PC-Grade Parallel Processing and Hardware Acceleration for Large-Scale Data Analysis

Yang Su

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the degree of Doctor of Philosophy

School of Computing and Engineering

University of Huddersfield

October 2009

Acknowledgments

I would like to thank the School of Computing and Engineering at the University of Huddersfield for providing this great opportunity of study and facilitating me throughout this project. I wish to thank my colleagues at the Computer Graphics, Imaging and Vision (CGIV) Research Group and the Centre of Precision Technology within the University of Huddersfield for their continuous and consistent help and support to the project and myself.

First and foremost, I would like to express my sincere gratitude to my director of studies, Dr Zhijie Xu, for his exceptional support and guidance throughout the project. Having Dr Xu as an adviser has been an amazing experience. He was willing to take a chance on my research from the beginning, and has always pushed me to fill in that one last detail to elevate the level of my thinking and my work.

Great appreciation also goes to my second supervisor, professor Xiangqian Jiang, whose help and support has been of significant benefit to me during the project.

A great deal of consideration and thanks must go to my family. My parents, ChengXiang and Sufen Su, continue to be my role models for living life with passion, creativity, and hard work. More than anyone else, however, I want to thank my wife, Li Ma, and son, Jiayao Su, for their patience and support throughout this very long journey, at least, that is how I felt. They have sacrificed many days without me, yet all of this would be for nothing without them.

Abstract

Arguably, modern graphics processing units (GPU) are the first commodity, and desktop parallel processor. Although GPU programming was originated from the interactive rendering in graphical applications such as computer games, researchers in the field of general purpose computation on GPU (GPGPU) are showing that the power, ubiquity and low cost of GPUs makes them an ideal alternative platform for high-performance computing. This has resulted in the extensive exploration in using the GPU to accelerate general-purpose computations in many engineering and mathematical domains outside of graphics. However, limited to the development complexity caused by the graphics-oriented concepts and development tools for GPU-programming, GPGPU has mainly been discussed in the academic domain so far and has not yet fully fulfilled its promises in the real world.

This thesis aims at exploiting GPGPU in the practical engineering domain and presented a novel contribution to GPGPU-driven linear time invariant (LTI) systems that are employed by the signal processing techniques in stylus-based or optical-based surface metrology and data processing. The core contributions that have been achieved in this project can be summarized as follow. Firstly, a thorough survey of the state-of-the-art of GPGPU applications and their development approaches has been carried out in this thesis. In addition, the category of parallel architecture pattern that the GPGPU belongs to has been specified, which formed the foundation of the GPGPU programming framework design in the thesis. Following this specification, a GPGPU programming framework is deduced as a general guideline to the various GPGPU programming models that are applied to a large diversity of algorithms in scientific computing and engineering applications. Considering the evolution of GPU's hardware architecture, the proposed frameworks cover through the transition of graphics-originated concepts for GPGPU programming based on legacy GPUs and the abstraction of stream processing pattern represented by the compute unified device architecture (CUDA) in which GPU is considered as

not only a graphics device but a streaming coprocessor of CPU. Secondly, the proposed GPGPU programming framework are applied to the practical engineering applications, namely, the surface metrological data processing and image processing, to generate the programming models that aim to carry out parallel computing for the corresponding algorithms. The acceleration performance of these models are evaluated in terms of the speed-up factor and the data accuracy, which enabled the generation of quantifiable benchmarks for evaluating consumer-grade parallel processors. It shows that the GPGPU applications outperform the CPU solutions by up to 20 times without significant loss of data accuracy and any noticeable increase in source code complexity, which further validates the effectiveness of the proposed GPGPU general programming framework. Thirdly, this thesis devised methods for carrying out result visualization directly on GPU by storing processed data in local GPU memory through making use of GPU's rendering device features to achieve real-time interactions.

The algorithms employed in this thesis included various filtering techniques, discrete wavelet transform, and the fast Fourier Transform which cover the common operations implemented in most LTI systems in spatial and frequency domains. Considering the employed GPUs' hardware designs, especially the structure of the rendering pipelines, and the characteristics of the algorithms, the series of proposed GPGPU programming models have proven its feasibility, practicality, and robustness in real engineering applications. The developed GPGPU programming framework as well as the programming models are anticipated to be adaptable for future consumer-level computing devices and other computational demanding applications. In addition, it is envisaged that the devised principles and methods in the framework design are likely to have significant benefits outside the sphere of surface metrology.

List of Publications

1. Yang Su, Zhijie Xu (2009) **"Parallel Implementation of Wavelet-based Image Denoising on Programmable PC-grade Graphics Hardware".** Signal Processing, ISSN: 0165-1684, In Press, Corrected Proof.

2. Yang Su, Zhijie Xu and Xiangqian Jiang (2009) **"Real-time VE Signal Extraction and Denoising Using Programmable Graphics Hardware".** International Journal of Automation and Computing, ISSN: 1476-8186, Vol.6, Issue 4, pp.326-334.

3. Yang Su, Zhijie Xu, Xiangqian Jiang and J. Pickering (2008) **"Discrete Wavelet Transform on Consumer-Level Graphics Processing Unit".** Proceedings of Computing and Engineering Annual Researchers' Conference 2008, ISBN 978-1-86218-067-3, UK. pp. 40-47.

4. Yang Su, Zhijie Xu and Xiangqian Jiang (2008) **"Stream-Based Data Filtering for Accelerating Metrological Data Characterization".** Proceedings of the 14th International Conference on Automation & Computing, ISBN 978-0-9555293-2-0, September 2008, London. pp. 81-85.

5. Yang Su, Zhijie Xu and Xiangqian Jiang (2008) **"GPGPU-based Gaussian Filtering for Surface Metrological Data Processing"**. Proceedings of the 2008 12th International Conference Information Visualisation, July 2008, London. pp. 94-99. ISSN:1550-6037.

List of Figures

List of Tables

List of Abbreviation

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ARB	Architecture Review Board
ASIC	Application-Specific Integrated Circuit
ASMP	Asymmetric Multiprocessing
CFD	Computational Fluid Dynamics
Cg	C for Graphics
CMT	Chip Multithreading Technology
COM	Component Object Model
CPT	Centre of Precision Technology
CWT	Continuous Wavelet Transform
CUDA	Compute Unified Device Architecture
D3D	Direct3D
DMA	Direct Memory Access
DFT	Discrete Fourier Transform
DWT	Discrete Wavelet Transform
EIB	Element Interconnect Bus
FBO	Frame Buffer Object
FBS	Filter Bank Scheme
FFP	Fixed Function Pipe-line
FFT	Fast Fourier Transform
FP	Floating-Point
FPGA	Field Programmable Gate Array

GDI	Graphics Device Interface
GE	Geometry Engine
GKS	Graphical Kernel System
GL	Graphics Library
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GPGPU	General-Purpose Computing on GPU
GRF	Gaussian Regression Filter
GUI	Graphics User Interface
HLSL	High-Level Shading Language
HP	Hewlett-Packard
HPC	High Performance Computing
IC	Integrated Circuit
IDWT	Inverse Discrete Wavelet Transform
IPPS	Integrated Parallel Processing Systems
LTI	Linear Time-invariant
MAD	Multiply and Add
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
MSE	Mean Square Error
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OPD	Optical Path Difference
OSSI	Optical Spectral Scanning Interferometry
Pbuffer	Pixel buffer
PDE	Partial Differential Equation

PFP	Programmable Function Pipeline
PHIGS	Programmer's Hierarchical Interactive Graphics System
PPE	Power Processing Element
PSNR	Peak Signal-to-Noise Ratio
PVM	Parallel Virtual Machine
RC	Resistor and Capacitor
R & D	Research and Development
RGBA	Red, Green, Blue and Alpha
RISC	Reduced Instruction Set Computer
SDK	Software Development Kit
SGI	Silicon Graphics Inc.
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SM3	Shader Model 3.0
SM4	Shader Model 4.0
SMP	Symmetric Multiprocessor
SMT	Simultaneous Multithreading Technology
SNR	Signal to Noise Ratio
SPE	Synergistic Processing Elements
SSE	Streaming SIMD Extensions
T & L	Transform and Lighting
VBO	Vertex Buffer Object
VLSI	Very-Large-Scale Integration
VTF	Vertex Texture Fetch

Table of Contents

Chapter 1 Introduction

High Performance Computing (HPC) has been a widely studied topic in scientific research and engineering applications since the appearance of modern computers in the 1940s. A straightforward approach to this goal is to continually increase the processors' processing speed through technological innovations such as scaling CPU's frequency. However, a processor's frequency is limited to its power consumption for the reason that the core's power usage is scaled to its frequency (Rabaey, 1996). The limitation of power consumption actually hampered the trend of continuously increasing of CPU's processing power for HPC.

Another obvious approach to the goal is through better "structuring" of the process and data to maximize the efficiency of the computer. The fact that massive amounts of data can often be processed by the same function simultaneously; and/or many tasks can be performed concurrently for scientific computations had encouraged the extensive researches on the so-called "parallelism" in contemporary computer architectures. Generally speaking, the parallelism in computer architectures evolves along two directions -- a single computer with multiple cores or multiple processors such as supercomputer; or multiple computers working together on similar tasks through structures such as computer clusters or computer grids (Ian Foster, 1995; Sinnen, 2007). Both solutions have raised the issue of the cost of building those parallel computers, which often results in a dilemma between the computational performance and the hardware cost. In the past, parallel computers were often restricted to high profile government funded major scientific projects across the globe.

In recent years, an ever increasing number of consumer grade applications, in such areas as multimedia and graphics, have been pushing the performance boundary in between professional and amateur computers. For example, modern computer games have seen a big increase in the demand for computational power to cope with advanced graphics and Artificial Intelligence (AI) processing.

This situation had resulted in the innovation and production of the so-called consumer-level parallel processors; the best representative of which are today's Graphics Processing Units (GPUs). With a peak-speed performance over 933 Gigaflops (GFLOPS), the computation capacity of the latest GPUs dwarf today's commodity CPUs in terms of speed and cost. With the increasing programmability-empowered flexibility of modern GPUs, many researches and development projects have been focusing on the conception of general-purpose computing on GPU (GPGPU) with the aim of tacking computationally intensive tasks previously only processed on CPUs. Many traditional parallel computing paradigms and techniques have been mapped to GPGPU, including grid and cluster, synchronous and asynchronous processes.

This research project explored the concept of stream computing within the GPU design and programming paradigms. It then devised a programming framework for GPGPU applications, specifically for handling data intensive metrological analyses, on the basis of the inherent parallel architecture patterns of GPUs. The devised programming framework is then used for design the algorithm mapping models for GPGPU-based signal and image processing tasks.

1.1 Research Motivation

The main motivation of the research reported in this dissertation originates from the demand for real-time massive data processing power in a practical engineering domain – surface metrology. Metrological data often comes in huge volumes, and its visualization and profiling produce a serious problem for computational efficiency that has long been a bottleneck for surface analysis (Stout and Blunt, 2000). In general, the framework for processing surface metrological data is equivalent to a linear time-invariant (LTI) system, from which many signal processing algorithms originated (Blunt and Jiang, 2003). In this research, the main focus has been to explore the feasibility of adopting a consumer grade GPU to achieve data and process parallelism for generic LTI

systems, and to benchmark its hardware acceleration factors as well as the corresponding realization criteria.

In detail, the motivations of using the GPU for surface metrological data processing can be detailed as follows:

- A GPU is one of the most cost-effective, easily accessible forms of hardware available for implementing parallel processing among many existing parallel architectures (Owens et al., 2007). A typical GPU, equipped with several hundreds of arithmetic processing cores, will cost only a fraction of the price for a multiprocessor array with equivalent numerical processing power.
- Most researches reviewed in this project only focused on the segregated performance of algorithms run on GPUs. In practice, the GPU is still only a coprocessor of the CPU despite its amazing computing speed, i.e., a complete GPGPU program must also include settings and tasks run on CPU. Therefore, the performance evaluation of GPGPU implementations should also take into account the tasks performed on the CPU and the corresponding overhead of data communication in between the two. The ambiguity on this point has raised doubts on the GPGPU's practical values in engineering domains. This research tackles the challenges through exploring the performance of GPGPU-based surface metrological analysis/tasks in a comprehensive range of practical settings.
- For GPGPU researchers, there exists the challenge of how to effectively and efficiently represent computational resources and tasks on a GPU. The challenge is rooted in the fact that GPUs were initially developed to facilitate graphics rendering rather than general computational tasks (e.g., numerical modelling, linear computing, or signal processing (Owens et al., 2007). Traditional graphics Application Programming Interfaces (APIs) employ the GPU as a graphics device for dealing with elements such as textures, triangles, and pixels. To map an algorithm in terms of those primitives is not a straightforward operation, even for those developers who are familiar with computer graphics. The result was complex and entangled programming approaches, which often hindered the overall effort of harnessing the potential

of GPUs as mainstream computing devices. The experiments designed in this work aligned themselves to the target of obtaining a clear conception and practical approach to GPGPU programming. In addition, this effort is accompanied by the main GPU vendors such as ATi and Nvidia corporation, who have managed a continuous evolvement of GPU hardware architectures, for example, a uniform platform for the GPU programming. The research also investigated the influence of the GPU's hardware evolution on the future GPGPU programming framework.

