The Performance Evaluation of HASS

In the section, we discuss the performance results comparing HASS with some of the contemporary heuristics. In that regard, we implement a simple string searching application over a large text file to benchmark our algorithms. The application finds out all strings in the test file which conform to certain patterns. The system configuration is: Intel Xeon E5540 processor with 4 cores; 6 gigabytes of main memory; nVidia Tesla C1060 GPU which has 30 Stream Multiprocessors (SM) and each of them has 8 stream processors (SP) (i.e., totally 240 SPs); nVidia Quadro FX1800 GPU which has 8 SMs and each of them has 8 SPs (i.e., totally 64 SPs); and Windows 7 64-bit. The large input search-space can be divided into smaller sub-search-spaces, and these sub-search-spaces can be assigned to different processors to be searched concurrently and independent of one another. So this search problem falls into divisible load category of data farm pattern. In the following discussion, the term task is used to indicate a search over a subset of strings from the text file and the term load is used to indicate a sub-search-space.

Figure 29 shows a comparison among four approaches for scheduling loads of equal sizes for the above application. Min-min is chosen as a comparison subject based on the experimental results presented in [27], where Min-min gives better performance than most of other independent task scheduling heuristics. When implementing Min-min, we use the sizes of the tasks to estimate their execution times: we assume that the execution time is proportional to the size. In order to yield the estimated execution time, we choose a load, perform the computation on each processor, and record the execution time of this load. Subsequently, we compare the size of other loads with this one and estimate their

execution time based on the comparison results. However, as discussed before, when the size of a task is too small, the execution time may not be proportional to the load size in the case of GPU computing. Therefore, in order to eliminate this anomaly of small loads, the loads chosen for these experiments have the size that is not less than the number of SPs of the GPU with most SPs. In our experiment environment, the GPU having most SPs is Tesla C1060 with 240 SPs. Therefore, the sizes of the loads in our experiment are all more than 240. To be more specific, each load contains more than 240 strings.

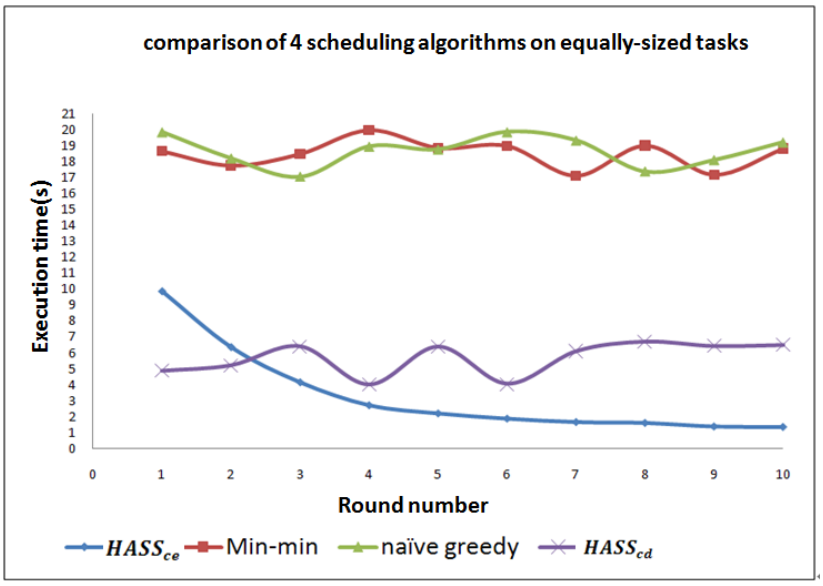

Figure 29: Comparison of four scheduling algorithms on tasks of equal sizes

In the above experiment, the same amount of load (a set of strings) is scheduled to the system in each round, and four different strategies are applied. The above experiment result demonstrates that $\mathrm{HASS_{ce}}$ is an adaptive heuristic since this approach can exploit the scheduling result from the last round to adapt its scheduling decision in this round. Therefore, for $\mathrm{HASS_{ce}}$, the scheduling result is better and better as the round number increases. In contrast, $\mathrm{HASS_{cd}}$ is not an adaptive approach, and its performance is not as

good as $HASS_{ce}$. Besides, in this experiment, the reason why Min-min and the naïve greedy have similar result is that Min-min has to yield a schedule on the fly before the loads are assigned to the processors, while the naïve greedy does not have this latency. In this experiment, the total makespan is the combination of the execution times of all rounds. By minimizing the execution of each round, a minimum makespan can be generated. Therefore, among the four approaches, $HASS_{ce}$ would produce the minimum makespan according to the experiment results.

Figure 30 shows a comparison among four approaches for scheduling tasks with unequal sizes for the search application, which means when reading data from the text file to the main memory, the data blocks with the different size are created.

