COMPUTE LANGUAGE INTERFACE: A TRANSPARENT WRAPPER LIBRARY FOR MULTI CPU-GPU

by

OOI KENG SIANG

Thesis submitted in fulfilment of the requirements for the Degree of

Master of Sciences

March 2013

ACKNOWLEDGEMENTS

First of all, I wants to express my gratitude to my supervisor Dr. Chan Huah Yong for spending his precious time supervising my research work and providing me all the necessary advises and equipment for my research study. Thanks to School of Computer Sciences given me the well-equipped research environment to complete my research study.

I wants to give thanks to my parents for supporting my study ever since I was small until today. Thanks to my beloved who always supports and encourages me. Without their supports, I would not be able to make it so far.

I want to say thank you to all my friends and lab members whom help me go through all different kind of troubles and problems during my research study. Last but not least, thanks to the Ministry of Higher Education for offering me the MyMaster scholarship which helps me to pay off the school fee in USM.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API – Application Programming Interface

BrookGPU – Brook for GPU

Cg – C for Graphic

CLI – Compute Language Interface

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

DNA – Deoxyribonucleic acid

FLOPS – Floating Point Operations per Second

GFLOPS – Giga Floating Point Operations per Second

GLslang – OpenGL Shading Language

GPGPU – General-Purpose Computing on Graphics Processing Unit

GPU – Graphics Processing Unit

HLSL – High Level Shader

MPI – Message Passing Interface

OpenCL – Open Computing Language

OpenGL – Open Graphic Language

PCIe – Peripheral Component Interconnect Express

RPC – Remote Procedure Call

SIMD – Single Instruction Multiple Data

SSI – Single System Image

TCP/IP – Transmission Control Protocol and Internet Protocol

ANTARMUKA BAHASA PENGIRAAN: SATU SALUTAN PUSTAKA YANG TELUS UNTUK CPU-GPU BERBILANG

ABSTRAK

Keupayaan pemprosesan bagi Unit Pemprosesan Grafik (GPU) didapati lebih berkesan daripada sebelum ini. Aplikasi jenis data paralelisme dan intensif pengiraan terbukti lebih baik apabila dijalankan dalam GPU berbanding dengan Unit Pemprosesan Pusat (CPU). Namun begitu, rangka-rangka kerja pengaturcaraan bagi GPGPU yang ada masih tidak dapat menyokong CPU dan GPU terletak di komputer lain dalam sistem pengkomputeran teragih. Kajian ini mengemukakan CLI yang merupakan satu salutan pustaka yang membolehkan aplikasi OpenCL untuk mengakses secara telus bagi semua CPU dan GPU yang ada dalam sistem pengkomputeran teragih dengan MPI. CLI direka untuk meningkatkan kebolehskalaan aplikasi OpenCL dalam sistem pengkomputeran teragih di samping dapat mengekalkan set API yang sama dalam pusaka OpenCL asal. Aplikasi dapat menggunkan semua CPU dan GPU yang ada di dalam komputer yang berbeza dengan sistem pengkomputeran teragih seolah-olah semua CPU dan GPU terletak di dalam satu komputer yang sama. Ujikaji dalam kajian ini menunjukkan bahawa aplikasi yang menggunakan CLI dengan dua GPU dapat mempercepatkan masa pemprosesan keseluruhan sebanyak 44 peratus berbanding dengan satu GPU sahaja. Manakala aplikasi yang menggunakan satu CPU dan satu GPU dapat mempercepatkan masa pemprosesan keseluruhan sebanyak 51 peratus dengan overhed hanya 0.1 peratus tambahan berbanding dengan rangka kerja pengaturcaraan yang asal.

COMPUTE LANGUAGE INTERFACE: A TRANSPARENT WRAPPER LIBRARY FOR MULTI CPU-GPU

ABSTRACT

The Graphics Processing Unit (GPU) processing capability is getting more powerful than before. Compute intensive and data parallelism applications are proven to perform better on the GPU than on the Central Processing Unit (CPU). However, available General-Purpose Computing on Graphics Processing Unit (GPGPU) programming frameworks which are available publicly are unable to reach beyond the single computer limitation to utilize multiple CPUs and GPUs at different computers in a distributed computing system easily. This study presents the Compute Language Interface (CLI) which is a wrapper library that enables the existing OpenCL applications access to all available CPUs and GPUs in a distributed computing system through Message Passing Interface (MPI) transparently. It is designed to improve the scalability of the OpenCL applications on a distributed computing system while maintaining the same set of application programming interface (API) in the original OpenCL library. The applications can access all available CPUs and GPUs in different computers in a distributed computing system as if all the CPUs and GPUs are in the same computer. One of the experiments shows that the application running on two GPUs using the CLI can reduce the overall processing time by 44% if compared with one GPU. While the application running on one CPU and one GPU can reduce the overall processing time by 51% with 0.1% increases of overhead if compared with the native programming framework.