- A rich and advanced body of work is also documented in this report on the architecture patterns developed for GPUs in the last decade. These centred around the parallel architectures, stemmed from CPU paradigms. The work aimed to investigate the architectural patterns of various GPUs, to form a generic guideline for the future design of application frameworks for GPGPU programming.

Driven by above goals and targets, the research works in this project were designed and developed around a practical engineering domain, the surface metrology. This was carried out in collaboration with the Centre of Precision Technology (CPT) at the University of Huddersfield, which is a centre research on surface metrology. The outcome of the research is expected to have potential value for the wider engineering and scientific communities.

1.2 Research Questions and Evaluation Strategy

Although sporadic researches in GPGPU domain have been carried out in recent years, those researches were normally focusing on a specially tailored application, which requires extensive and intricate considerations on the hardware feature of the employed graphics card that possibly resulting in different GPGPU solutions which are difficult to carry out performance evaluation. This is due to the fact that the hardware structure of GPUs and their programming platforms have evolved dramatically in the last decade and created many

diversions. Therefore, it is of vital importance to define common principles or rules to guide the GPGPU application design, so those principles can be extensively applied to cover the different generations of hardware and software tools. Therefore, this is the first question that needs to be tackled within this thesis.

As stated in the aforementioned research motivations, the thesis is based on the practical engineering applications for data analysis and processing in surface metrology. The ultimate task of surface metrology is to profile a surface using the measured and processed data, to which the ideal solutions are to increase the number of samples and to employ more sophisticated algorithms to achieve higher data accuracy, but often with the deteriorating computational efficiency as a cost. Therefore, the challenge of data analysis in surface metrology is largely attributed to the dilemma of data accuracy and computational efficiency. The second challenge faced by this research is whether the GPGPU concept and existing techniques can sufficiently support a flexible solution to the complex processes normally involved in metrological data operations. The feasibility and practicality of the solution will be evaluated by two vital parameters - the speed up factors and data accuracy of the deployed GPGPU programs. It is noted that the result of the evaluation will determine the validity of the designed GPGPU programming models

Normally, a complete GPGPU program comprises three parts, the tasks need to be implemented on CPU in serial mode, the tasks can be deployed on GPU in parallel, and the interface instructions between CPU and GPU. It is noticed that many researches of GPGPU emphasize the acceleration on tasks implemented on GPU, which doesn't adequately reflect the GPGPU's overall promotion on computational efficiency. How to consider the impact of the first and third part of a program on the overall performance of an application is another question that this thesis will discuss. It is anticipated that the solutions to this question will vary with the variant nature of different applications. In the case studies, the communication or data exchange in between the CPU and GPU pairs has been treated as the core issues. The validation of the devised solutions will be

validated through testing the run time of data visualization and its weightings in the overall application time.

1.3 Outlines

The research method deployed in this thesis follows a typical research pipeline involving research motivation clarification, research question definition, domain review, concept formulation, model development, experiment, and result analysis and performance evaluation with real application data. After introducing the research motivation and research questions in Chapter 1, the dissertation began in Chapter 2 with a review of different types of parallelization, the recent development of GPU hardware, related graphics API, and corresponding high level shading languages and shader models. In Chapter 3, the conception and computational models of GPGPU are reviewed by highlighting some classical applications in GPGPU. The architectural patterns for parallel computing are also reviewed in Chapter 3 with the aim to link the specific type of parallel architectural pattern to GPGPU programming. Chapter 4 presented the general GPGPU programming framework that is devised based on the work in chapter 3and explained in details various GPGPU programming models for implementing different algorithms in LTI systems and their evaluation approach in terms of the speed up factor and the data accuracy. Chapter 5 reports the result from testing the GPGPU programming model and its software prototype for various filtering algorithms in 3D surface profiling. A classical Gaussian filter was chosen for its popularity in the designe of the GPGPU programming model to evaluate its acceleration performance. The case study in Chapter 5 represented the research findings on a simple LTI system in the spatial domain. Chapter 6 presented the programming model for accelerating a computational expensive process of wavelet-based denoising. The designed model ensured most of the computations of the discrete wavelet transform were performed on the GPU rather than the CPU for the maximum speed gain. The algorithm tested in this experiment is a cascaded LTI system in the spatial domain. Chapter 7 has followed another

approach to realizing the GPGPU's parallel processing framework. It was applied to the data analysis tasks in an optical spectrum scanning interferometry system, which has been used for nano-level surface metrology in the CPT Centre at the University of Huddersfield. This proposed model employs the Compute Unified Device Architecture (CUDA) as the new programming tool for realizing a LTI system in the frequency domain. Further theoretical and technical discussions were recorded in Chapter 8, as well as the conclusions of the study. The contributions and the anticipated future works were presented in Chapter 9.

Chapter 2 Review of Related Work

Parallel processing is a form of computation in which data are either being processed by the same group of functions simultaneously; or multiple tasks are carried out on the same input concurrently. There are four levels of parallelism in contemporary computers at bit, instruction, data, and task levels (Sinnen, 2007).

In this chapter, the 4 levels of parallelization are reviewed and an overview on the evolution of GPU hardware structures and their parallel programming tools are also provided. As a coprocessor, a modern GPU achieves data-level parallelism through its own dedicated memory (DRAM) and columns of arithmetic cores, each consists of a group of registers, shared memory, caches, etc. The innovative design and its continuous evolution led to the raw processing ability of GPUs exceeded that of CPUs by the start of the New Millennium. The latest Nvidia Tesla C1060 GPU released in 2008 could sustain up to 933 Gigaflops (GFLOPS[1]) while the Intel Pentium4 CPU appeared on market approximately same time can only manage 104 Gigaflops when assisting SSE (Streaming SIMD Extensions) instruction set were employed (Nvidia Corporation, 2009). At the same time, the improvement on GPUs has ensured its flexibility, which is backed up by the programmability, continuous renovation and update of GPU's hardware structure. It has achieved an amazing annual updating rate of 2.8 since 1993 (Owens et al., 2007).

2.1 Levels of Parallelism

Parallelism in computing is generally classified into bit-level, instruction-level, data-level, and task-level which are closely related to processors' architectures (Almasi and Gottlieb,1990).

[1] 1 GFLOPS means 10 billion floating-point operations per second.

The bit-level parallelism was the first form of parallel computing and was introduced by the first appearance of the very-large-scale integration (VLSI) based fabrication technology of integrated circuit (Sina et al, 2003). The concept was driven by the demand for doubling computer word sizes that represents the amount of information the processor can execute per cycle (EI-Rewini and Abd-El-Barr, 2005). Chronologically, 4-bit processors were substituted by 8-bit ones, and then 16-bit to 32-bit and 64-bit ones nowadays. Although the concept of bit-level parallelism is quite simple, it is essential for many advanced extensions and applications.

Instruction-level parallelism reorders instructions in a computer program and then combines them into groups that can be executed in parallel without altering the ultimate result. In modern processors, an instruction is implemented through a multi-stage instruction pipeline, in which each phase corresponds to a different processor's action (Berkovich, 1998). Different stages of variant instructions can therefore be overlapped to achieve instruction-level parallelism. For example, a Reduced Instruction Set Computer (RISC) processor has a five-stage pipeline which consists of fetch, decode, execute, memory access, and write back operations (Steve, 1995). The instruction-level parallelism is achieved through the following canonical orders, where the grey column stands for:

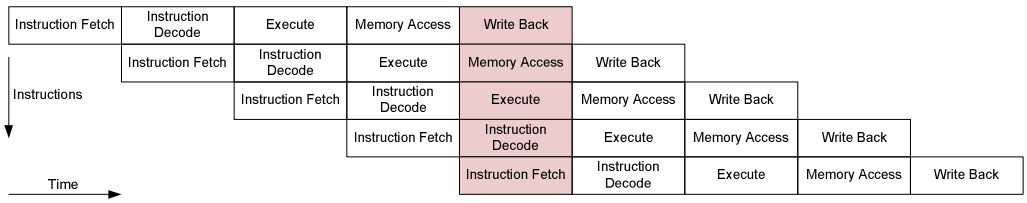


Figure 2.1 Different stage overlap of instruction pipeline in RISC machine

In addition, some processors which are known as superscalar processors can implement multiple instructions simultaneously if these instructions have no data dependency between them (Goossens, 2001).

In contrast, data parallelism is a more rigid form of parallelization in which all the elements in a data set are processed simultaneously by the same instructions. Data parallelism is often embedded in program loops, so it is also referred as

loop-level parallelism. Based on the relationships between instructions and data streams, Flynn summarized in 1972, four categories of common computing architectures, known as Flynn's taxonomy (Foster, 1995):

- Single Instruction Stream, Single Data Stream (SISD)
- Multiple Instruction Stream, Single Data Stream (MISD)
- Single Instruction Stream, Multiple Data Stream (SIMD)
- Multiple Instruction Stream, Multiple Data Stream (MIMD)

Among those, data parallelism is classified as a form of SIMD, which is normally achieved in a multiprocessor system, for example, consider a dual-core CPU unit carrying out a matrix addition operation. At runtime, the first core of that CPU adds all elements from the top half of the two matrices, while the second core adds all elements from the bottom half of the matrices. With the two cores working in parallel, the matrix addition will take half the time it would have if operations were performed in serial on a single-core CPU.

Compared with data parallelism in which the same instruction is implemented on multiple data sets, the task parallelism invokes a parallel program which issues independent calculations on either a single or multiple data streams (Rastello et al., 2003). Based on this definition, the aforementioned MISD and MIMD are both belong to the genre of task parallelism. However, some workers (Schneider and Rossignac, 1995; David et al., 1994) argue that MISD is actually a type of instruction-level parallelism, since the data streams processed by the instructions are the same as indicated in Figure 2.1. In a multi-processor system, task parallelism is realized when each processor executes a different thread (or process) on the data. The threads may execute the same or different instructions. In the general case, different threads communicate with each other through passing data from one thread to the next as part of a workflow (William and Rajeev, 2007).

It is obvious according to the definition of parallel computing and computing architecture, the MISD, SIMD and MIMD modes can all be employed for various degree of parallel computing. In these computing architectures, instructions are the control signals sent or received by processors or control units, while data

streams are the output or input of memory. The hardware structures of the MISD, SIMD and MIMD are shown in Figure 2.2 to Figure 2.4 respectively.

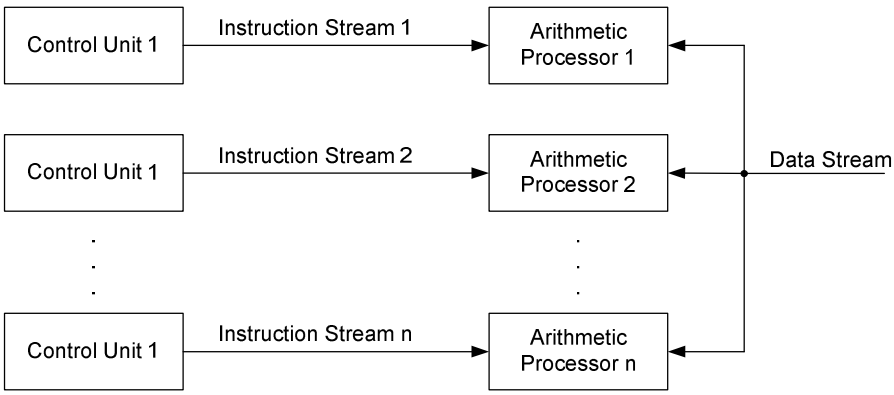


Figure 2.2 Models of a MISD architecture

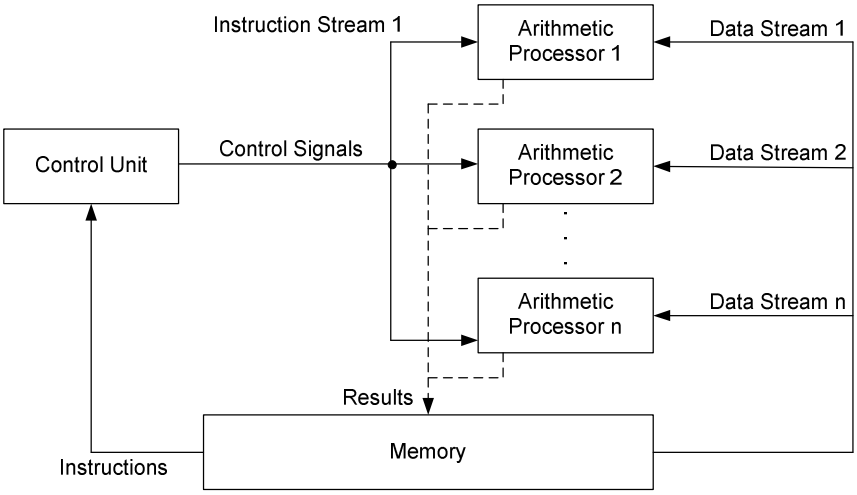


Figure 2.3 Models of a SIMD architecture

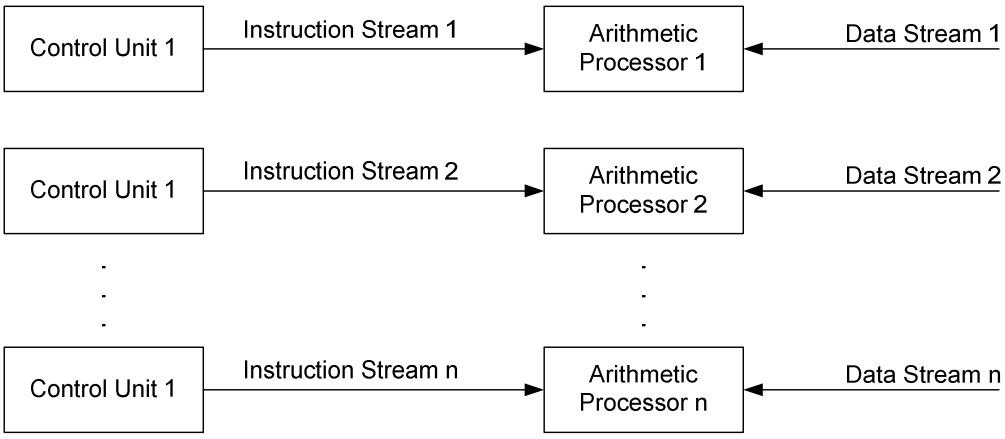


Figure 2.4 Models of a MIMD architecture

2.2 Types of Parallel Hardware

Memory units are a key element in all computing devices, where initial, intermediate, and resulting data are stored temporarily for further processes. Global memory in a parallel computing architecture can be a shared memory which is accessed by all processing elements in the same memory address; or a distributed memory in which each processing element has its own local address space (Foster, 1995). The term “distributed” means the memory is either logically distributed, or physically distributed. A shared and distributed memory is an integration of the two forms, in which every processing element has its own local memory as well as the ability to access memories on other non-local processors (Huang et al., 2004). Access to local memory is normally faster than to non-local memories.