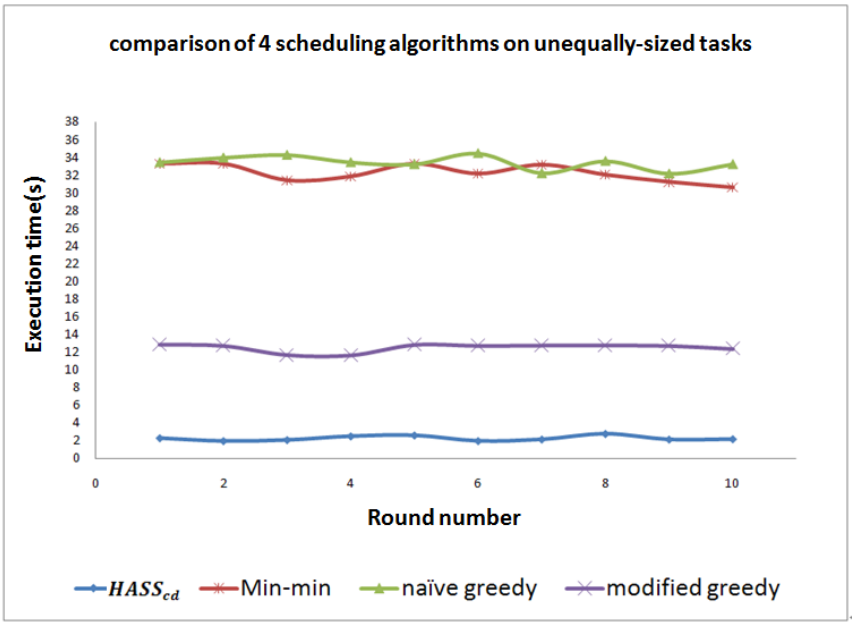

Figure 30: Comparison of four scheduling algorithms on tasks of unequal sizes

The above figure shows that $HASS_{cd}$ outperforms other three approaches for unequally-sized tasks, and modified greedy works better than the naïve greedy and

Min-min. The reason why Min-min and the naïve have the similar results is same as the experiment shown in Figure 29.

Figure 31 shows the results of comparing $HASS_p$ with Qilin approach in a single-CPU-single-GPU configuration. The system configuration is the same as before: Intel Xeon E5540; 6 gigabyte main memory; nVidia Tesla C1060 GPU; and Windows 7 64-bit. The size of each load is 240, which is the number of SPs in the GPU. In each round, 500 loads are scheduled to the system by using HASS and Qilin respectively. During the time interval between round 1 and round 7, Qilin is performing a training phase to yield a near-optimal schedule. Starting from round 8, the execution times yielded by HASS and Qilin are approximately equal to each other. As demonstrated in [30], Qilin can generate approximately optimal schedules for one-CPU-one-GPU system. Therefore, HASS can also yield a near-optimal schedule, as discussed at chapter 3.

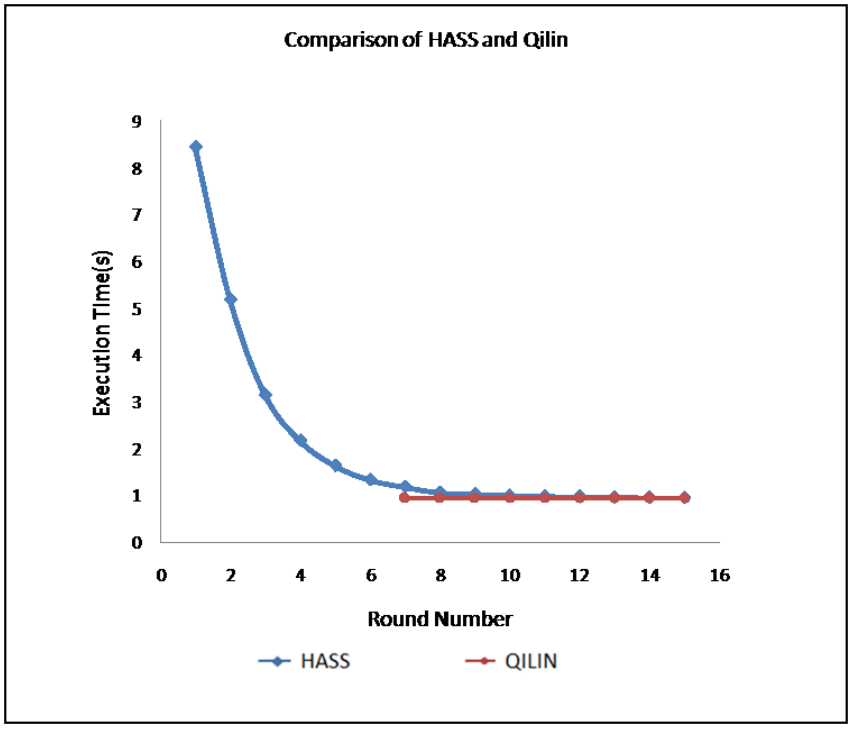

Figure 31: Comparison of $HASS_p$ and QILIN

The Figure 32 shows the distribution of tasks among three processors by using $HASS_{ce}$ in the experiment shown in Figure 26: one CPU and two GPUs. In the first round, 53% tasks are assigned to the CPU, 32.2% tasks are assigned to GPU 1 and 14.8% tasks are assigned to GPU 2. As the round number increases, the distribution among the three processors keeps changing and the execution time keeps decreasing, which implies that the load is more and more "balanced".