CHAPTER 1

INTRODUCTION

1.1 Introduction of General-Purpose Computing on Graphics Processing Unit

Over recent years, graphics processing unit (GPU) has played a more significant role, rather than just accelerating the computational of graphics rendering from central processing unit (CPU). Now GPU can use its massive floating-point computational power that is originally designed for graphics rendering computation to perform non-graphical computational (Buck, et al., 2004). The technique used to perform non-graphical computational on the GPU that is usually handled by the conventional CPU is called general-purpose computing on graphics processing unit (GPGPU) (Wu & Liu, 2008).

Driven by the demand in computer gaming and console gaming industry, the GPU technology has been growing faster and becoming cheaper. The GPU was even proven to outperform the CPU in compute intensive and data parallelism application (Buck, et al., 2004; Fan, et al., 2008; Lawlor, 2009). In year 2011, the latest high-end workstation GPU like NVIDIA Tesla C2070 and AMD FireStream 9370 are capable of performing over 515 Giga floating point operations per second (GFLOPS) (NVIDIA, 2010) and 528 GFLOPS (AMD, 2011) in double precision calculations. On the other hand, high-end workstation CPU like the Intel Xeon Processor X7560 is only capable of performing over 82 GFLOPS in double precision calculations (Intel, 2011). This is because CPU is designed to execute common workloads, such as basic

arithmetical, logical and input-output operations of the system. In contrast, GPU is only designed to process large amounts of independent data in parallel with its massive number of scalar processors (Aoki, et al., 2010).

Although the GPU is a powerful yet low cost resource, adoption of GPU in massive distributed computing or scientific computing is still relatively low (Owens, et al., 2007). This is because there are only limited number of programming frameworks available to program on GPU currently and many of them are unable to scale beyond a single computer as they are designed for a single computer environment (Aoki, et al., 2010). Complicated architecture design and implementation involving different frameworks are required to develop a GPGPU application that is capable to execute on multiple computers that span across the network (Fan, et al., 2008; Lawlor, 2009; Moerschell & Owens, 2008). The problem becomes even worse, when the computation involves heterogeneous computing resources from various types of hardware and vendors.

The GPU indeed is very powerful for executing compute intensive and data parallelism applications and it is very suitable to be used as an alternative computational resources in massive distributed computing or scientific computing. The major problem with the GPU currently is that the programming frameworks are not designed for massive distributed computing because the framework cannot support processors or resources beyond a single computer (Aoki, et al., 2010). This research targets to solve the problems using the Compute Language Interface (CLI) which is a transparent wrapper library for GPGPU applications. It allows applications execute transparency on multiple GPUs and CPUs located in multiple different computers in the same network. The objectives are to improve the

performance and load balancing among all available resources, while keeping the development of GPGPU applications as simple as possible to avoid a steep learning curve.

1.2 Background of the Problem

The GPU is indeed a powerful and low cost computing power for scientific computing or computing that requires massive computing power (Owens, et al., 2007). Presently, the CPU remains as the most popular choice in the distributed computing even though the GPU was already proven to outperform the CPU in compute intensive calculation. This is because the programming frameworks available for the CPU are widely used and much simpler if compared with programming frameworks for the GPU (Lee, et al., 2009). Development of GPGPU applications on the distributed computing systems remains a tedious task for programmers, as they need to solve all the problems and challenges before the application can takes the advantages of all GPU processing power on all computers in the distributed computing systems.

A programming framework or application framework is a set of reusable class library, application programming interface (API) and structure that can be used by other applications (Fayad, 2000). The number of public available programming frameworks that can support GPGPU are currently very limited. The programming framework selection is further narrow down because each of the available programming framework has its own limitations (Owens, et al., 2007). The most widely used CUDA programming framework is designed to support only NVIDIA's GPUs (Aoki, et al., 2010), while DirectCompute for the GPGPU development which

is part of the popular DirectX programming framework is designed to support only in the Microsoft Windows platform (Wu & Liu, 2008). Although the OpenCL is designed to solve the problem of vendor and platform limitations by enabling the application to utilize CPUs, GPUs, Cell Processors and other parallel processors without tightly having to specify hardware or vendors, currently all hardware vendors are implementing their own versions of OpenCL library that only supports their own products.