A modern parallel computer often consists of a number of state-of-the-art processors, for example, vector, RISC, X86, IA-64, where these processors can be arranged into various forms of shared memory modules. A number of those modules can be further integrated into distributed memory-based parallel computers, such as a cluster machine. If required, multiple parallel computers can be connected into a synchronous or asynchronous network.

2.2.1 Multicore Structure

There are multiple execution units called cores in a multicore processor. The style of the instruction implementation in a multicore processor is different from that in a superscalar processor. A superscalar processor can implement several instructions per clock-cycle from one instruction stream - the so-called thread. In contrast, a multicore processor can implement several instructions per clock-cycle from several instruction streams (Thomaszewski et al., 2008). Recent hardware advancement has proven that actually each core in a multicore processor can act as a superscalar one as well, i.e., each core implements several instructions from one instruction stream on every cycle (Steven et al., 1997).

In terms of actual production, the Intel's Hyper-Threading (Intel Corporation, 2007) is one of the best known simultaneous multithreading machine which is an early form of pseudo-multicoreism, while Intel's Core and Core 2 processor series are the true-meaning multicore architectures (Intel Corporation, 2008). The latest IBM's Cell CPU is another representative form of the multicore technology (Gschwind, 2007).

2.2.2 Symmetric/Asymmetric Multiprocessor Structure

Multicore processor systems employ a single processor that has multiple pipelines for integer and floating-point operations. Multiple identical processors can also be connected to a single shared main memory to form a symmetric multiprocessor (SMP), in which the processors are capable of accessing the same shared memory through a bus or crossbar switch (Kaya, 2005). The SMP system allows any processors to carry out any task simultaneously. Based on properly designed operating system, a SMP system is able to readily transfer tasks across processors to distribute the workload evenly.

However, in implementation, the bus contention for enabling more than one processor to allocate data on the bus simultaneously can be a bottleneck and

limits the scale of the processor numbers in a SMP system, which results in the fact that the processors in a SMP system is normally less than 32. The alternative solution for the SMP is an asymmetric multiprocessing (ASMP) structure in which a group of separate specialized processors are employed for specific tasks (Robert et al., 1998). In contrast to the SMP of assigning all of the tasks in the system identically, an ASMP system only assigns specific tasks on specific processors. The common ASMP structure is a kind of clustered multiprocessors in which just a portion of the entire memory can be accessed by all processors (Cai et al., 2004).

2.2.3 Cluster Structure

As indicated above, ASMP structures can practically be categorized as the cluster structure according to Flynn's taxonomy that can be viewed as a way of building low-cost and distributed-memory MIMD computers. Gene Amdahl from IBM, who put forward Amdahl's Law for parallel computing, defined the distinction between the multiprocessor computing and the cluster computing in 1967(Moncrieff et al., 1996). Stated simply, the main difference is the communication modes where in multiprocessor computing it is issued inside the computer through internal bus structures, while in cluster computing it is based on the outside network such as local network, wide access network(WAN), or the Internet.

Based on the packet switching networks invented in 1962 (Natalia and Victor, 2006), the first commodity network employing computer cluster theory was presented by the ARPANET project in 1969 (Douglas, 2009). As the ARPANET evolved into the Internet, the original computer cluster connected by a Packet switching network also grew into the "proper" cluster in which the communications between the nodes uses the TCP/IP protocol, based on the Ethernet network framework (Thomas and Zsolt, 2007).

The first successful commercial clustering product was the VAXcluster released by DEC in 1984 (Thomas and Zsolt, 2007). Besides supporting parallel

computing in general terms, the VAXcluster also support the shared file systems and the peripheral devices. Following the success of VAXcluster, various commercial clusters were released in turn, such as the Tandem Himalaya and the IBM S/390 Parallel Sysplex, both released in 1994 (Thomas and Zsolt, 2007). With the growing maturity of cluster computers, the parallel computing ethos has encouraged further development into techniques such as grid computing where more focus has been put into the throughput of a computing utility rather than running a deliberately designed, optimized, and tightly-coupled jobs.

2.2.4 Grid Structure

In grid computing, a number of computers (irrespective of their individual architectures) are loosely connected via a network. In the most extreme case, each machine (including the properties of connections between them) is assumed to be different. This makes for an extremely heterogeneous system, which requires the coarsest level of parallelization since the work must be divided into independent units that can be completed on different computers at different speed, and returned to the main solution coordinator at any time and in any order without compromising the integrity of the solution (Thomas and Zsolt, 2007). Although there are tasks that are naturally amenable to this level of parallelization, a broader applicability of this approach requires much further research and infrastructure development. Successfully tested cases so far has been focused on the analysis of very large sets of independent data blocks, in which the problem lies in the total size of data to be analyzed.

2.3 Overview of GPU Architecture

2.3.1 The Origins of Graphics Processing

In the 1960s and 70s, graphics devices were just viewed as a kind of output equipment for computers. Limited to the hardware status, developers commonly considered the criteria for Graphics User Interface (GUI) from the view of

software capacity and adaptability. The Graphical Kernel System (GKS) (Hopgood et al., 1983; Enderle et al., 1984) and Programmer's Hierarchical Interactive Graphics System (PHIGS) (Howard et al., 1991) were representative standards. A typical graphics pipeline is defined by those standards as depicted in Figure 2.5.

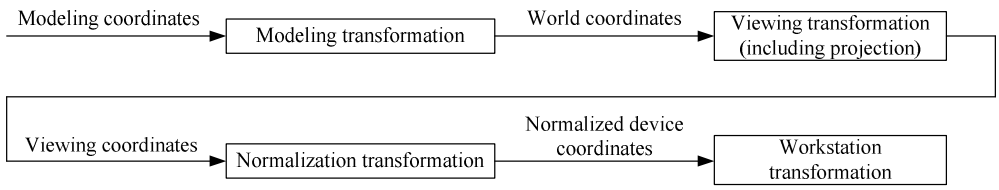


Figure 2.5 Abstract graphics pipeline defined in PHIGS

In the early 1980s, “new” graphics processors, that had been inspired by the innovative geometry engine (GE), were launched by various manufacturers. Graphics cards developed at this stage were dominated by graphics processor, which was an integrated chip on the computer motherboard with built in geometrical functions. The core of a GE is the support for floating point number computation between any 4-component vectors (Clark, 1982). These computations were used for coordinate transformation, blending and projection. A complete three-dimensional (3D) graphics pipeline can be accomplished by 12 such geometrical elements. James Clark, the designer of the geometry engine, then setup Silicon Graphics Inc. (SGI) in 1981 on the basis of GE technology (Watt, 1999). SGI had a significant influence on the development of computer graphics in the following decade; Graphics Library (GL) and the subsequent OpenGL became the industry standard of GUI for graphics processing.

As stated earlier in this section, the two key performance indications for modern graphics devices are the raw processing speed and the flexibility – the ability to adapt or customise. From 1980s to 90s, some basic functions of graphics processing could be accessed by lower-end graphics cards, attributing to the performance enhancement of the GE core. However, most of the applications of 3D graphics were still only manageable by higher-end workstations. At most stages of the graphics pipeline flow, functions were still actually accessed and executed by the CPU. Although the notation of GPUs appeared before 1995,

they were only viewed as graphics accelerators, instead of as a programmable core and a flexible processor. In the era of CPU dominance, a prominent event was the adoption of Single Instruction Multiple Data (SIMD) for fragment operations in 1992. SIMD is a technique traditionally employed by parallel computing applications to achieve data-level parallelism. In 1980, a research group at the University of North Carolina in USA first employed SIMD in their graphics software, Pixel-Planes (Fuchs and Poulton, 1981; Fuchs et al., 1989) and Pixel-Flow (Molnar et al., 1992), which marked the take-off of dedicated vector-computation -- though still at the software level and driven by CPU.

2.3.2 Evolution of GPU's Hardware Architecture

The evolution of post-90s GPUs can be divided into 5 stages, display adapter, transform and lighting (T & L) chip, programmable shader, CineFX engine, GPGPU unit, and multi-core.

- **Stage 1 – mid 1990s**

Before 1995, the graphics core was mainly functioned as a "display adapter". The graphics hardware were developed by main stream manufactures like Intel and AMD for desktop displays with occasional 2D acceleration ability. With the emerging of 3D computer games, the conception of "3D acceleration" began to take more shares in the design of graphics hardware. The 3DFX VooDoo series from Nvidia were first released in 1995 and were widely viewed as the market pioneers of the new generation of graphic cards with "3D acceleration" functions. To achieve the innovative 3D acceleration, the 3DFX VooDoo series were first equipped with two remarkable features, Z-buffer and texture mapping. The prior Z-buffer, also called depth buffer, resolves the visibility problem in 3D scenes through storing the depth information of a generated pixel (the z-coordinate) in a reserved buffer (Blasquez and Poiraudeau, 2004). The latter, texture mapping, renders the detail of an object's surface through applying textures, or colours to all projected pixels of a computer-generated 3D model (Pharr et al., 2005). The two major players of the commercial graphics device market, Nvidia and ATI, also

released their graphics cards that had similar functions of 3DFX VooDoo, the Nvidia Riva TNT and ATI Rang series.

Although a great leap from the earlier graphics software based graphics processors, the key problem of the products at the time was that the actual geometry processing was still carried out on CPU, which presented a heavy burden on CPU efficiency and seriously implicated the real-time performance of many 3D applications such as computer games. The abstract of the mid-90s graphics pipeline is depicted in Figure 2.6.

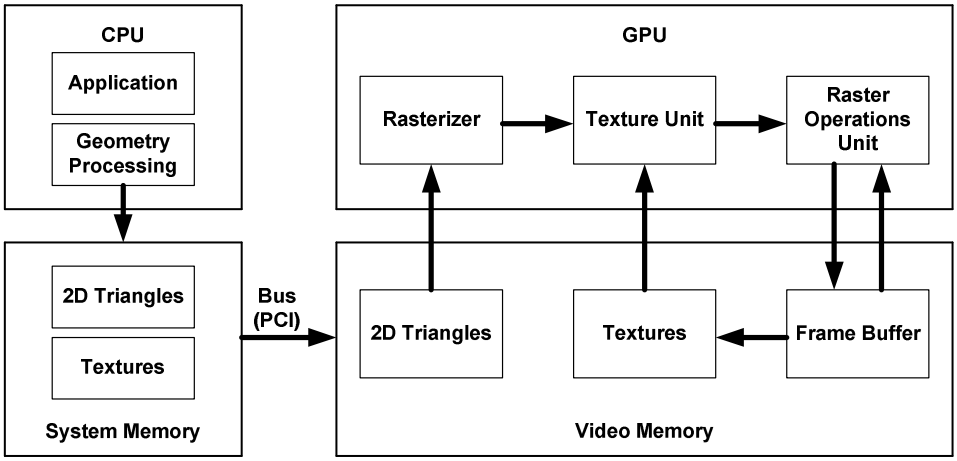


Figure 2.6 Abstract graphics pipeline between in 1995 and 1998

- **Stage 2 – late 1990s**

Followed the success of Riva TNT2, Nvidia released GeForce 256 and GeForce 2 successively which were viewed as the first “true” GPUs because the computations for geometric transformation and lighting (T & L) were embedded in the core of the graphics cards. Hence graphics functions were carried out on the independent graphics cards (Seitz, 2006). At about the same time ATI published their counterpart that has the approximate functions equivalent to the Nvidia’s GeForce 256 —the ATI Radeon 7500 (ATI Corporation, 2007). Thus the year 1999 was widely viewed as a new era in the evolution of GPUs for distinctively separating GPU and CPU functions. The abstract of the graphics pipeline at the time is shown in Figure 2.7.