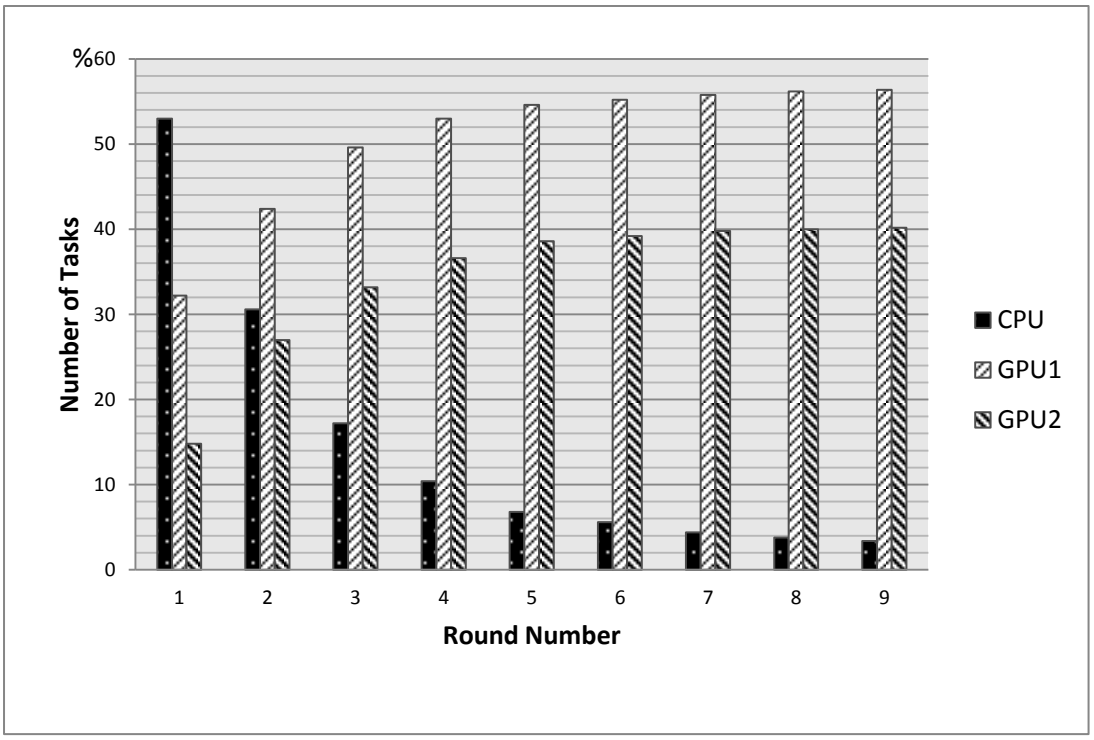

Figure 32: The distribution of tasks among the three processors when using the HASS algorithm

For the unknown-sized task scheduling category, the modified greedy heuristic (Chapter 3) is found to give a smaller makespan as compared to the naive greedy heuristic. Figure 33 presents the experimental results in applying both the modified and the naive greedy heuristics to schedule an identical set of tasks. Each task works on a search space to find a target pattern. However, in contrast to the previous application, in this case a task stops once the target is located in the sub-search-space. Therefore the execution time is not

proportional to the size of the search space but depends on the location of the target in the search space, and hence the time cannot be predicted a priori. As a result, heuristics like Min-min and HASS cannot be applied in this case and a greedy approach is the only feasible way to proceed. Compared with the naive greedy heuristic, the modified greedy heuristic takes advantage of load bundling which helps reduce the makespan significantly.

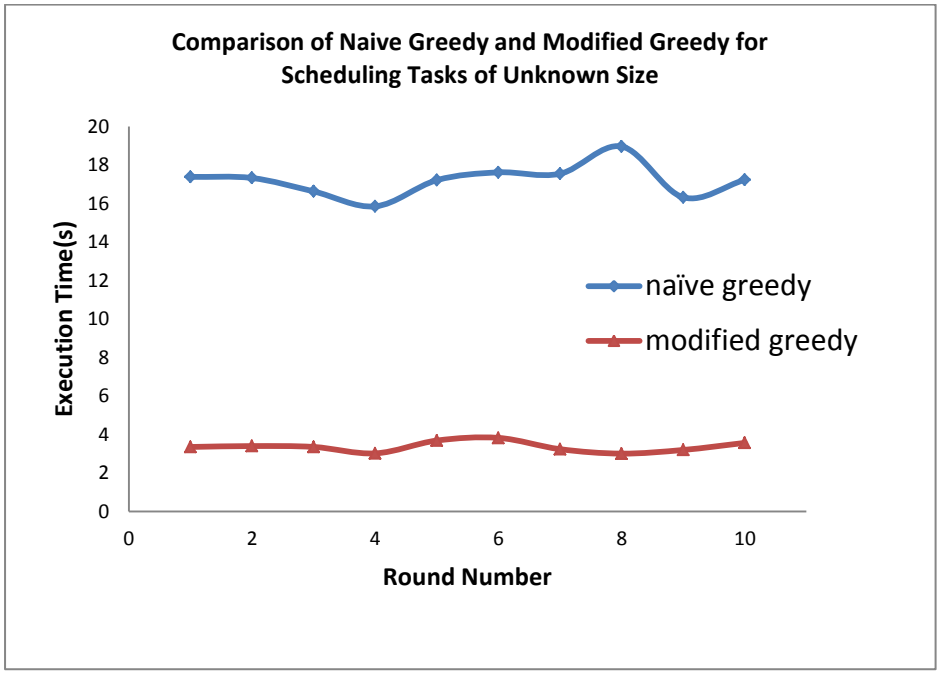

Figure 33: A comparison of naive greedy and modified greedy

According to the above benchmarks, HASS is found to perform better as compared to min-min and the naïve greedy strategies for the farm patterns with known load sizes. For a single-CPU-single-GPU environment, HASS gives approximately the same performance as Qilin, but Qilin in its current form cannot be applied to multiple-GPU environments. For the unknown size category, the greedy strategy with load bundling is found to perform better in comparison with a naïve greedy strategy.

Besides the experiments presented above, other two experiments: matrix multiplication and calculation of π using Monte Carlo simulation [38] are also conducted to verify the performance of the scheduling strategies. These experiments yield the similar results as those in the above experiments. The results also demonstrate that our strategies promise a good performance for farm pattern applications on the CPU-GPU architecture.

Chapter 5 An Implementation of the Scheduling Framework

This chapter presents an implementation of the scheduling framework. This framework is designed for CPU-GPUs environment. It takes advantage of the scheduling strategies discussed in chapter 3. The framework provides pre-implemented building blocks to users to help them develop parallel applications on CPU-GPUs efficiently. The current application interfaces are for the farm pattern(s) discussed in the previous chapters.