Since a single computer can only have a few Peripheral Component Interconnect Express (PCIe) slots for GPUs; therefore, GPU resources are usually limited to a maximum of four GPUs. This because most existing GPGPU programming frameworks only support a single computer environment and do not support multiple computers environment across the networks. To scale beyond the limitation of PCIe slots on a single computer, programming frameworks or protocols like Message Passing Interface (MPI), Remote Procedure Call (RPC) or other networks programming frameworks are required to integrate with GPGPU programming framework to enable communication among computers (Aoki, et al., 2010; Lawlor, 2009). The complicated frameworks and the high difficulty of integrating multiple programming frameworks have caused the low adoption of GPU in massive distributed computing or scientific computing (Fan, et al., 2008; Lawlor, 2009). Many still prefer CPU over GPU because the CPU programming frameworks or libraries are much widely used and easier to implement if compared with the GPU in an environment with many computers.

To simplify the implementation of a GPGPU application which is capable to execute on multiple computers environment, many research projects choose to let

GPU handles all computational tasks, while let CPU handles all communication tasks among computers (Lawlor, 2009). This is because the GPU is unable to access directly to other devices such as network cards. The GPU has many advantages over the CPU in executing compute intensive and data parallelism applications, but the CPU computing power should not be ignored (Fan, et al., 2008) especially when current CPUs can easily scale up to 6 cores on a single CPU chip and a single computer can have multiple sockets for multiple CPUs. The CPU has also been proven as powerful as the GPU when appropriate optimization is applied (Lee, et al., 2010). The computational power and the functionalities of the CPU, other than just performing communication tasks should be taken into consideration. This helps utilizing all available resources more effectively while providing a better performance in the distributed computing systems.

Although the idea of utilizing all available resources like CPUs and GPUs to speed up the computational time is very simple, optimizing applications and distribute the workloads to different types of processors can be very challenging (Aoki, et al., 2010). This is because some low-end CPUs and GPUs have limited processing power and memory to handle any extra workload. Some low-end CPUs and GPUs might even slow down the overall processing time. Furthermore, the distributed computing systems are commonly built using different models and types of GPUs and CPUs. This will further increases the challenges in optimizing applications running on them. Multiple test runs are required to optimize a GPGPU application currently because different GPU models perform differently. In addition, the performance of GPU is not directly proportional to the processor's frequency or number of scalar processor like CPU. This makes optimizing application designed for CPUs and GPUs become more challenging.

1.3 Research Problem

This thesis concentrates on the challenges faced by integrating GPUs and CPUs that span across multiple computers to achieve better scalability and performance results as discussed in section 1.2. The main problem in this research is "How to enhance existing programming framework to support CPU and GPU that span across multiple computers?"

The sub problems of this research are as follows:

- How to improve the scalability of existing programming framework on GPGPU and CPU?
- How to enhance the existing programming frameworks to enables applications access to all available processors transparently?
- How to further optimize an application running on many different types of device be without involving a major modification?

1.4 Justification of Research Problem

Even though the powerful computational power of GPU might be the answer for the ever increasing demand of compute intensive applications, the existing GPGPU programming frameworks are designed for a single computer environment only (Aoki, et al., 2010). They are still not powerful enough for applications like computer graphic rendering applications that are normally saw in the movies or scientific simulation applications which require large amount of computing powers.

Computational power of GPGPU needs to be scaled beyond the limit of a single computer. Implementation difficulty and lack of frameworks to support the GPGPU in the distributed computing environment has caused the low adoption of GPU in the distributed computing systems. A better framework that enables GPGPU applications to run on multiple computers environment is thus required. Nevertheless, the framework should be as simple as possible.

Although some research projects propose frameworks that allow the GPGPU to scale across multiple computers on the distributing computing systems, those frameworks do not fully explore all available resources like CPUs (Owens, et al., 2007). This happen because the GPU's processing power is far more powerful than the CPU. While at the same time CPU is required to handle extra tasks like communication among computers which does not involve GPU at all (Lawlor, 2009; Fan, et al., 2008). This might be true for computers with a single CPU, as it will be overloaded by the communication and operating system tasks. However, for a computer with two or more CPUs, it will be a waste of resources if the other CPUs processing power is left out during the computational time. Two or more CPUs built in a workstation computer or server are very common now and applications need to utilize the available resources more effectively. Since there is a huge difference between the CPU and the GPU in terms of architectural design and processing capabilities, a more in-depth study of the ratio of workloads between CPUs and GPUs is required.

1.5 Objectives

The main objective of this research is to improve existing programming frameworks on GPGPU and CPU to gain the following advantages:

- To scale beyond the limit of a single computer and executes on one or more CPUs or GPUs at the same time.
- To enable applications transparently access to different CPUs and GPUs in different computers.
- To provide a better way of controlling and optimizing application running on GPUs and CPUs in a distributed computing system.