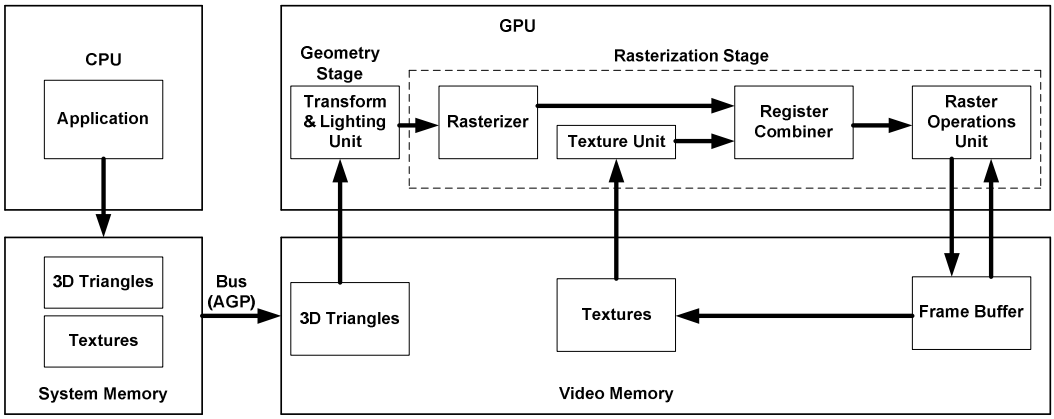
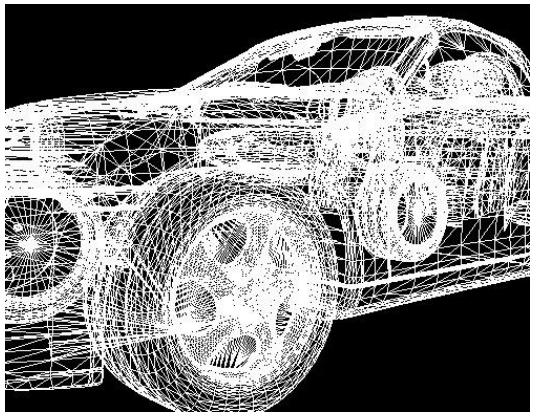


Figure 2.7 Abstract graphics pipeline (integrated T & L) at late 1990s

The shifting of the T&L from CPU to GPU was a great boost to the real-time polygon/vertex processing capacity, while the idealised local illumination models – directional, point, and spot – had simplified the computation and greatly increased the final rendering quality. Figure 2.8(a) and (b) highlight the enhanced GPU capability on polygon numbers and lighting.



(a) Rendered polygon on GeForce 256

(b) Lighting effect of GeForce 256

Figure 2.8 The enhanced GPU capability (Courtesy to Nvidia Corporation)

In contrast to a previous generation GPU -- Riva TNT2 which had just 2 parallel rendering pipeline, GeForce 256 has provided 4 parallel rendering pipelines. Each pipeline has a dedicated texture unit to access textures in parallel in each rendering pass (Nvidia Corporation, 2009). However, most of the GPU functions of this generation were still largely hard-wired in the physical IC chips and provided little flexibility for customization.

- **Stage 3 – early 2003**

In 2001, Nvidia released the GeForce 3, one of the first to integrate a programmable vertex shader. Vertices, points in a 3D space, marking the intersection of edge, are the most primitive elements in 3D geometry. However, they are also the most important “bricks” for forming line segments, polygons, and meshes (wireframe models). The vertex shader is a compiler for generating vertex information which includes attributes such as coordinates and colours. The evolution of the programmable vertex shader from the original fixed-function-only graphics pipeline enabled modellers and programmers to have more space to design and render detailed 3D models according to specific application scenarios – a vital feature for modern computer games. Figure 2.9(a) shows a rough and jumpy skin surface rendered by a programmable vertex shader in comparison

with a less detailed smooth surface rendered by a fixed-function graphics pipeline.

(a) (b)

Figure 2.9 A 3D head rendered by vertex shader and fixed-function graphics pipeline respectively (Courtesy to Nvidia Corporation)

In 2002, Nvidia released its GeForce 4 series in which the programmable vertex and pixel shaders were both available. The GeForce 4 series added the static and dynamic flow control in its design, which was absent in the GeFoece 3. While the vertex shader controls the vertex attributes, the pixel shader manipulates each pixel's colour fill-up that is issued by certain transfer functions. In a demo rendered clip released by Nvidia, as shown in Figure 2.10, the intricate details of the mermaid's hair and the minute tail shift are controlled by specific pixel shaders and polynomial transfer functions designed by graphics programmers.

Figure 2.10 The animation effect produced by pixel shader (Courtesy to Nvidia Corporation)

As well as exploiting the newly introduced programmable vertex and pixel shaders of the graphics cards at the time, the processing speed was further accelerated by the continuously expanding of the number of parallel rendering streams. For example, up to 16 textures can be processed simultaneously in the GeForce 4 series, which had become the technical foundation for high-definition graphics. ATi corporation, another heavy-weight GPU vendor, has also had its flagship product – the Radeon 8000 series – pushed to the market around the same period with the programmability as the key selling point (ATI Corporation, 2007). Generally, the hardware architecture of GPUs at this stage can be summarized as in Figure 2.11.

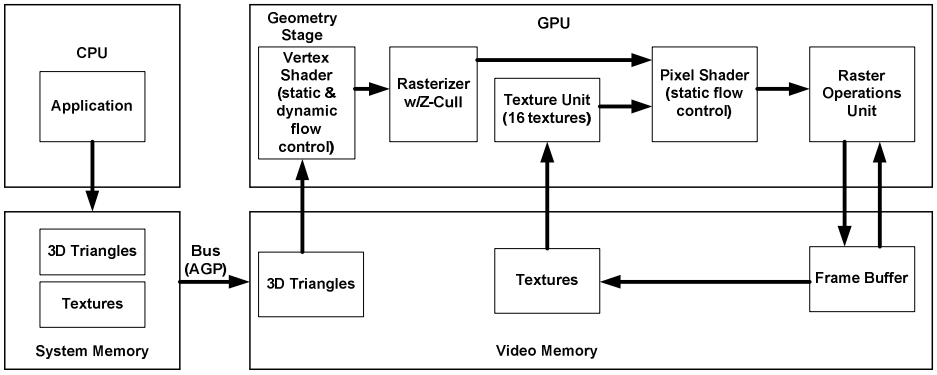


Figure 2.11 Hardware abstracts of GPUs with programmable vertex and pixel shaders

- **Stage 4 – mid 2000s**

Accompanied by the gradual maturing of the shader technology, Nvidia further developed the CineFX engine in its GPU product in the mid-2000s aiming to produce the cinematic visual effects. ATi soon followed up with its own SmartRender technology similar to the CineFX. CineFX is now in its 4th generation and is consisted of three cores -- vertex shader, pixel shader, and intellisample texturing. The version upgrade of CineFX has largely reflected the underlying shader model evolution which expands the available instruction set. For example, CineFX 3.0 employed Shader Model 3.0 (SM3) while CineFX 4.0 has deployed Shader Model 4.0 (SM4) which will be explained in detail in Section 2.4.4. Since the trade off between graphical quality and computational efficiency

will always be a problem for GPU designers, the CineFX engine has introduced the intellisample technology to alleviate this dilemma. The intellisample technology is formed by two key parts – Colour Compression and Dynamic Gamma Correction, which are integrated in GPU's IC chips. Colour Compression ensures image quality through the so-called lossless compression, while the Dynamic Gamma Correction boosts the image vividness through using the adaptive texture filtering technology.

After the short market presence, CineFX 1.0 and 2.0 were quickly replaced by the CineFX 3.0 which was widely viewed as a matured technology and welcomed by fans of high-definition graphics. CineFX 3.0 was first embedded in Nvidia's GeForce 6800 released in 2004, the counterpart from ATi is RADEON X800. Both the GeForce 6800 and the RADEON X800 supported high-level shading language (HLSL) – a C-like programming language such as OpenGL Shading Language (GLSL) and C for Graphics (Cg) for vertex shader and pixel shader development. In addition, the Shader Model 3.0 (SM3) employed by the device supports 32-bit float-point precision that results in fewer artefacts (Nvidia Corporation, 2009). Another distinctive feature of SM3 is its ability enabling the vertex shader access to textures at runtime, which is crucial for performing GPU-driven simulations. It was common for GPUs of this generation to support 64-Bit colour depth at the stage of pixel shading. On the IC chip design and manufacturing side, the GeForce 6800 contains 222 million transistors that ensured a theoretical peak up to 40 GFLOPS in contrast to 6 GFLOPS for a 3 GHz Intel Pentium4 SSE unit released in the same period (Pharr et al., 2005). Only 6-month later, Nvidia released the GeForce 7800 GTX and ATI had its RADEON X1800 released, which has seen the GPU raw power almost quadrupled to 160 GFLOPS (Owens et al., 2007). The abstract model of the graphics pipeline of this generation is drawn as in Figure 2.12 – notice the highlight is on the mutually available texture memory in comparison to previous generations of GPU (see Figure 2.6, 2.7 and 2.11).

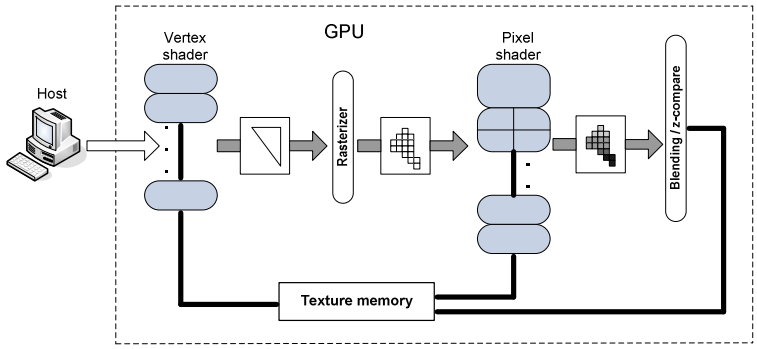


Figure 2.12 Model of the graphics pipeline of GPU released in 2004-2005

Since this project will largely based on the new found power of vertex/pixel manipulation of this generation of GPUs and beyond, it is useful to explain the actual workflow of it. The logic flow of vertex shader embedded in the GeForce 6800/7800 series is depicted in Figure 2.13 (Collange et al, 2007). Its working order can be simplified into the following phases: the host memory on CPU side sends the vertices' information across the CPU/GPU border, a vertex shader is then initiated to perform transformational (translation, rotation, and scaling) operations and local illumination calculation. Most of the computation will be based on arithmetic terms such as Multiply and Add (MAD), Exponential functions (exp, log), Trigonometric functions (sin, cos) and Reciprocal functions ($1/x$ and $1/\sqrt{x}$) to form physics equations, while the innovative texture memory access has enabled vivid rendering effect and real-time simulations such as shape deformity.

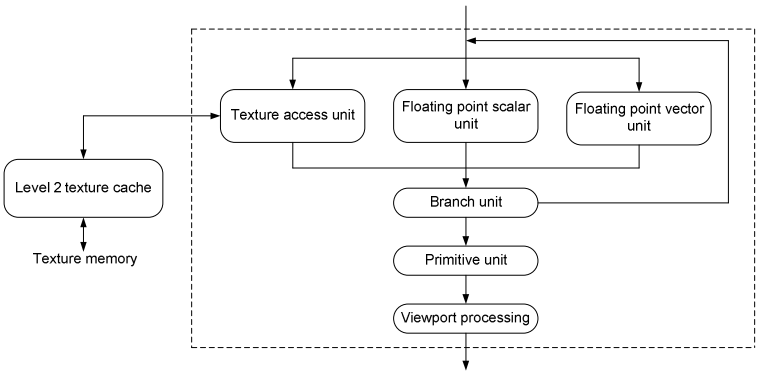


Figure 2.13 Vertex shader model of Nvidia GeForce 6800/7800 released in 2004-05(Courtesy to Collange)

The workflow of the pixel shader can be depicted as in Figure 2.14 (Collange et al, 2007). There are two floating-point (FP) unit appended with a Mini Arithmetic Logic Unit (ALU) that promotes the computation efficiency of FP numbers. The first FP unit can carry out up to 4 MAD operation at a time and accessing textures via the texture unit. The result is then sent to the second FP unit for up to 4 further MADs. A pixel shader of this model also includes a level-1 texture cache for rapid data accessing.

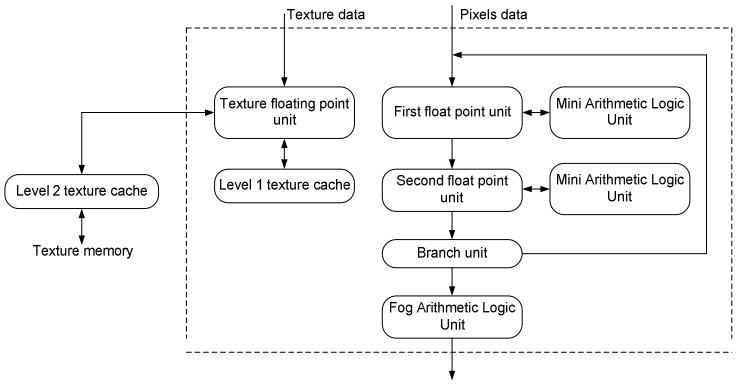


Figure 2.14 Pixel shader model of Nvidia GeForce 6800/7800 released in 2004-05 (Courtesy to Collange)

- **Stage 5 – current trend**

With the evolution of shader technologies, the concept of General-Purpose computing on the GPU (GPGPU) has become more and more popular, with the aim of addressing problems based on data-level parallelism. The earliest GPGPU programs in 2001 (Owens et al., 2007) was mainly focused on the areas of tailor-made applications such as image processing and matrix operations, while the latest GPGPU development has seen its extension into the applications of pattern recognition, signal processing, and physics simulation (Owens et al., 2007).