5.1 The Application Interfaces

The scheduling framework is implemented using C++. The object-oriented techniques of generic programming are also applied in the implementation. We implement the following four interfaces for different categories of farm pattern.

```
template<typename T>
void dataFarmPT(void (*cpuWorker)(T*), void (*gpuWorker)(T*), void
(*generator)(vector<T>* ), void (*partitioner)(T), void (*bundler)(vector<T>), void*
resultPool);
```

(a)

```
template<typename T>
void dataFarmCB(void (*cpuWorker)(T*), void (*gpuWorker(T*)), void
(*generator)(vector<T>* ), void (*bundler)(vector<T>), void* resultPool, enum farmKind
kind);
```

(b)

```
Template<typename T>
void dataFarmIL(void (*cpuWorker)(T*), void (*gpuWorker(T*)), void
(*generator)(vector<T>* ), void* resultPool);
```

(c)

```
void FunctionFarm(void (*generator)(vector<Wrapper*>* ), void* resultPool);
```

(d)

Figure 34: The interfaces for different categories of farm pattern

Function dataFarmPT (Figure 34(a)) is for the partition case of data farm pattern. The interface is implemented using C++ template; hence it is a generic function and can be instantiated by users. As shown in the prototype of the function, users must provide two implementations of workers: one is for the CPU and another is for the GPUs. This is because the programming models of the CPU and the GPU are different, so a universal implementation of worker functions for both CPU and GPU is impossible. When implementing CPU and GPU worker functions, programmers need to use of some existing development tools such as TBB or OpenMP for the CPU and CUDA for the GPU. Another parameter of function dataFarmPT is a task generator and its input parameter is the task pool which is a container accepting tasks produced by the generator. The task pool is a component of the build blocks and it is implemented using *vector* from STL (Standard Template Library) [31] of C++. The next two parameters of dataFarmPT are *partitioner* function and *bundler* function. Partitioner is used to partition a task into small blocks and bundler is for task bundling. These two methods are provided by users to the interface as function pointers. The last parameter is the *resultPool* which is used to return the computational results to users.

Function dataFarmCB (Figure 34(b)) is for the combination case of data farm pattern. It is similar to dataFarmPT except for two parameters. It does not have a partitioner, since the tasks are quite small and are unnecessary to be partitioned. This function requires another parameter: *kind,* which is of enumerated type and is used to indicate the type of the farm: the sizes of the tasks are the same, the sizes are distinct, or the sizes are unknown. *kind* is one of *Equal*, *Distinct*, or *Unknown*. The value of *kind* decides the scheduling strategy that is going to be chosen.

Function dataFarmIL (Figure 34(c)) is for indivisible load farm. Since in this type of farm, the loads cannot be partitioned or bundled arbitrarily, the partitioner and the bundler are unnecessary. Hence the input parameters of this function are a CPU worker function, a GPU worker function, the task generator and the result pool.

Function functionFarm (Figure 34(d)) is for the function farm pattern. Since in this pattern, the tasks are objects that encompass functions and the relevant input data, it is impossible to partition them. Therefore neither a partitioner nor a bundler is needed. The only two parameters are the task pool that contains the objects and the result pool that return the computation results to users. It should be noted that the type of tasks in the task pool is pointers pointing to an object *Wrapper*. *Wrapper* is an abstract base class that must be inherited by the user-defined task classes. This Wrapper class contains one virtual member function: callFunction(), and two attributes: *exeTime* and *PD*. callFunction() works as an adapter wrapping the function given by programmers. When this function is called, it will subsequently call the functional member function with the input specified by users. This adapter function allows the framework to designate one worker to execute a function (i.e., a task in this pattern) abstractly without knowing the specific function name and parameters. The attributes *exeTime* and *PD* are used for yielding a scheduling strategy and they are indicated by users. The scheduler of the framework will read the value of these two attributes and exploit them to make scheduling decisions as discussed in chapter 3.