1.6 Significance of the Research

Driven by the ever increasing demand for more powerful processing power by compute intensive applications such as simulation application and gaming (Owens, et al., 2007; Fan, et al., 2008), the GPU's high computational power is indeed an attractive resource that can be used as an alternative resource other than the CPU. Different types of programming frameworks have been introduced to ease the developer tasks in GPU programming, but GPU cluster or multiple computers environment involving communication among GPUs on different computers remains a big problem and challenge (Fan, et al., 2008). Developers need a better programming framework that supports GPGPU execution on multiple computers environment without major modification in the existing applications.

Consumer level GPU that supports OpenCL or CUDA are commonly found in laptops or computers nowadays. Grid computing systems that build using many

different computers including the consumer level of computer are require a simple yet powerful programming framework or library to enable applications to utilize the GPU resources. Application executing on different models and vendors of GPUs and CPUs might become a big problem because some low-end CPUs or GPUs might slow down overall processing time. Applications need a better way to utilize all available resources, yet a simple way to optimize the applications to adapt to different hardware environments.

1.7 Scope of the Research

The scope of this research covers the following:

- Review the existing programming frameworks in order to propose an improved library wrapper that enables applications to execute on GPU or other parallel processors like CPU that span across multiple computers on a network.

- Primary focus is on improving the performance and scalability of CPUs-GPUs in the distributed computing system; others like security, reliability and load balancing will not be discussed in detail.

- Analyse the workloads ratio for GPU and CPU when combining both different types of processors for computing.

1.8 Research Methodology

Figure 1.1 shows a preview of research procedures from analysing existing programming frameworks until the design and testing of the proposed framework. While Figure 1.2 shows the focusing research area in this study and the proposed solution.