Although it has long been reorganized that GPUs can be treated as a parallel co-processor rather than mere graphics accelerators, the traditional GPU rendering pipeline had brought problems and challenges to researchers and developers in the past. Firstly, each discrete step of the accelerated algorithm need to be strictly mapped to the exact part of the rendering resources in the pipeline, which

is a tedious work for developers who are unfamiliar to graphics programming. Secondly, sometimes serious waste can occurred when work is distributing between vertex and pixel shader. The first problem requires intensive mathematical skills, while the second one demands knowledge of computer hardware, which are often unfamiliar to application developers.

The GPU's parallel computational capability is largely determined by the number of rendering pipelines available. The number of vertex and pixel shaders available in traditional graphics pipelines is determined by the anticipated ratio of the need for the functions during rendering. For example, the Radeon X1800 has 8 vertex shaders and 16 pixel shaders (ATI Corporation, 2007), and the GeForce 7800 has 8 vertex shaders and 24 pixel shaders (Nvidia Corporation, 2009); the ratio is 1:2 and 1:3 respectively. The workflow in a GPU for transforming 3D geometries into 2D graphics follows the order of vertex shader, pixel shader, and then to the framebuffer. Thus the actual number of parallel streams is limited by the narrower section of the pipeline, in this case, the number of vertex shaders. However, most of the GPU vendors advertise the number of rendering pipelines by emphasizing on the number of pixel shaders only, for marketing reasons. Some might argue that most of the successful GPGPU showcases are implemented on pixel shaders alone. Though the matter of fact is, without careful and pains-takingly tedious balancing of the workload, a problem can arise in which either all the vertex shaders are heavily working while most of the pixel shaders are idle, or vice versa. This situation can be illustrated by the following figure.

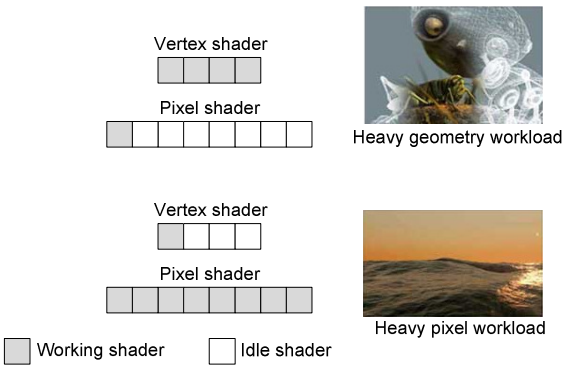


Figure 2.15 Workload unbalance in traditional rendering pipeline

To solve the problem of workload imbalance between dedicated pixel shaders and vertex shaders, Nvidia and ATI released Geforce 8800 (Nvidia Corporation, 2009) and Radeon HD2000 (ATI Corporation, 2007) successively in 2006. These two GPUs have employed a brand-new framework which adopted a unified pipeline architecture without a distinctive vertex and pixel shader borderline, as depicted in Figure 2.16.

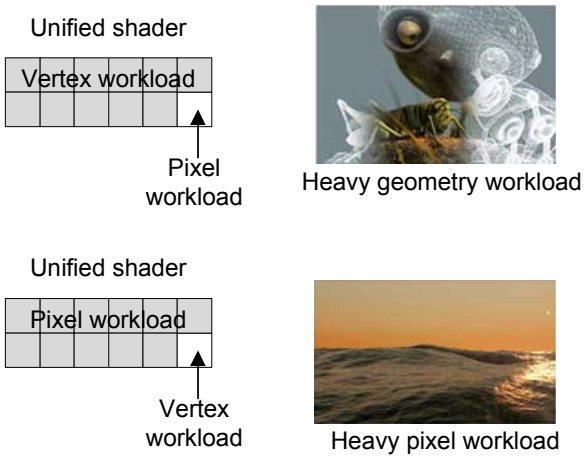


Figure 2.16 Workload allocation in unified pipeline

The employment of unified shader has made the Geforce 8800 and Radeon HD2000 into intrinsic parallel stream processors. The GeForce 8800 contains 128 unified shader units which are consisted of 681 million transistors and can sustain up to 518.4 GFLOPs at peak (Nvidia Corporation, 2009). An abstract view on the architecture of this generation of GPUs is shown in Figure 2.17.

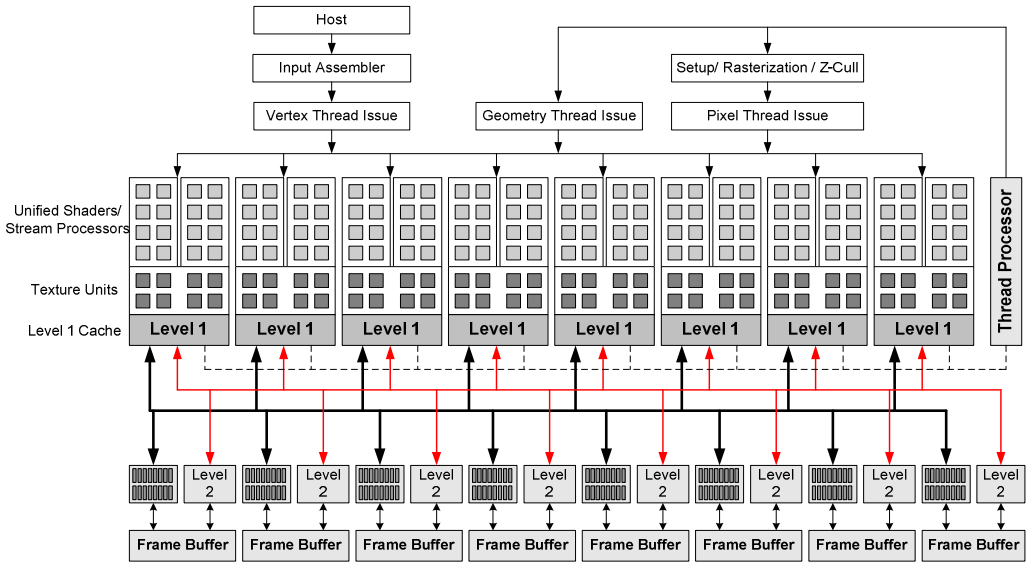


Figure 2.17 Architecture of unified shader arrangement (Courtesy to Nvidia Corporation)

In this design, the 128 unified shaders are clustered into 8 groups. Each group therefore consists of 16 unified shaders for accessing 8 texture units and a number of level 1 and level 2 caches. It is also apparent in this design that each unified shader can export the processed data to be “recycled” by other streams and practically forming the loop of thread processors. This GPU architecture guaranteed it can operate as a SIMD parallel processor with high efficiency. All the GPUs of this generation support IEEE754 double precision floating-point number arithmetic standard.

Following the first appearance of unified pipeline in 2006, Nvidia further released its mini-supercomputer series in 2008, Tesla computing solutions, which enable an energy efficient parallel computing framework with the improved precision to be built. The Tesla C1060, Tesla S1070 and Tesla Personal Supercomputers are the lower, middle, and higher end of the series and are all capable of meeting the challenges from data intensive, high performance computing (Nvidia Corporation, 2009). The Tesla C1060 architecture involves 240 cores and supports double precision floating-point computation with the peak rate up to 933 GFLOPs (Nvidia Corporation, 2009). Both the Tesla S1070 and the Tesla Personal

Supercomputers have equipped with 960 cores for larger scale applications (Nvidia Corporation, 2009).

2.4 Graphics APIs and Shading Languages

The vision of implementing general-purpose algorithms on computer graphics hardware for extra speed was first introduced in 1990, when Lengyel et al. used a rasterization device for robot motion planning, 4 years later Cabral et al. accelerated tomographic reconstruction on the VGA device (Blasquez and Poiraudeau, 2004). However, most of the early experiments had been designed for proof-of-concept and were never intended nor applicable for the mass PC market. The situation changed significantly with the introduction of programmable commodity graphics hardware in 2001 that boosted the popularity of this approach for wider application domains. Shortly after, the acronym GPGPU ("general-purpose computations on GPUs") was coined for this new research and development (R & D) domain. As stated in Section 2.3.2 that prior to 2006 the only way to access the raw power of graphics hardware was via the detour of graphics APIs and shading languages since no explicit GPGPU development tools where available. As a consequence, most of the work and research carried out before the date were focused on the implementation techniques and know-how rather than the core algorithms involved. While this has changed lately with the debut of the unified pipeline and Compute Unified Device Architecture (CUDA), this section still decides to present a broad overview of GPU programming solutions including both graphics APIs and shading languages. The text will describe OpenGL and DirectX (APIs), and Cg and HLSL (shading languages), with reference to the architecture of conventional PC graphics software, sketched in Figure 2.18.

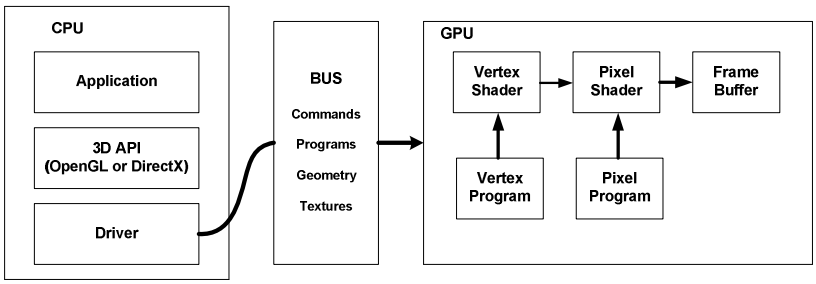


Figure 2.18 PC graphics API architecture

2.4.1 The Direct3D Route

The functionality of Microsoft's DirectX Application Programming Interface (API) is wrapped in the form of Component Object Model (COM) and managed code interfaces. DirectX constitutes graphics, audio, input, and network cores, depending on the version (Adams, 2003). Among the components, DirectDraw (prior to version 8) is for defining 2D graphics directly on the screen space and the Direct3D (D3D) is for handling 3D graphical task (Adams, 2003). Prior to DirectX, OpenGL was the dominant API on the market for consumer level rendering tasks. The situation finally changed with the formal publication of DirectX 7 by Microsoft in September 1999 after a prolonged trial period of its earlier versions.

A prominent feature of the Direct3D API in DirectX 7 is the new addition of the Transform and Lightning (T & L) pipeline hard-wired on the graphics card, which first conjoined the speed and quality of the computation of expensive lighting and geometrical calculations. The flagship off-the-shelf product at that time was Nvidia's GeForce 256. Although the joint force of the DirectX 7 software and the GeForce 256 hardware brought PCs into the GPU era, the pattern of Fixed Function Pipe-line (FFP) only allowed limited number of graphical and geometrical algorithms to be accessed in the configuration style, rather than programmed to specification.

The Programmable Function Pipeline (PFP) under which developers can have a degree of flexible control over vertex and pixel processing was realized by DirectX 8 released in November 2000 (Parberry, 2001). From then on, the

hardware-routed T & L in Direct3D 7 was formally substituted by vertex and pixel shader techniques in Direct3D 8, which made the GPU a true programmable processor. However, in Direct3D 8 shaders have to be programmed in assembly language, which is hard to master for most application-level programmers. The Direct3D 8 series introduced shader models 1.0/1.1/1.3/1.4 successively with the early Nvidia GPU products supported Shader models 1.0/1.1/1.3, and its ATI counterparts supported all versions of shader model 1 series (Szirmay-Kalos et al., 2008).

In December 2002 Microsoft released its most famous and successful Direct3D 9 API which supports improved shader model 2.0 and 3.0 (Szirmay-Kalos et al., 2008). Shader model 2.0 added static flow control to the vertex shader, and Shader model 3.0 enabled static and dynamic flow control of both the vertex and pixel shaders. Apart from the extension of the supported shader instructions, the most prominent feature of Direct3D 9 is its support for the 64-bit RGBA color in pixel shading, and the 128-bit precision (32-bit for each colour channel) floating-point computation (Luna, 2003), which further improved the visual effect and rendering quality.

The latest version of DirectX is the DX10 with the elementary graphics module Direct3D10 (D3D10) that was released in November 2006. It was designed around the new driver model in the Windows Vista operating system and supports shader model 4.0. The notable difference between D3D10 and the previous versions is that it employed a so-called geometry shader in its graphics pipeline (Walsh, 2008), which executes after the vertex shader with the whole primitives and/or their adjacency information as inputs to the process. When operating on triangles, all three vertices will become the geometry shader's input, and the output might emit zero or more primitives, which are then rasterized and passed on to the pixel shader. The benefit of the geometry shader is that it allows the manipulation of meshes on a per-primitive basis, that is, vertices can be treated as a single vertex array, a line segment (two vertices), or as a triangle (three vertices).