5.2 Components of the Scheduling Framework Applications

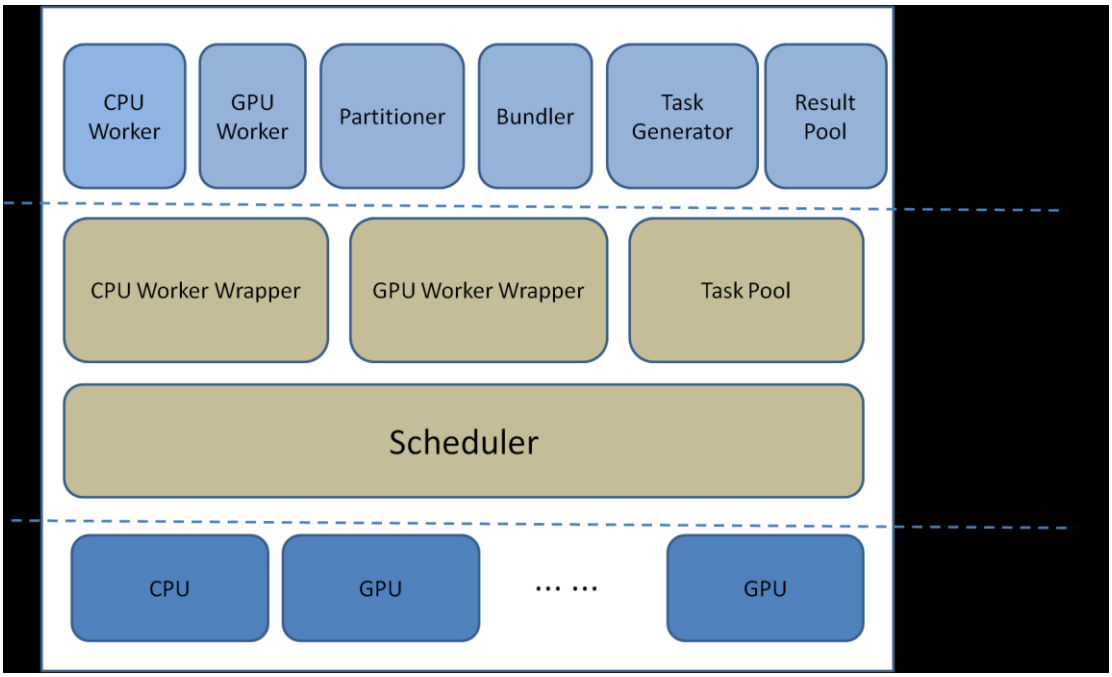

Figure 35: The components of an application developed using dataFarmPT

An application developed using dataFarmPT is comprised of nine components that are classified into two parts: one is the application-specific part provided by programmers, the other is the application-independent part provided by the skeleton. The components from the former part include: CPU worker, GPU worker, Partitioner, Bundler, Load Generator and Result pool. Their functionalities of them have been discussed in the last section. The second group of components are hidden from users by the skeleton includes: CPU Worker Wrapper, GPU Worker Wrapper, Load Pool and Scheduler. CPU Worker Wrapper and GPU Worker Wrapper are used to map the CPU worker function and GPU worker function to CPU processor and GPU processors. The reason behind these wrappers is that, even if GPU worker is a function mainly performing computation on GPU, it is called by CPU and all relevant works, except computing the input tasks, such

as initializing GPU environment, transferring data and calling kernel, are carried out on the CPU, which means that for the execution of GPU worker function, it still requires some execution time on CPU. Therefore, mapping CPU worker wrappers and GPU worker wrappers to different CPU threads can improve the performance. Another component of the skeleton is the Load Pool. It contains loads produced by the generator. The last component, Scheduler, is the key constituent that implements the scheduling approaches discussed in the last chapter. The functionalities of this component are to retrieve loads from the load pool, make scheduling decisions based on certain criterions discussed before, and then assign the tasks to the CPU and the GPUs.

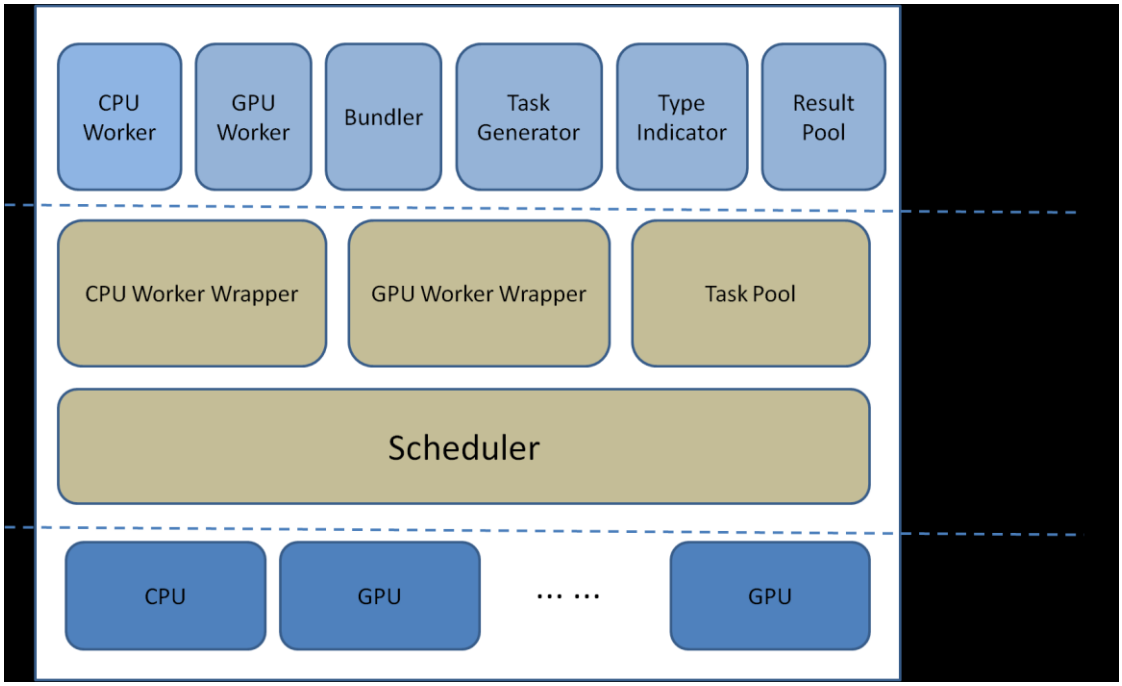

Figure 36: The components of an application developed using dataFarmCB

An application implemented using the dataFarmCB has a similar structure as the previous one. The difference is that there is no Partitioner, but another component—Type Indicator. Type Indicator is used to indicate the type of the pattern for the Scheduler. For different

types (e.g., tasks with equally-size, and tasks with unequally-size), Scheduler will apply different strategies as discussed in a previous chapter.

The structure of applications derived from functionFarm is relatively simple. Only four components exist in the application: Task Generator, Result Pool, Task Pool and Scheduler. The functionalities of these components are similar to the previous cases and hence are not elaborated.