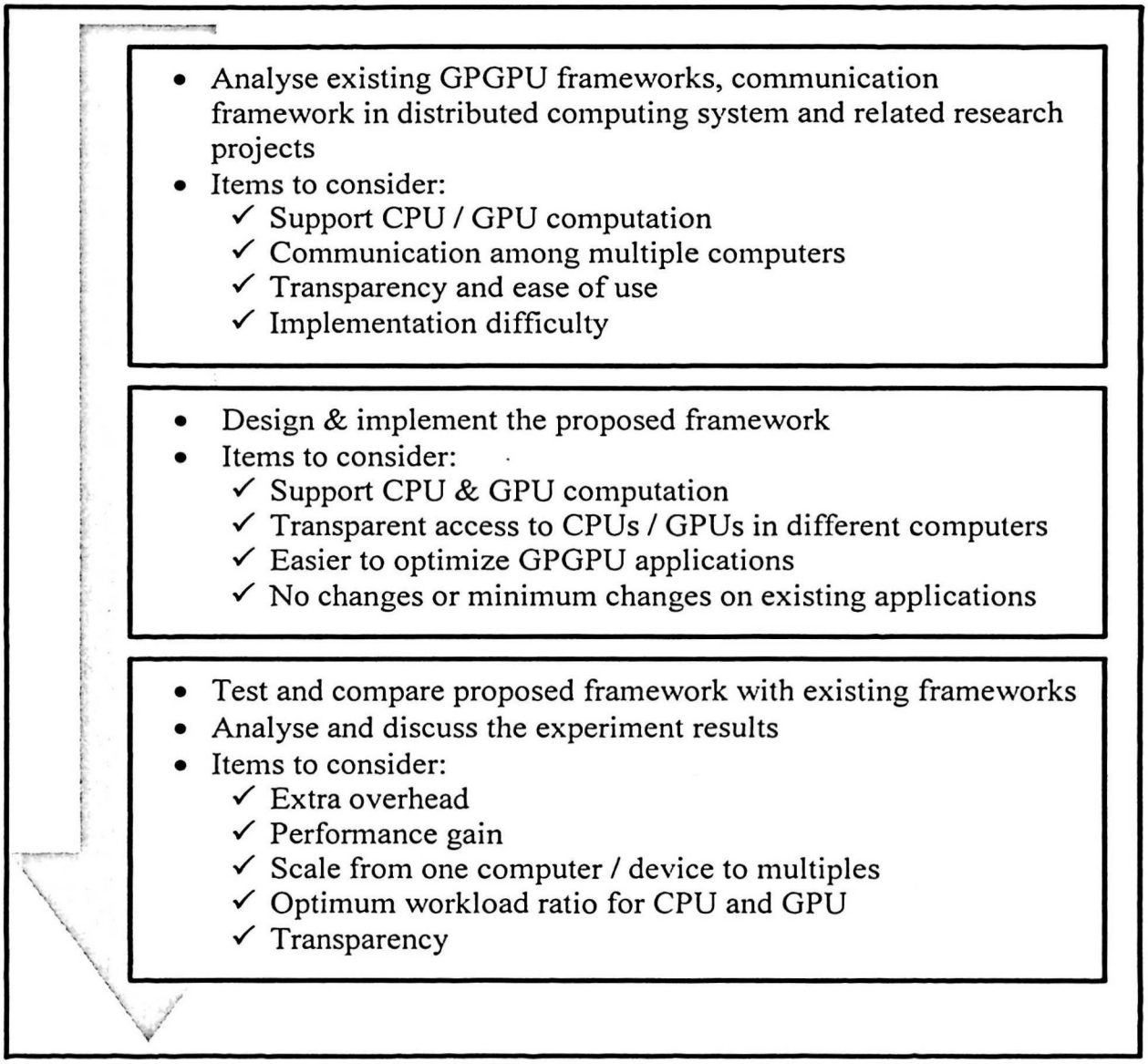

Figure 1.1: Research procedures

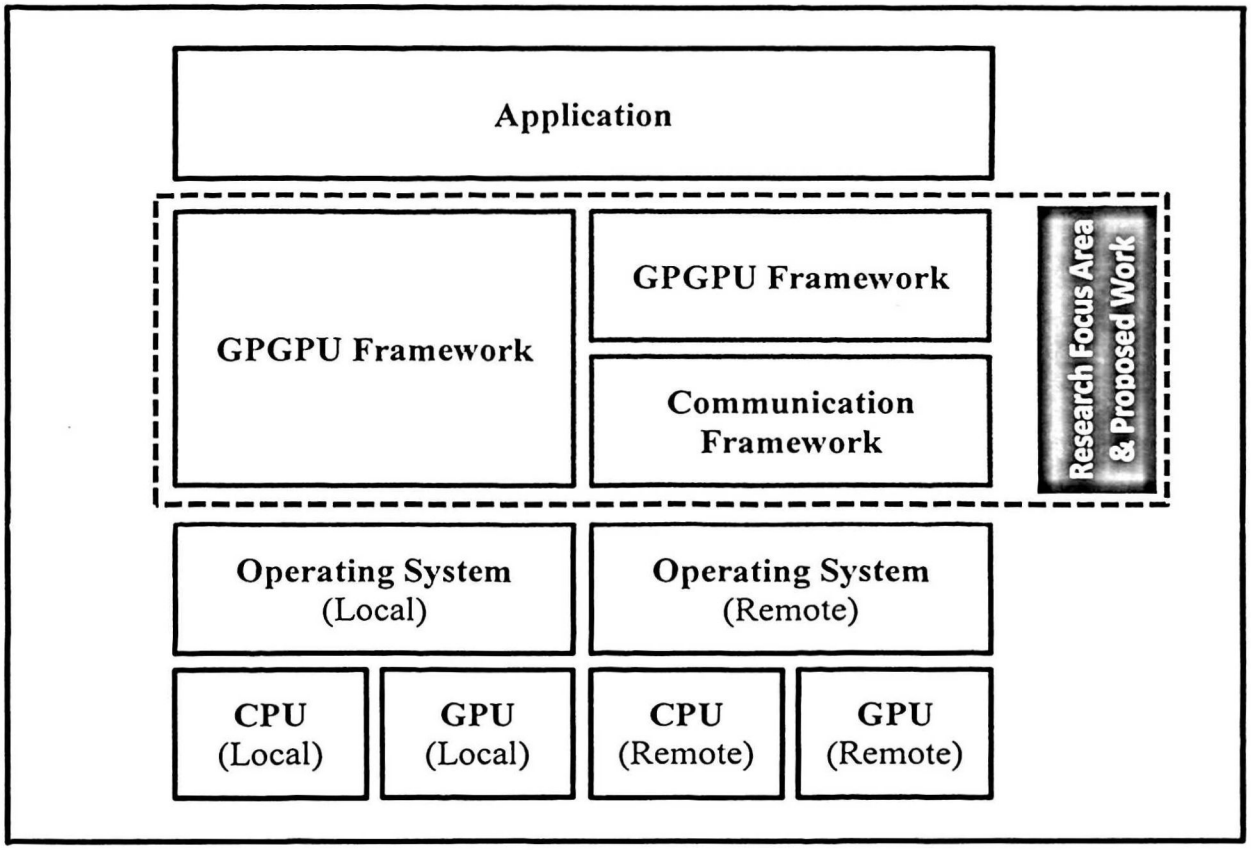

Figure 1.2: Research focus area

1.9 Contribution

At the end of this research, this study produces the following contributions:

- A wrapper library for GPGPU and CPU that enables applications to scale beyond a single computer in a distributed computing system.
- Simplify the enhanced programming framework to allow the applications to transparently access all available GPUs and CPUs in a distributed computing system.
- A better way to control and optimize applications running in the environment with many different types of devices without complicated changes.