2.4.2 The OpenGL Route

Another identical route to access the GPU feature set is through the OpenGL. OpenGL (Open Graphics Library) was originated from the IRIS GL that was developed by the high-end workstation manufacturer Silicon Graphics (SGI). The steering group of this API – the OpenGL Architecture Review Board (ARB), which was formed by peoples from companies such as SGI, Intel, IBM, NVIDIA, ATi, Microsoft, Apple, was founded after the SGI's first release of OpenGL 1.0 in July 1992. One of the key tasks of the OpenGL ARB is producing an industry standard for OpenGL, and its tool kits, through common agreements among the ARB members. The approved standards are then published as specifications based on the C programming language. Only those APIs that passed all the tests regulated by the specification can be referred as official OpenGL. The first product of this process, OpenGL 1.1, was formally released in 1995 (Hill, 2001).

The original OpenGL specification serves two main purposes (Hill, 2001):

1) To insulate the complexities of interfacing with various 3D graphics accelerators, including GPUs, by exposing to programmers a single and uniform API;

2) To encapsulate the varying capabilities of hardware structures through enforcing all implementations to support the full OpenGL feature set.

As a graphics API, OpenGL's basic function is to process primitives such as points, lines and polygons, and convert them into pixels. The overall operations are carried out by the OpenGL state machine that is specified since the OpenGL 1.1 (Silicon Graphics Inc., 1996). Divergences of the OpenGL ARB partners, caused the first few releases of OpenGL APIs to be rather slow comparing with DirectX. Prior to version 1.5, the updates for OpenGL mainly focused on the minor modifications to the precious release. This situation had lasted for nearly a decade till July 2003. When the OpenGL ARB formally released its version 1.5 with the major innovation of the embedded "OpenGL Shading Language", known as GLSL (Kessenich et al, 2006). Since its debut in 2003, GLSL has become one

of the popular shading languages to develop interactive graphics and visualisation applications across operation systems from UNIX, Macintosh, Microsoft Windows, to Linux. This interchange ability enabled programmers to easily transfer their programs across most major commercial operating systems and hardware platforms.

In 2004 3Dlabs, a UK semiconductor company, substituted the dominant role of the SGI in the OpenGL ARB and unleashed the OpenGL 2.0 on the basis of OpenGL 1.5. It greatly improved the efficiency for some common operations from the previous versions and also added new features on creating photo-realistic, real-time 3D graphics that can be referred on SGI's website (Silicon Graphics Inc., 2005). The latest development has seen OpenGL 3.0 becoming widely available with roughly equivalent features and powers to D3D10.

2.4.3 Dedicated GPU Languages -- Cg and HLSL

The earliest form of shading languages is constituted by assembly instructions such as 'mov' and 'mod'. Although high on operating efficiency, in practice they are difficult to use and maintain. With the growth of the complexity of shader programs, the limitations of the assembly language approach were becoming more evident for the following reasons (Owens et al., 2007):

- Programs written in shader assembly language are difficult to program and debug;
- The number of instructions in an assembly shader is limited;
- Some flow control instructions are hard to issue in a shader assembly language, e.g., the loop instruction.

To better explore the new found computing power of GPUs, it is essential to employ a convenient shading language for GPU programming. Developers from the GPU giant NVIDIA defined and implemented a new shader language in late 2001, working in close collaboration with Microsoft. It was the earliest effort from Nvidia and Microsoft to devise a uniform specification for all GPU languages. The

results were two languages, NVIDIA's “C for graphics” (Cg) and Microsoft's “High-Level Shading Language” (HLSL). Although the two languages share identical syntax and semantics, they differ by ideology: Cg was designed as an additional layer on top of all popular graphics APIs, i.e. OpenGL and Direct3D, with a small performance penalty; while the HLSL offers a cleaner interface to applications through a tighter integration into the dedicated Direct3D framework.

In contrast to the early shading languages such as the Renderman Shading Language from Pixar Animation Studios and the Stanford Real-Time Shading Language, Cg and HLSL evolved on all aspects of graphics. Many functions have been added to address the functionality of the newly released GPUs; control flow operators were being supported; vectors with up to four scalars, and matrices up to 4 × 4 in size were supported; and some object-oriented techniques have been included. Changes can also be found in their software architecture design, for example, though the concept of the “programmable pipeline” still exists, it is combined with the idea of a virtualized machine that leads to the concept of language profiles. Cg is currently still under active development, with most of the changes applying to the architecture, rather than the language itself. In contrast, Microsoft seems has decided to break the compatibility of the two languages with the release of Direct3D10 which supports “geometry shaders”.

2.4.4 Evolution of Shader Models

As briefly mentioned in Section 2.3.2 and 2.4.3, before the release of CUDA in 2006, the main stream shading languages for GPU programming included HLSL from Microsoft, GLSL for OpenGL, and the Nvidia’s Cg. Although these shading languages differ in their designs, they actually follow a common uniform specification – the Shader Model that is defined by the aforementioned CineFX engine proposed by Nvidia described in Section 2.3.2. The CineFX engine regulates the specifications for vertex shader and pixel (or fragment) shader in accordance with the GPU’s hardware structure, e.g., number of stream processor and register, shader clock, memory amount. Table 2.1 lists the key specifications of the 3 major Shader Models.

Table 2.1 Key specifications of Shader Models (SM)

	SM 2	**SM 3**	**SM 4**
Max of Vertex Instruction Executed	65536	65536	65536
Length of Pixel Instruction	512	65536	unlimited
Constant Registers in Vertex Shader	≥ 256	≥ 256	16×4096
Constant Registers in Pixel Shader	32	224	16×4096
Temp Registers in Vertex Shader	13	32	4096
Temp Registers in Pixel Shader	32	32	4096
Loop count register in Pixel Shader	No	Yes	Yes
Static Flow Control in Vertex Shader	Yes	Yes	Yes
Dynamic Flow Control in Vertex Shader	Yes	Yes	Yes
Dynamic Flow Control in Pixel Shader	Yes	Yes	Yes
Vertex Texture Fetch	No	Yes	Yes

2.5 Languages for General-Purpose Computations

Up till now, this chapter has been focusing on types of graphical processing and data parallelism enabled by the GPU. However, most data intensive computation from wider world of application domains don't describe their tasks in the terms of vertices, polygons, and pixels. The aforementioned APIs and shading languages devised for graphics applications limit the wider acceptance of GPGPU in real-world applications because of the extra demand for graphics knowledge. For over a decade, the aforementioned approaches were the only way to develop applications on the GPU, however this has changed lately. This section provides a brief review on 3 GPU-based programming languages that are not mapped to the graphics route.

2.5.1 Brook for GPUs

The Brook language from the Stanford University in USA is one of the first substantial efforts in simplifying GPGPU application development. It was initially designed primarily as a programming language for “streaming processors” (Dally et al., 2003). Buck et al. (2004) adapted Brook to harness the capabilities of computer graphics hardware; making it the first general-purpose language for the GPU (Buck et al., 2004). Brook extends the C programming language by inducing the concept of streams, a collection of elements, where each element will be manipulated by the same computations. Streams are different to arrays in conventional serial computing because there is no index operation and the element dependencies are forbidden. The functionality that is applied to each stream element is called a kernel, which is comparable to a “shader”.

The application development in Brook is a two-phase process; first the task is coded and compiled to a set of C++ files, and then the C++ files are loading and execution on the host machine. One major drawback of this approach is the target operating system, that is the graphics device specifications and the graphics API, has to be specified in advance.

2.5.2 CUDA – “Compute Unified Device Architecture”

Echoing the hardware architecture evolvement, Nvidia has devised a new generation parallel programming tool set. The Compute Unified Device Architecture (CUDA), enables simplifies the application development tasks to a C-programming job.

The CUDA GPGPU toolkit was published by NVIDIA at the end of 2006. Similar to Brook, its syntax and semantics follow the standard ANSI C style, and also support stream types and their corresponding operations (Nvidia Corporation, 2009). In contrast to Brook, the CUDA toolkit can generate executable instructions on both the CPU and GPU without the need of any intermediate C++ files. In addition, CUDA does not need any graphics back-ends for storing and

displaying computational results. The current CUDA version supports unique features such as branching, looping, pointers, large kernels, and multiple threads.

In addition to the intrinsic functions, the CUDA framework also includes extra utility libraries for operations such as linear algebra and the fast Fourier transform (FFT) that are important for applications like digital signal processing. The detailed programming specification in CUDA will be further discussed in Section 7.3 in combination with a case study.

With the release of the unified shader architecture and the CUDA-based computing model, data-parallel processing on GPU has extended from the earliest graphics applications to other scientific and engineering domains such as signal processing, physical simulation, computational biology and even computational finance.

2.5.3 CTM – "Close-to-the-Metal"

At about the same time of NVIDIA's release of CUDA, its main market rival ATI (now part of the giant micro-processor manufacturer AMD) introduced the CTM platform -- a data-parallel virtual machine that allows direct communication with ATI graphics devices (Segal and Peercy, 2006). Similar to the CUDA architecture, many features are imposed by this approach, including the ability to read, modify, and write memory in a single program, to directly access host memory, or to cast between formats without explicitly copying the data. CTM is distributed as a library that allows "managed connections" to one of the three units of the graphics hardware to be opened, used, and closed: 1) The "command processor", which is programmed via an architecturally independent language. 2) The "data-parallel processor" that is programmed via a native (architecturally dependent) instruction set. 3) The "memory controller" which allows direct access to the graphics and the main memory.

As the name implies, CTM is used to access graphics hardware at a very low level, close to the machine code. It was designed for manual fine tuning of programs, and for exploring the GPUs horse power, but not for every-day use on

ordinary applications. Furthermore, a CTM application is responsible for all problems occurring at debugging, which increases the development complexity and cost.

2.6 Summary

Based on Leslie Lamport's (Sinnen, 2007) definition, there are multiple levels at which parallelization can occur in a computational platform; the simplest micro-parallelization takes place inside a single processor and usually does not require the intervention of the programmer to implement. The so-called medium-grained parallelization for its intermediate repetitive core is normally associated with the host language's semantics, and often appears in the form of advanced computational tasks, loop level parallelization. While efforts had been made in automating this level of parallelization with optimized compilers in the past, the results of those attempts were only of moderate success (Sinnen, 2007). For more advanced computational tasks, coarse-grain parallelization is often deployed which requires distributed memory parallel computers and are almost exclusively coded by the specially-trained programmer – not the application developers.

In practical engineering applications, there exist extensive specific processing procedures, such as reconfigurable computing and linear algebra matrix operations, which are implemented in specialized parallel devices, such as DSP, field programmable gate array (FPGA). Often, the key for the success of those devices is the cost, hence the invention of the term consumer-level or commodity-grade parallel processing. The majority of the attempts to date have focused on low-level data parallelism, but the recent trend has witnessed the interest shift to higher level parallelism, including instruction and task parallelism.

As an outstanding representation of this trend, GPU has been hotly pursued to become a "hardware accelerator" for software algorithms. Modern GPUs, and their application development tools, have undergone multiple generations of development as reviewed in this chapter. It is noted in this review that GPU is a rendering device in which a computational model distinct from those in CPU

programming is employed. Therefore it is essential to introduce conventional notations and programming methods that map to the graphics concepts in GPGPU, which will be the focus in Chapter 3.

Chapter 3 General-Purpose Computing on Graphics Card and Architectural Pattern in Parallel Computing

GPU's rapid evolution on its hardware design, coupled with the enhancing programmable capacity, has made GPGPU widely applicable in various domains of scientific computing, e.g., computational geometry, physically-based simulation, linear systems solution, partial differential equation, and database queries (Owens et al., 2007). Between 2001 and 2006, limited to the GPU pipeline structures that normally included vertex, rasterization, and pixel stages, the key GPGPU tasks had been focusing on how to efficiently implement an algorithm in the fixed rendering pipeline through mapping general-purpose computations to graphics hardware resources. Therefore, the key question to the GPGPU efforts was what types of computations map well to GPUs as briefly discussed in Chapter 2. Simply speaking, two key attributes of computer graphics computations, data parallelism and independence, will determine the outcomes and levels of success in a GPGPU application.

In this chapter, key GPGPU concepts and techniques will be discussed in terms of CPU-GPU analogies that refer to terminology such as stream, kernel, scatter, gather, task computing, render to texture, and multi-passes. Comparing to serial programming, parallel programming is more complex to realize due to the greater degrees of freedom involved. Serial programming normally only involves a single thread or time divided task tablets (multi-threads) of computation at any one time, while parallel programming involves multiple threads of computation which also need to communicate and synchronise with each other. It is essential for computer scientists to design a fixed set of high-level constructions for capturing common computational patterns from the parallel computer platforms (Flynn, 1966). Extensive researches have been carried out since the very beginning of

PC-grade parallel programming with this aim in mind. Based on these researches, four general classifications of parallel architectural pattern have been presented in this chapter, which are namely, divide and conquer, Pipes-and-Filters, communicating sequential elements, and processor farms.

3.1 Foundational Function Blocks: Streams and Kernels

For GPGPU applications, there are two essential components, stream and kernel, that distinguish data and instructions passed through the pipeline. A stream in GPGPU can be defined as the collection of data sets that need to be operated by the same computation. Multiple streams expose the so-called data parallelism due to the fact that all the data can be processed in parallel simultaneously. A kernel is the function or functions designed to perform the computations on each stream element. GPU's parallel processing ability appears not only in guaranteeing multiple stream elements being processed in parallel, but also on ensuring multiple kernels being executed in parallel (Marziale et al., 2007). The concepts of the stream and kernel in GPGPU computing are sketched in Figure 3.1.