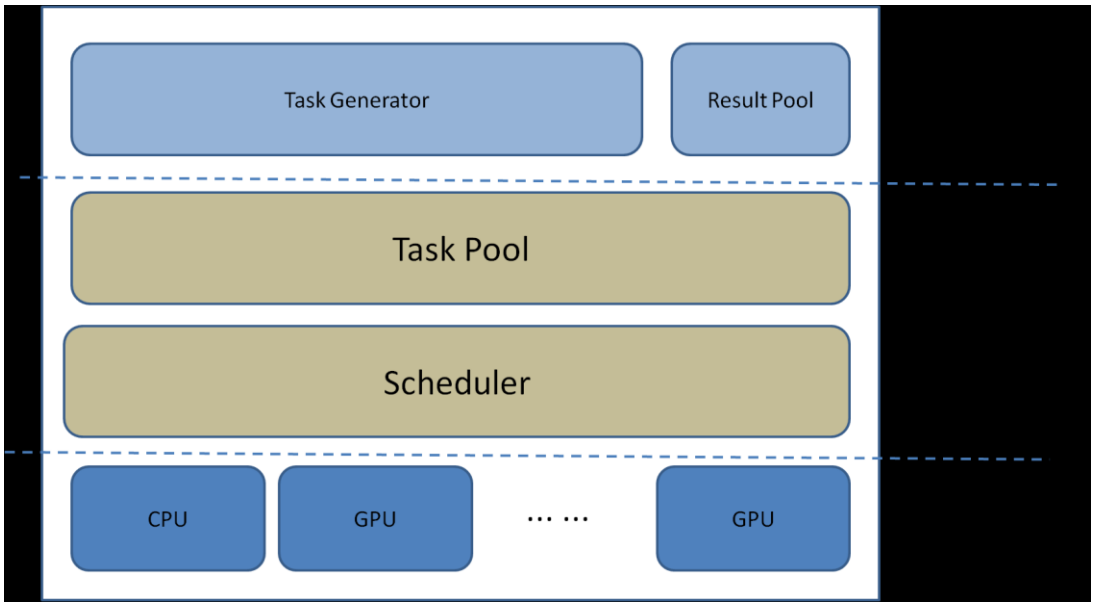

Figure 37: The components of an application developed using functionFarm

An application developed using the dataFarmIL has the similar structure as the one presented in Figure 37.

5.3 An Example of Developing Applications using the Scheduling Framework

Considering the following problem: the input is a large database of personal information, and the requirement is to find out all persons whose birthday is later than 1965 and the salary is above $1000 as quickly as possible. This problem can be solved using the farm pattern: a set of data records are retrieved from the database and put into load pool, and then are divided into different blocks (i.e., a sub-set of data records) and assigned to CPU and the GPUs for searching.

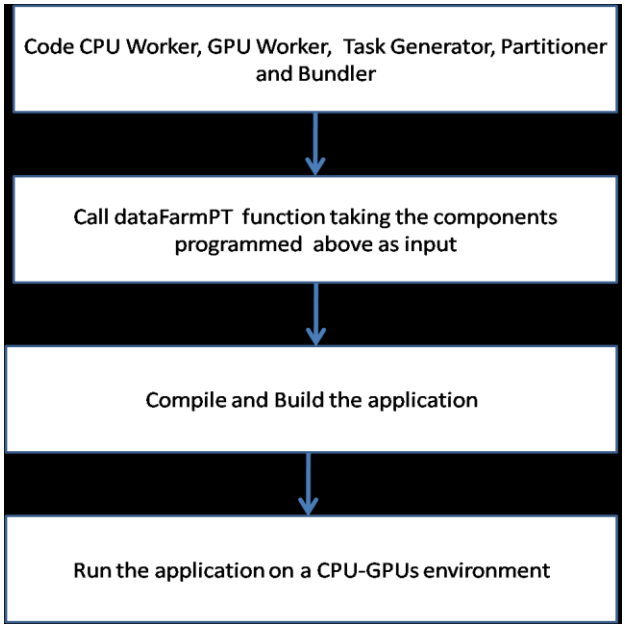

Figure 38: The flow of developing an application using the framework

The above figure shows the development flow using the framework. Users firstly build several necessary components and then call the interface functions by using these components as input parameters. Subsequently, the program can be compiled and run on any CPU-GPGPUs systems.

During the execution, Task Generator is responsible for reading data items from the database and put them into the Task Pool. The amount of data items is dependent on the size of the main memory. After the reading finishes, Scheduler employs the approaches presented in chapter 3 to divide the input task and assign different blocks to CPU and GPUs.

```
#include “farmPattern.h”
#include <vector>
using namespace std;

//the data structure to store the record reading from the database
struct PersonalInfo
{
    int birthday;
    int salary;
    // other information;
}



// definition of CPU worker
void cpuWorker(vector<PersonalInfo> task)
{
    unsigned int size = task.size();
    for(int i = 0; i < size; i++)
    {
        if(task[i].birthday>1965 && task[i].salary > 1000)
            //put task[i] into the result buffer;
}
}

//the kernel function that performs the searching using GPU
__global__ void kernel(PersonalInfo* personalInfo, PersonalInfo* results, int*
resultSize)
{
    Int index = threadIdx.x+blockIdx.x*blockDim.x;
    if(*(personalInfo+index).birthday>1965 && *(personalInfo+index).salary>1000)
    {
        *(results+index).birthday = *(personalInfo+index).birthday;
        *(results+index).salary = *(personalInfo+index).salary;
```

```
            //copy all other information from personalInfo to results;
            atomInc(resultSize); // increase resultSize atomically to avoid race condition
        }
}

//definition of GPU worker
void gpuWorker(vector<PersonalInfo> task)
{
        unsigned int taskSize = task.size()*sizeof(PersonalInfo));
        PersonalInfo* h_task = (PersonalInfo*)malloc(taskSize);
//copy all data from vector to a C-like array, since CUDA can only copy array
        for(int i = 0; i<task.size(); i++)
        {
h_task[i] = task[i];
}
        PersonalInfo* d_personalInfo;
        PersonalInfor* results;
        //allocate the memory
        cudaMalloc(&d_personalInfo, taskSize);
        cudaMalloc(&results, taskSize);
        cudaMemcpy(d_personalInfo, h_task, taskSize, cudaMemcpyHostToDevice);
        int resultSize;
        kernel<<<256, task.size/256+1>>>(d_personalInfo, results, &resultSize);
        int copySize = resultSize*sizeof(PersonalInfo);
        PersonalInfo* h_results = (PersonalInfo*)malloc(copySize*sizeof(PersonalInfo));
cudaMemcpy(h_results, results, copySize, cudaMemcpyHostToDevice);

//copy the results from the array to the result vector
for(int i = 0; i<copySize; i++)
            resultVec[i] = h_result[i];

// release the memory
cudaFree(d_personalInfo);
cudaFree(results);
free(h_results);
}

void partitioner(vector<PersonalInfo> task, int& chunkNum,
vector< vector<PersonalInfo>* >& smallBlockBuffer)
{
        int size = task.size();
        chunkNum = size/M; // M is the number of SPs of the GPU with maximum SPs
            for(int i=0; i<chunkNum; i++)
              smallBlockBuffer.push(&task + i*M);
```

```
}

void bundler(vector<PersonalInfo>* bundledTask,
vector< vector<PersonalInfo>* >& smallBlockBuffer)
{
    //bundle all the small data blocks together
}

void taskGenerator(vector<PersonalInfo>* taskPool)
{
    // read a group of personal information from the database;
}
int main()
{
    vector<PersonalInfo> taskPool;
    vector<PersonalInfor> resultPool;
    dataFarmPT(cpuWorker, gpuWorker, taskGenerator(&taskPool), partitioner,
bundler, &resultPool);
}
```