1.10 Thesis Organisation

Chapter 2 reviews the existing GPGPU and distributed computing technologies and research projects. The chapter is divide into several sections to discuss each technique in detail. This includes GPGPU, distributed computing, GPGPU in distributed computing systems and combining CPU-GPU in distributed computing systems. All related works are also included at the end of this chapter.

Chapter 3 covers the proposed framework design in this research. This chapter begins with simple flow of research that explains each phase of the research. This is then followed by a discussion of the proposed theoretical framework. The theoretical framework is further supported with justification of the research problems, research design, research limitations and assumptions.

Chapter 4 explains in detail the technologies, techniques and methods selected to achieve higher scalability, better resources utilization and simpler framework for GPGPU in the distributed computing systems. The implementation of the proposed framework and prototype is explained throughout the whole chapter.

Chapter 5 contains the experimental methods and results of this research. The experiments covers the performance of existing programming frameworks and the programming framework proposed by this research in various scenarios. The results are further analysed and discussed in this chapter.

The last chapter, Chapter 6 concludes the findings and outcomes of this research. The chapter also includes the suggestions for future works that could be done related to this research.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Other than CPU, the GPU is a very powerful and cheap computational resource, which can be used in the distributed computing systems. Examples of applications that can take advantages on those powerful computational resources are Deoxyribonucleic acid (DNA) sequencing simulation, molecular modelling, weather forecasting, video processing, 3-dimension computer graphics rendering and many more. One of the problems is that the cost to develop a GPGPU application currently is very high. This is because developers face a lot of different challenges in developing a GPGPU application with the existing GPGPU programming frameworks.

This chapter explores and discusses in depth of the GPGPU programming frameworks evolution from the past until today. Protocols or programming frameworks used for communication among computers in the distributed computing systems are also included in the study to provide a foundation for the selection of technologies and technique to be used in this research. Other research projects concerning the GPGPU and the CPU-GPU used in distributed computing systems are discuss in detail in the following section. Related works about the GPGPU execution model and the distributed computing flow are also included. At the end of this chapter, the suitable techniques or frameworks are selected to integrate into the proposed frameworks.

2.2 General-Purpose Computing on Graphics Processing Unit (GPGPU)

A GPU is built with many scalar processors that enables it to execute a single instruction on multiple data at the same time. The powerful processing power of GPU had attracted the attention of programmers since 2002 to perform non-graphical computational tasks. Before 2004, writing a GPGPU application was a tedious task because hardware and programming frameworks to support the GPGPU were very limited. Programmers were required to write the programming model into multiple graphic fragments or vertex shaders manually using languages or library like C for Graphics (Cg), High Level Shader Language (HLSL) or OpenGL Shading Language (GLslang) so that it can be processed by the GPU. With the limitations in graphic programming framework, expressing algorithms in shading language can be extraordinarily hard (Buck, et al., 2004).

Buck et al. (2004) proposed the very first public available GPGPU framework, Brook for GPU (BrookGPU) (BrookGPU, 2007). BrookGPU was designed as a high level programming language to address the problem of matching algorithm using the low level graphic programming interface and graphic shader. The framework cuts the arduous work needed to understand the shading languages and the underlying hardware architecture (Buck, et al., 2004). The development of BrookGPU has been discontinued since year 2007, but BrookGPU has brought the GPGPU development to a new level where programmers no longer required any knowledge of graphic shader to program GPGPU.

NVIDIA further simplified the development of the GPGPU by introducing Compute Unified Device Architecture (CUDA), which is currently the most widely used proprietary programming framework in GPGPU development (Lawlor, 2009). CUDA provides programmers with a high level API for GPGPU with extended C programming language, which is already familiar among most programmers. This allows programmers to focus on writing efficient parallel applications without wasting their time to learn everything from ground zero (Garland, et al., 2008). Unlike BrookGPU, CUDA is still continuously updated and maintained by NVIDIA, but NVIDIA specially designed CUDA to support only NVIDIA's GPUs.

Besides CUDA, DirectCompute is an API developed by Microsoft for GPGPU development on the Windows platform. DirectCompute is the subset of APIs in DirectX proprietary programming framework that is included in DirectX since version 11 (Wu & Liu, 2008). The DirectX is the most popular framework used in the development of video games currently. The DirectCompute is designed to run on any DirectX 10 capable GPU, which is commonly found in the market today without any vendor limitation. However, it is limited to Windows platform that supports at least DirectX 10 such as the Windows Vista and Windows 7. Just like CUDA, DirectCompute can only run on GPUs.