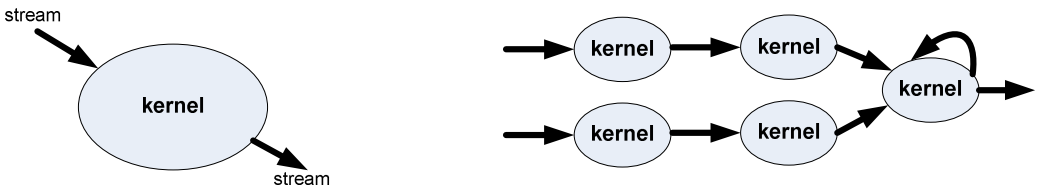


Figure 3.1 Stream and kernel in GPGPU programming

As indicated by the diagram and above discussions, both the vertex and pixel processing stages satisfy the parallel processing definitions through forming kernels and streams, however, since the pixel stage is located at the rear end of

a graphics pipeline and chosen to the temporary storages such as the frame buffer, it is more commonly used to issue complex algebraic computations. Actually, as a matter of design principle, the number of pixel shaders in a modern GPU often a few folds more than that of vertex shaders, which results in pixel shaders become much more powerful parallel processors than vertex shaders (Owens et al., 2007). Therefore, it is a common practice implementation details can be found in Chapter 5 and 6 in a GPGPU application that raw data are formed into pixel streams corresponding to textures stored in the GPU memory to be processed by instructions coded in pixel shaders. The following two subsections provide the analogies between CPU and GPU for the streams and kernels.

3.1.1 Data Streams

The native data layout on CPUs is a 1-dimensional (1D) array. A higher-dimension array can be accessed through offsetting coordinates into a separated 1D array. For example, the element $a[u][v]$ of a 2D array of the size $M \times N$ can be mapped into $a[u*M+v]$, assuming array indices begin from 0.

The native data layout for a GPU, however, is a 2D array in the form of textures or texture samplers. For example, in the graphics API -- OpenGL, a texture can be created by the instruction $glGenTextures()$ (Microsoft, 2006), whilst its size and data type can be specified by the instruction $glTexImage2D()$(Microsoft, 2006). In addition, since a pixel can have four colour channels -- red, green, blue and alpha (RGBA), if all are utilised to store elements of a vector, then a texture of the size $N \times N$ can support the maximum vector length of $4 \times N \times N$. Figure 3.2 shows a vector of 16 elements being stored in a piece of 2×2 texture with all 4 RGBA channels employed. In general, for an array with *N* elements, when mapped to a texture unit, the texture size can be expressed as:

$Height=LowerBorder[sqrt\ (N/4\ )]$;

$Width=UpperBorder[(double)(N/4)/(double)Height]$;

Except 4D colour textures, another type of texture, the luminance texture, often in greyscale uses only one channel which is also extensively used in GPGPU programming (Victor et al., 2005).

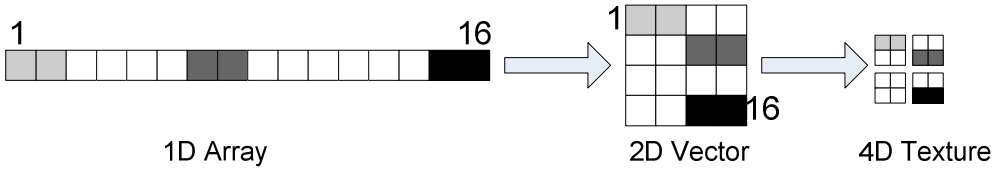


Figure 3.2 Data storage in RGBA textures

3.1.2 Instruction kernels

In the CPU programming paradigm, if all elements in a vector need the same operation, a loop would be used to iterate over these elements; in this case, the instructions inside the loop is the kernel. In contrast, in a common GPGPU program, the instructions on data are written in a shader program rather than being enclosed in a loop. The multi-stream processors of a GPU will act as computational workhorses to perform the kernel computations on data streams. The programmable parts of the GPU, vertex and pixel/fragment shaders, consist of a number of parallel processing units. In most of the reported GPGPU implementation cases, unlike the theoretical usage of vertex shaders for computation, fragment programs are more commonly employed for the “loop” cases since they provide more parallel channels. All the information exposed about the process is the "address" -- the texture coordinates in which the components of data streams reside. Therefore all works are carried out in parallel without any data interdependence.

3.2 GPGPU Task Computing

To implement a task on the pre-2006 GPUs (the latest model will be discussed in Chapter 4 and 7), it is essential to invoke the instructions from a specific graphics API (i.e., DirectX or OpenGL) to access the fragment program that is written in a particular shading language. The orders of the process are as follows:

1. The application task is analysed and divided into independent processing elements. Each element is mapped to a kernel and being abstracted as a fragment program with one or more data streams as the program's input and output. The input and output data streams are normally reside in a GPU's texture memory. The instructions in a kernel can then be implemented on each data component in parallel.

2. To activate a kernel at runtime, the computational range (or the area of the "output stream") need to be specified. In a typical GPGPU implementation, this can be issued through drawing a quad by invoking API instructions such as *glQuad*() and *glVertex2f* where *glQuad*() is for drawing a quadrilateral rectangle on the image plane. The size of the quad is specified by the instruction *glVertex2f*() (Pharr et al., 2005).

3. The rasterizer, 3D-to-2D transformer, then creates fragments for every pixel located in the quad, producing enormous large amount of 2D fragments. The next step will see each fragment being processed by the activated kernel programs.

4. The kernel programs access the arbitrary locations in a GPU's memory through predetermined coordinates of the texture that stores the actual data. In another expression, the computational domain will be specified in the input texture through configuring texture coordinates, which is followed by the process of drawing a quad on the image plane.

5. The output of a kernel program, in contrast to the multi-data vertex shader output, is a vector of values, often used in conventional graphical application as colours. It can be the ultimate results of an application, or the intermediate results stored as a texture in the framebuffer to be used by other kernel programs. Furthermore, some practical applications involve a series of passes ("multipass"), re-visit to the pipeline, to complete.

Based on the above operational break-downs, it is clear that the key to a successful GPGPU application is the usage of textures, thus the investigation in this part of the research is on how to access textures, read and write the GPU's memory. It is well-known that the CPU program often access complex data through pointers, however, the pointer is not supported by the fragment program. The actual read and write operations to access GPU memory is rather indirect, which are referred as scatter and gather.

As defined by Owens et al. (Owens et al., 2007), a scatter process is equivalent to the operation in C-like language in the form of: $x[i] = z$. In contrast, a gather operation is equivalent to the C expression $z = x[i]$. In other words, the gather operation actually corresponds to GPU's texture fetch which is further influenced by the projection style in the pipeline and the specified rendering area where a quad is drawn. In this project, textures are accessed by using OpenGL functions such as *glTexCoord2f*(), *glMultiTexCoord2f*() (Microsoft, 2006), and *glVertex2f*(), the instruction *tex2D*() or *texRECT*().

In contrast, the scatter operation can not be directly implemented as the gather operation since all fragment addresses in the frame buffer can not be explicitly expressed. The solution for this problem is through either using a specific program to classify the location of a given fragment in the framebuffer, or using another texture to perform the write operation. However, such solutions can not be supported by the pre-2006 GPUs. GPGPU programmers at the time had to make use of various programming techniques to alleviate this problem. In OpenGL, these techniques include binding a texture to a Pixel buffer (Pbuffer) or

a Frame Buffer Object (FBO). The data in the texture can therefore be updated through rendering the Pbuffer or FBO. Therefore, in many GPGPU programs, the scatter operation is also referred as “Render-to-Texture”.

As stated earlier in the section that a GPGPU task needs to be manually divided into several independent parallel sections before kernel definition can start in the form of shader programming. As a normal practice, each input parameter and output/return variable will be assigned with a data type, as well as a specified semantic symbol to identify a particular parameter’s state. List 3.1 shows a general form of a kernel in the shading language Cg.

```
float minimum ( float2 left_top:                TEXCOORD0,
                float2  right_top:              TEXCOORD1,
                float2 left_bottom:             TEXCOORD2,
                float2 right_bottom:            TEXCOORD3,
                uniform samplerRECT tex0: TEXUNIT0,
                uniform samplerRECT tex1: TEXUNIT1 ) : COLOR
{
        // Assigning input values to temporary variables
        // Actual process takes place  at here
        // Return the result or result flag
}
```

List 3.1 Parameter’s semantic binding in a kernel

In the kernel that named as *minimum*, parameters $tex0$ and $tex1$ are all bound to the semantic TEXUNIT to identify their nature as textures, parameters *left_top*, *right_top*, *left_bottom*, and *right_bottom* are all bound to the semantic TEXCOORD to identify that they are texture coordinates used for texture fetch operations. According to the definition in Cg, the output variable must be bound with type COLOR in accordance with the pixel display. If multiple parameters are bound with the same variety of semantic symbol, they will be differentiated by various index such as TEXUNIT0, TEXUNIT1, and TEXCOORD0, TEXCOORD1. For multi-pass GPGPU application, the so-called ping-pong manner is adopted for the transform between the texture read and write modes. Therefore, if a texture is

used for read-only in the current rendering pass, then it will be employed for write-only process in the next rendering pass, and then transformed back again.

To execute a fragment program on a GPU, it should be first loaded and activated by the graphics API instructions. For example, the OpenGL instructions for loading a Cg program are *cgCreateProgram*() and *cgGLLoadProgram*() ; while the instructions for OpenGL activating a Cg program is *cgGLBindProgram*() . After the activation, the fragment program is issued on the GPU through the instruction of rendering a suitable geometry, usually by drawing a quad. The operation of drawing a geometry will generate fragments from the input geometry through the rasterizer. These fragments become output pixels after processing by fragment program (Shirley, 2005). Before drawing a geometry to trigger the fragment program, the essential initialization for operations on transformation matrix, such as the matrix mode specification, must be implemented. Following the matrix mode specification, the area of corresponding viewport must be configured as well, which determines the maximum output region at different stages in the rendering pipeline. These essential initialization steps are shown in List 3.2.

```
Set the projection matrix stack as the target of all matrix operations;
Replace the current matrix with the identity matrix;
Specify the viewport of orthographic projection, which follows the operations on projection matrix stack;
Set the modelview matrix stack as the target of all matrix operations;
Replace the current matrix with the identity matrix;
Specify the viewport on computer's screen, which follows the operations on modelview matrix stack;
```

List 3.2 Configuration for transformation matrix mode

After the initialization, the actual running of the fragment program is triggered by the process of drawing a quad which also specifies the computation range and the texture fetch scope in the fragment program. It is achieved by setting up the

x- and y- coordinates of the four corners of the drawn quad and the specific texture, as shown in List 3.3.

The input vertices and the vertex shader determine which group of pixels are generated. Through specifying four vertices coordinates of a quad issued by the OpenGL instruction $glVertex2f()$, the output range of the computation is directly under control. As shown in List 3.3, individual texture-pixel (texel) is sampled according to an 1:1 or 1:X mapping proportion between pixels and texels. A simple example is to find the maximum/minimum value in a n-element vector by the “sort” computation of “parallel reduction”, which is to be introduced in the following section.

```
Invoke the API instruction, such as glBegin(GL_QUADS), to draw a quad ;
Specify the x- and y- coordinates of the left-top point on a texture mapped to the quad;
Specify the x- and y- coordinates of the left-top vertex on the primitive in the form of  quad;
Specify the x- and y- coordinates of the right-top point on the texture;
Specify the x- and y- coordinates of the right-top vertex on the quad;
Specify the x- and y- coordinates of the right-bottom point on the texture;
Specify the x- and y- coordinates of the right-bottom vertex on the quad;
Specify the x- and y- coordinates of the left-bottom point on the texture;
Specify the x- and y- coordinates of the left-bottom vertex on the quad;
```

List 3.3 Specification of computation range and texture fetch in a fragment program

3.3 Render-to-Texture

For the GPU-based applications requiring multiple rendering passes, data stored in the texture memory can be updated in the process of loops. However, as discussed in previous sections, legacy GPUs don’t allow direct scatter operation on texture memories. The refreshment of texture memory can be archived by

techniques such as rendering to texture in which a texture is bounded with a Pixel Buffer (PBuffer) or a Frame Buffer Object (FBO) and then being updated by rendering to the PBbuffer or the FBO. Both medias are the so-called off-screen buffers that are accessible by specific OpenGL functions (Oat, 2005; Persson, 2007). These temp-storage mechanisms allow programmers to generate complex procedural images in video memory that can then be in turn bound as textures or even being read back into CPU's memory. It has resulted in the extensive applications of the PBuffer and FBO in almost all GPGPU pilot projects before 2006.

For the sake of their flexibility and adaptability, as well as their implications on the PC-grade parallel processing frameworks (Chapter 4), the PBuffer's and the FBO's usage are briefly explained at here. PBuffers are implemented as a Windows Graphics Library extension on the Microsoft Windows OS. The usage of PBuffer includes the following steps (Oat, 2005):

- PBuffer setup and initialization;
  The creation of PBuffer can be issued by the OpenGL instruction *glGenBuffers*(). This instruction creates either a single PBuffer or multiple buffers in the form of arrays. Since every OpenGL-based GPGPU application has at least one object called a GL context which involves device context and render context, creating a PBuffer requires the programmer to have good knowledge on the current device context and render context associated with the application.

- Rendering to the PBuffer;
  When rendering to the PBuffer begins, the program need to specify that any frame buffer operation is now targeted to the created PBuffer. In the same way, a PBuffer can be used as a data source for any read commands such as *glReadPixels*() by setting the current context. Once a PBuffer is set as the current "write" context, the program will render the results to this PBuffer.