As shown in the above code segment, by using the scheduling framework, users only need to focus on the programming of the functional part of the application. Once the functional part is implemented, users can call the APIs and provides the implementation as input parameters to the framework which can handle all other application-independent issues such as spawning and termination of threads, task distribution and dynamic load balancing.

The development process shown above demonstrates that the scheduling framework can significantly simplify the construction of parallel applications on CPU-GPGPUs platform, because the programmer is liberated from the complex systems-specific issues of the underlying platform.

Chapter 6 Conclusion and Future Work

Compared with sequential application development, parallel application development has to deal with additional and complex issues such as creation and termination of processes, mapping processes to processors, communication and synchronization among the processes, and so forth. In parallel programming, besides the programming-related issues, a key issue is task scheduling. In the past several decades, several algorithms [23][24][25][26][27] were proposed to solve different kinds of scheduling problems, e.g., divisible load scheduling [22], independent task scheduling [9] and DAG scheduling [8]. Most of these algorithms are designed for CPU clusters and Grid. In recent years, a newly-arising parallel system, the CPU-GPGPUs platform which has some different features from the CPU clusters and the grid, has been intensively studied. The scheduling problem on this platform is the focus of this thesis.

In order to design scheduling strategies for the CPU-GPGPUs platform, we first explore its architecture and compare it with some other traditional parallel computing systems such as cluster, grid, and multi-core CPU. This architecture is a combination of message-passing and shared-memory models: the association of CPU and GPUs yields a distributed memory system and they communicate with each other through messages. Since CPUs and GPUs are multi- and many-core processors respectively, each of them is also a shared-memory parallel system. Furthermore, the GPUs for general-purpose computing are quite different from multi-core CPUs in several aspects, including the architecture, the programming model (i.e., MIMD versus SIMD) and the execution model.

Another focus of this research is parallel patterns [13]. Parallel patterns are used to describe recurring problems in the parallel computing field and to abstract the common characteristics of a group of problems. Traditional uses of patterns are to facilitate in application design and development. However, as is one focus of this research, these common characteristics can also be used to guide the task scheduling of a parallel program. In this thesis, we thoroughly study one classic pattern—the farm pattern. We first classify the farm pattern and for each category we propose heuristic(s) to solve its scheduling problem.

The farm pattern can be classified into two broad categories: data farm and function farm. Two variations of data farm includes: divisible load farm and indivisible load farm. Divisible load farm further has three variations: the one with equal size tasks, the one with distinct size tasks and the one with unknown size tasks. Another taxonomy of divisible load farm is based on the scale of the task size: if the task size is fairly large, then the scheduling of tasks in this farm falls into Divisible Load Scheduling (DLS) problems. If the task size is comparatively small, the farm will employ Independent Task Scheduling strategies.

Based on the classifications mentioned above, we propose a set of scheduling strategies (HASS: Heterogeneous Architecture Scheduling Strategies). $HASS_p$ is the heuristic for divisible load farm with equal task size and unequal task size using divisible load scheduling model; $HASS_{ce}$ uses independent task scheduling model to solve divisible load farm with equal task size; and $HASS_{cd}$ is for the farm whose tasks are of distinct sizes. $HASS_i$ is for the task scheduling of indivisible load farm and $HASS_f$ is for

function farm. The major differences between these strategies and the traditional approaches are that the former make use of the features of the underlying CPU-GPGPUs system, based on classifications of the pattern(s), into consideration while assigning tasks to workers, which contribute towards improved performance.

The same scheduling strategies for the farm pattern can be extended to some other patterns, e.g., data-flow and pipeline.

As the scheduling strategies have been developed, we further implement a scheduling framework, based on the classified farm patterns, on the CPU-GPGPUs system. The framework hides some complexities arising from parallel application development. It abstracts the application-independent parts which contain the structural information and parameterizes the application-specific functional components. The applications developed using the framework is comprised of two parts: A functionality part provided by users, and another part that is implemented in the framework, encompasses the pattern-related information such as the participants and their relations, and the scheduling strategies. The evaluation results of the framework are also presented using a simple file searching application.