To solve the vendor and platform limitations, Apple Inc. initiated Open Compute Language (OpenCL), which is an open royalty-free standard parallel programming for modern processors that includes CPU, GPU, Cell Processor and other parallel processors (Kim, et al., 2011). The OpenCL has received full support from hardware vendors such as AMD, NVIDIA, Intel and IBM. Although the OpenCL was initially designed for cross operating system and cross hardware vendor

programming framework, currently all hardware vendors are implementing their own versions of OpenCL API that only support their own products (Aoki, et al., 2010). It is possible to utilize CPUs and GPUs from different vendors, but it requires tedious work to make it happen. The OpenCL standard is currently maintained by Khronos Group. Table 2.1 shows the summary of all GPGPU frameworks that have been discussed.

Table 2.1: Comparison of GPGPU frameworks

	Brook for GPU	CUDA	OpenCL	DirectCompute
GPU Support	Yes	Yes	Yes	Yes
CPU Support	Yes	No	Yes	No
Supported GPU	NVIDIA GPU & AMD GPU	NVIDIA GPU Only	Depend on GPU Vendor	NVIDIA GPU & AMD GPU
Supported Platform	Cross Platforms	Cross Platforms	Cross Platforms	Windows Platform Only
Popularity	Low *(Project Discontinued)*	High	Medium	Low

2.3 Distributed Computing System

A distributed computing system is consists of multiple computers that are connected and communicated through computer networks. Each computer has its own local memory, which is not share with other computers. It is called distributed memory and all computers communicate through data transferring or message passing. The goal of a distributed computing system is to speed up the processing of a large problem by dividing the problem into many smaller problems that can be processed by many computers. Since the existing GPGPU programming frameworks

are designed for a single computer environment, distributed computing frameworks or protocols are required to integrate into GPGPU programming frameworks to provide the communication function among computers in the distributed computing systems.

Message Passing Interface (MPI) is a language independent communication protocol that is used to program parallel distributed computer. The simple concept of point to point communication and collective communication have made MPI as the de facto standard used in distributed memory systems like computer clusters or grid computing (Lawlor, 2009). Since the MPI runs on a distributed memory system, the programmer is required to keep track of the data locality and communication explicitly (Silcock & Goscinski, 1996). Besides, the MPI is originally designed and intended for CPU processing only and GPU is yet to be supported by any public available MPI programming framework (Fan, et al., 2008).

Remote Procedure Call (RPC) is an inter-process communication protocol that allows applications to execute procedure call on another remote computer just like executing procedure call on the local computer. RPC is able to abstract away the details of communication and data movement (Silcock & Goscinski, 1996). Unlike the MPI, the RPC reduces the burden of programmers by simplifying the procedure call without explicitly controlling the data movement and processes involved in the communication. The simplicity of RPC comes with a price, which is the performance penalty. While the MPI requires one message between processes communication, the RPC requires two messages to archive the same communication (Silcock & Goscinski, 1996).

Single System Image (SSI) is a cluster operating system that allows a group of computers to be viewed as a single computer. It allows applications to run on the system to access the resources in other computers as if they were on the same computer (Lottiaux, et al., 2005). Some SSI such as the OpenSSI or Kerrighed are built with a Distributed Shared Memory (DSM) which is a form of memory architecture where it groups all memories which are distributed on multiple computers in one single virtual address space (Moerschell & Owens, 2008). This allows applications to execute on a distributed computer system to access all memories just like a centralized computer system. Although the simplicity and high transparency of SSI make application programming on distributed computer systems very easy, they come with a great performance penalty cost. While the MPI requires one message between processes communication, the SSI requires at least four to 12 messages to archive the same communication through DSM (Silcock & Goscinski, 1996).

Table 2.2 shows the comparison of communication protocols and architectures commonly used in distributed computing systems. The MPI and the RPC are the most commonly used communication protocols in the distributed computing systems currently. Meanwhile, DSM in the SSI is less often used in the distributed computing systems due to its complicated implementation and performance drawback. Each of the protocol and memory architecture discussed in this section has its own advantages and disadvantages, which need to take into consideration during the implementation.

Table 2.2: Comparison of frameworks and protocols used in distributed computing system

	MPI	RPC	SSI
Performance	High	Medium	Low
Transparency	Low	Medium	High
Popularity	High	High	Low
Memory Architecture	Distributed	Distributed	Shared

2.4 GPGPU in Distributed Computing System

GPGPU in distributed computing systems is not something new. Some research projects have tried to integrate existing GPGPU programming frameworks with communication frameworks to create an environment for GPGPU in distributed computing systems. Fan et al. (2008) proposed Zippy, a framework that is scalable on the GPU computer clusters. Zippy integrated Open Graphics Library (OpenGL) and MPI to create a high performance and non-uniform memory access modal. Zippy hides the low level communication details and allows applications to move global data without specifying the source and destination of GPU. Unlike modern GPGPU frameworks such as CUDA or OpenCL, Zippy uses shading language which is known to be difficult and limited because it is designed for 3D graphics and not for general computational usage.

Lawlor (2009) proposed cudaMPI, an improved version of CUDA with MPI like message passing functionality for communication among GPUs. It is specially designed for GPU computer clusters. Although the programming framework provides all the basic MPI functionality required to implement an application on GPU clusters, developers still require CUDA and MPI knowledge to develop an application using this programming framework. The framework can helps

programmers in optimizing the communication call but the steep learning curve and complicated memory management problems still remain.

The latest version of CUDA, version 5 introduced GPUDirect, which is a new technology from NVIDIA and Mellanox that enables GPU-to-GPU communication without involving any CPU in the communication loop (Mellanox Technologies, 2010). It is specially designed to improve the performance and efficiency of GPU clusters. The GPUDirect requires the latest version of NVIDIA Tesla or Fermi GPUs and Mellanox InfinitBand adapters to enable the GPU-to-GPU communication feature. Although this can improves the memory transferring speed among GPUs and simplify the communication among GPUs in GPU clusters, applications are still required to manually initialize and handle each GPU running on different nodes in the GPU cluster.

The steep learning curve and complicated framework integration have caused many developers to stay away from GPGPU computer cluster before they can enjoy the benefit of it. The GPGPU computer cluster needs a higher level of API and an easy to use programming framework in order for programmers to utilize the processing power of GPGPU computer clusters instead of building everything from the ground.

2.5 CPU-GPU Benchmarking

The floating-point operations per second (FLOP) is the most common measurement unit used to measure the performance of a distributed computing system or a supercomputer currently, but some researches claim that FLOP is not

suitable. This is because FLOP only focuses on measuring the raw processing power of the processors while other properties like network latency or memory transfer latency (Gregg & Hazelwood, 2011) are often left behind. While the processor's frequency and number of scalar processors are also commonly used to measure the performance of CPU, but it does not work well on the GPU either (Lee, et al., 2010).

Existing benchmarking tools are inadequate to measure the performance of a distributed computing system that consist both CPUs and GPUs. This because they are mainly optimized for specific processor like CPU only, GPU only or processor from specific vendor only (Lee, et al., 2010). Furthermore, some evaluation methods or tools ignore the memory transfer overhead which is important when measuring the performance in a heterogeneous distributed computing system (Gregg & Hazelwood, 2011).

2.6 Related Work

In the following sections, OpenCL and MPI execution model are explained in detail in order to provide the foundation for this two programming frameworks which are required in this research study. The CPU role in GPGPU distributed computing systems is included explicitly in the following section because it is one of this research's focuses which is excluded by many other GPGPU research projects.

2.6.1 OpenCL Execution Model

The OpenCL is an open royalty free standard programming framework that enables parallel programming across heterogeneous platforms like CPUs, GPUs and

other parallel processors (Khronos OpenCL Working Group, 2010). It is currently maintained by Khronos Group and adopted by many hardware manufacturers such as Apple, AMD, IBM, Intel and NVIDIA. The OpenCL consists of a subset of extended C programming language that used to write program for the kernel and a set of API to initialize and control the kernel.

The way the application executes on the GPUs using programming frameworks like CUDA or OpenCL is slightly different when compared with how the applications executes on the conventional CPUs. A GPGPU application can be divided into two main parts, the host program executes on the host or CPU and the program executes on GPU. The program that executes on GPU is also called kernel. The host program is used to setup and manages everything that is required for kernel to execute on the target devices, while the kernel is used to define the functions that required to execute repeatedly on devices like GPU (Khronos OpenCL Working Group, 2010).

Figure 2.1 shows the general flow of an application executing on a GPU using the OpenCL programming framework. First, the application needs to detect and retrieve all information on devices that are capable of executing the kernel. Then, the context is created base on the devices found. The context is very important because it contains all resources like devices, kernels and memory objects (Khronos OpenCL Working Group, 2010). After that, the command queue for each device is created. It is used to coordinate the execution of the kernels on the devices. The program source, which contains the implementation of kernel, is compiled and built into program object and then convert into kernel.