Since we only focused on the farm pattern in this research, as a future work, we intend to move on to other patterns and address their scheduling strategies. We need to analyze all variations of each pattern since current experiences show that different variations can have different scheduling strategies.

Bibliography

[1]. PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/.

[2]. MPI. http://www.mcs.anl.gov/research/projects/mpi/.

[3]. Intel Thread Building Blocks. http://threadingbuildingblocks.org/.

[4]. CUDA architecture. http://www.nvidia.com/object/cuda_home_new.html/.

[5]. General-Purpose Computation on Graphics Hardware. http://gpgpu.org/.

[6]. OpenMP Architecture. http://openmp.org/wp/.

[7]. J. Ullman. NP-complete scheduling problems. Journal of Computer and System Sciences, 10: 384-393, 1975.

[8]. H. El-Rewini, T.G. Lewis, and H. H. Ali. Task Scheduling in Parallel and Distributed Systems. Prentice-Hall, Inc. Upper Saddle River, NJ, NY, USA, 1994.

[9]. W.W. Chu, L.J. Holloway, L. Min-Tsung, and K. Efe. Task allocation in distributed data processing. Computer, 13(11): 57-69, 1980.

[10]. R.D. Blumofe, C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundation of Computer Science*. November 20-November 22, 1994. Santa Fe, FM, USA.

[11]. A. Aiken, A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and implementation*, July 1988, New York, NY, USA

[12]. PCI and PCI Express. http://en.wikipedia.org/wiki/PCI_Local_Bus. http://en.wikipedia.org/wiki/PCI_Express.

[13]. B. L. Massinggill, T. G. Mattson and B. A. Sanders. Patterns for Parallel Application Programs. In *PLoP Conference*, 1999.

[14]. T.G. Mattson, B.A. Sanders, B. L. Massingill. Patterns for Parallel Programming 1st edition. Addison-Wesley Professional, Sep 25 2004

[15]. M.I. Cole. Algorithmic Skeletons: structured management of parallel computation. MIT Press, Cambridge, MA, USA, 1989

[16]. D. Goswami. Parallel Architectural Skeletons: Re-Usable Building Blocks in Parallel Applications. PhD thesis, Department of Electrical and Computer Engineering, University of Waterloo, 2001

[17]. M. Cole. Bring skeletons out of closet: a pragmatic manifesto for skeletal parallel programming. Parallel computing, 30(3): 389-406, 2004

[18]. D. Caromel, L. Henrio, and M. Leyton. Type safe algorithmic skeletons. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-based Processing*, page 45-53, Toulouse, France, Feb. 2008. IEEE CS Press.

[19]. D. Goswami, A. Singh, and B. R. Preiss. From design patterns to parallel architectural skeletons. J. Parallel Distrib. Comput., 62(4): 669-695, 2002

[20]. H. Gonzalez-Velez, M. Cole. Adaptive structured parallelism for distributed heterogeneous architectures: a methodological approach with pipelines and farms. Concurrency and Computation: Practice and Experience, 22: 2073-2094, 2010

[21]. A. Grama, G. Karypis, V. Kumar, A. Gupta. Introduction to Parallel Computing 2 edition. Addison Wesley, Feb 17 2003

[22]. V. Bharadwaj, D. Ghose, and T.G. Robertazzi. Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems. Cluster Computing, 6: 7-17, 2003

[23]. O.H. Ibarra and C.E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. J. Assoc. Comput. Mach. 24, 2: 280-289, Apr. 1977.

[24]. D. Yu and T. G. Robertazzi. Divisible Load Scheduling for Grid Computing. In *Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, 1: 1-6, Marina del Ray, CA; USA, 2003.

[25]. O. Beaumont, H. Casanova, A. Legrand, Y. Robert and Y. Yang. Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems. IEEE Transaction on Parallel and Distributed Systems. 16: 207-218. 2005.

[26]. Y. Yang, K. Raadt and H. Casanova. Multiround Algorithms for Scheduling Divisible Loads. IEEE Transaction on Parallel and Distributed Systems. 16: 1092-1102. 2005.

[27]. T. D. Braun, H. J. Siegel and N. Beck. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. Journal of Parallel and Distributed Computing, 61: 810-837, 2001.

[28]. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen and R. F. Freund. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. Journal of Parallel and Distributed Computing, 59: 107-131, 1999.

[29]. Q. Hua, Z. Chen and C. M. Lau. A New Method for Independent Task Scheduling in Nonlinearly DAG Clustering. *In International Symposium on Parallel Architectures, Algorithms and Network*, Hong Kong, SAR, China, 2004.

[30]. C. K. Luk, S. Hong and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2009.

[31]. P.J. Plauger, A. Stepanov, M. Lee and D.R. Musser. The C++ Standard Template Library, 1st edition. Prentice Hall. 2001.

[32]. Virtual memory. http://en.wikipedia.org/wiki/Virtual_memory.

[33]. NVIDIA CUDA C Programming Guide version 3.2.

[34]. Intel Core i7-970 architecture. http://akensai.com/intel-i7-970.

[35]. M. Danelutto. Task Farm Computation in Java. High Performance Computing and Networking. Lecture notes in Computer Science, 1823:385-394, 2000.

[36]. M. Boyer, D. Tarjan, S.T. Acton, and K. Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *IEEE International Symposium on Parallel&Distributed Processing.* Rome, Italy, 2009.

[37]. G. Horacio, L. Mario. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software: Practice and Experience, 40(12): 1135-1160, 2010.

[38]. Berman Kenneth A, Paul Jerome L. Algorithms: Sequential, Parallel, and Distributed. Course Technology. Boston, USA, 2005.