Once the rendering is finished, the “write” context should be returned back to the frame buffer.

- Binding the PBuffer as a texture;
  If a PBuffer is bound with a texture object, rendering with a PBuffer bound to a texture object is exactly the same as rendering with a normal texture object.

- Freeing the PBuffer.
  It is a common practice as a PBuffer finished its task, it will be dynamically destroyed so that the memory associated with this PBuffer will be returned to the computing platform for other processes. In this case, the PBuffer will first be confirmed to decouple with the current texture by calling the instruction *wglReleaseTexImageARB*(). Then, the render context associated with the PBuffer will be deleted and the corresponding device context released. Finally, the PBuffer is deleted by calling *wglDestroyPbufferARB*().

PBuffer also exposed some problems in the practical applications, for example, each PBuffer requires a unique GL context (Persson, 2007). Thus the application has to keep the track of all PBuffer states, which brings a heavy workload to the processor since the operation of context switching is time-consuming, especially when there are multiple PBuffers being employed by an application. In addition, each PBuffer has its own color, depth and stencil buffers that are “internal” only. These problems are caused by the hardware design and graphical-oriented idealism. To overcome those problems, the Framebuffer Object (FBO) has been introduced in GPGPU applications. Comparing with the PBuffer-based approach for enabling intermediate result storage, the FBO has the following features:

- A FBO can be integrated directly with a regular texture
- Multiple FBOs can share the same GL context
- Independent from operating systems
- Rendering to the target device without needing a colour buffer
- Allow sharing across depth, stencil and colour buffers

The usage of the FBOs has the same process flow as the PBuffers, including:

- Create an FBO;
- Attach the colour buffer or the depth buffer to a texture;
- Render the texture with a fragment shader;
- Freeing the FBO.

The detailed instruction sets operating on a FBO will not be discussed here. It is noted that a single FBO can bind with multiple textures. The number of bound texture is determined by the number of colour buffer and depth buffer that the GPU hardware can obtain (Persson, 2007). For example, the Nvidia GeForce 7800 GPU can obtain 4 colour buffer and 2 depth buffer, so that it can bind with 6 textures in total, among them, the 4 textures can be bound to the colour buffers by the following OpenGL instructions.

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT, GL_TARGET, texture[0], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT, GL_TARGET, texture[1], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT2_EXT, GL_TARGET, texture[2], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT3_EXT, GL_TARGET, texture[3], 0);
```

Where $GL\_COLOR\_ATTACHMENT$ is the parameter that represents colour buffer. For example, if $texture[3]$ will be rendered to device, the write target will first be specified with the instruction $glDrawBuffer(GL\_COLOR\_ATTACHMENT3\_EXT)$. Then the next step is to invoke the fragment shader to render the colour buffer with index 3. The operations on the depth buffer are the same with those for colour buffer.

3.4 Embedded Parallelism in GPGPU

So far this thesis has provided a general overview to the GPU hardware and the GPGPU concept with the intention to convert the computational problems into the notations of computer graphics to issue parallel computing on GPU. Generally

speaking, a GPGPU task is carried out by the stream programming model equipped on all modern GPUs, in which data are represented as streams and the computations on them are performed by kernels. Comparing with CPU programming, GPGPU has two key features that need to be recognized – evolving dynamic flow control and vector-based data structures which are largely determined by the chosen GPU's hardware characteristics. For the convenience of further discussion, it is essential to first map the graphics pipeline to the stream programming model, then these two features will be analysed focusing on the programming model.

3.4.1 The Stream Programming Model

As stated in Section 3.1, all data running through a modern GPU can be represented as streams which are either input or output of a kernel program. The data types for streams can be of the simple ones such as integer or floating-point number or the more complex ones such as triangles or transformation matrices. Since the processing on separate stream elements within a kernel is independent, it guarantees the feasibility of mapping a series of kernel calculations onto a data-parallel hardware, i.e., an application can be realized by cascading several kernels together. The aforementioned Render-to-Texture has provided a mechanism to store the intermediate results in GPU's memory for the chaining process.

In fact, resources along the graphics pipelines are a good match for this cascaded structure in the GPGPU programming. The creation of a graphics imagery on computer's display involves the whole processes of developing a vertex program kernel, a triangle assembly rasterizer kernel, a clipping kernel, and then forwarding the output to the pixel kernel. Figure 3.3 shows the entire graphics pipeline being mapped onto the stream model. The arrows represent the transient stages making the communication between kernels explicit. This in turn ensures the data locality between kernels inherent in the graphics pipeline.

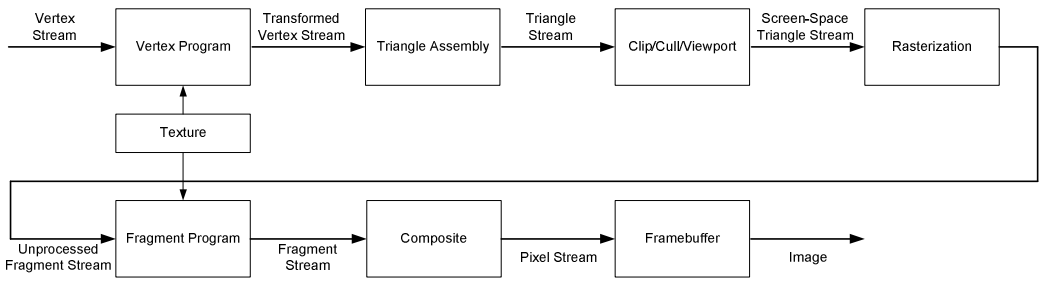


Figure 3.3 GPGPU's Stream Model (Courtesy to Shirley)

In a typical setting of GPUs between 2001 and 2006, the texture stream, vertex streams, and framebuffer streams are accessible to GPU programmers through assembly-style or high level shading languages (HLSL). Figure 3.4 highlights these three streams and their relationships. As shown in Fig. 3.3 and Fig. 3.4, vertex streams are stored in vertex buffers and being used as input streams for vertex programs. When shifted from the CPU main memory, it holds a list of vertex positions and a variety of per-vertex attributes such as colours, normals, and texture coordinates. Early graphics APIs did not allow GPUs to write to vertex streams directly (Owens et al., 2007), which had brought the problem of heavy overhead to the following fragment shaders due to the extra diversion needed for storing the intermediate results produced by a vertex shader to be stored at the rear-end graphics memory, the frame buffer.

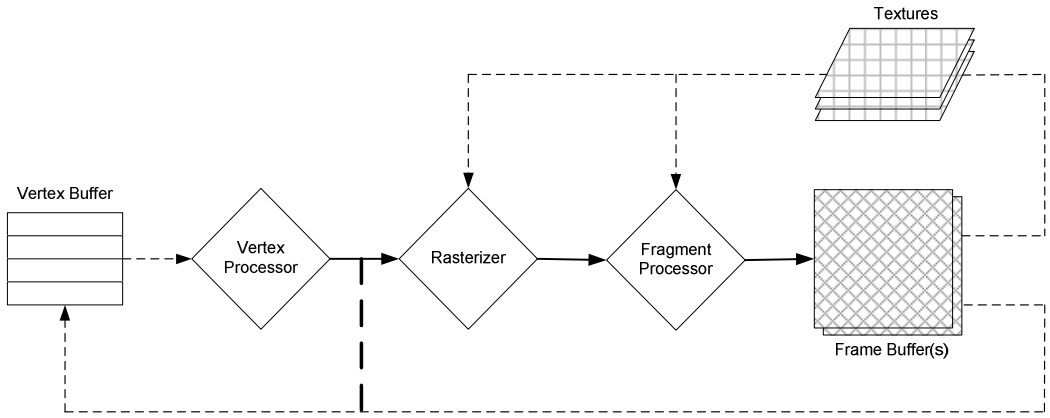


Figure 3.4 Streams in GPUs (Courtesy to Owens et al.)

As a relative recent development, the programming standards, Shader Model 3 and 4, have made it possible for the GPU internal processes to write to vertex streams, as indicated by the bold dot line in Figure 3.4. In general graphical application developers' eyes, this operation can be accomplished by either "copy-to-vertex-buffer" or "render-to-vertex-buffer" (Dally et al., 2004). These two terms for writing to vertex streams are actually achieved by the so-called Vertex Buffer Object (VBO) in Integrated Circuit (IC) chip designers' views, which will be further discussed in Section 7.4.4 through a case study on parallel data processing in the Optical Spectral Scanning Interferometry system. In addition, many significant simulations on the vertex displacement have been carried out by jointly using the VBO and the method of vertex texture fetch (VTF). For example, Losasso Frank and Hugues Hoppe of Microsoft Research have used them for a highly efficient terrain-rendering algorithm (Losasso and Hugues, 2004) which avoided the overloading of the GPU even as it shifts most of the repetitive and recursive work onto the GPU. Hagen and Hjelmervik have used the VBO and the VTF together to perform texture fetches at the vertices of a complex mesh to perform true displacement mapping on the water surface (Hagen et al., 2005).

The Frame-buffer streams are written by the fragment processor. As a long-lasting graphics resource (back buffer) comparing with the other two, they have traditionally been used to hold pixels for display onto the screen. The modern GPU design has seen frame buffers being used to hold the intermediate results from the pixel shader in multiple-pass rendering, which has been explained in the technique of Rendering-to-Texture in Section 3.4.

The texture streams are stored as arrays of texture properties in the graphics memory. Before the release of the GPUs with unified pipeline structure such as Nvidia's GeFoece 8, 9, and GTX 200 series, textures are the only GPU memory that is randomly accessible by fragment programs and vertex programs. If application programmers need to randomly index into a vertex or frame-buffer stream, the data must be first converted into a texture. For the convenience of

data access, textures in GPU can be declared as 1D, 2D, or 3D streams and being addressed with a 1D, 2D, or 3D address.

Corresponding to the stream programming model as introduced above, two common forms in CPU programming, flow control and data structure, have also appeared in GPGPU. However, their implementation are very different from their counterparts on CPU.

3.4.2 Flow Control

It is well-known that the flow control is essential and vital in modern programming. It can be presented in the graph form of branching and looping corresponding to the if-then-else, for, and while instructions in serial programming models. Legacy GPUs did not have native branching of this form, so other strategies were adopted to emulate these operations, which increased the complexity to GPGPU. The latest GPUs, from the releasing of NVIDIA GeForce 6 Series, have supported branching in vertex and fragment programs (Nvidia Corporation, 2009). Their native features of graphical functions can be used for non-graphical applications to maintain the speed-up performance. After all, GPU is intrinsically a SIMD processor and within a SIMD group, if multiple operations evaluate the branch conditions differently, then all branches must be evaluated carefully to avoid deteriorating performance caused by variant branching conditions on different data block in a stream when invoking a kernel program (Tomov et al., 2005). Therefore, other strategies and techniques need to be devised to reduce the cost of branching on GPUs. Most of the reported attempts have been focusing on a common strategy -- moving the evaluation of branch conditions outside the graphics pipeline, or even cross the GPU/CPU boundary.

1. Static Branch Resolution

The aim of adopting the static branch resolution is to avoid the expensive branching operation inside of the inner loops within the vertex or fragment programs, which normally requires the division of a stream into substreams. For

example, a task computation is divided into several sub-computations through accumulative fragment programs, but actually just one brand is issued simultaneously when the specific logic condition is satisfied. A classical application of the static branch resolution is the solving of a partial differential equation (PDE) on a discrete spatial grid (Krüger and Westermann, 2003). In that application, the whole GPGPU solution for this type of problems is mainly comprised of two fragment programs: one is used for processing the interior cells of the grid, while the other one working on the boundary edges. Therefore, the computational range of the fragment program for processing interior cells will exclude the outer one-pixel edge when drawing the quad; while the range for processing boundary cells will just include the outer single-pixel edge when drawing the quad.

In general, it seems that there is no any branching operation in the GPGPU solution that uses the static branch resolution. In fact, most of the surveyed GPGPU solutions were decomposed into several vertex or fragment programs which had the same or different kernels with variant compute ranges, which were allocated manually. The implementation order of those vertex and fragment programs have also been specified through graphics API instructions in advance. That is why the term “static” has been used in the name of the style.

Based on the above observation and analysis, it is concluded that the suitable occasion for employing static branch’s is when the operations employed by each branch corresponding to the constant condition over the complete input domain. From the view of computer graphics, this is certainly the case when an application will ultimately form a fixed image on the screen after the computation. Except solving a PDE, many linear algebraic operations can be classified into this category (Hagen et al., 2005).

2. Pre-computation

Pre-computation is often used in the scenario when the results of each branch is a constant in a fixed period of time or a number of iterations of a computation

(Owens et al., 2007). Once the results are subject to change, the operations corresponding to the branch evaluation will be triggered and the intermediate results are stored for use over subsequent iterations.

The GPGPU-based fluid simulation reported by Zhao et al. (Zhao et al., 2006) has extensively utilized this technique to avoid branching when computing boundary conditions at the edges of arbitrary obstacles in the flow field. In this setting, fluid cells with no neighbouring obstacles can be processed normally, but cells with neighbouring obstacles require more complex processes, for instance, these cells must check their neighbours to figure out in which direction the obstacle lies before using the directions to look up more data to be used in the computation. This operation for obstacle change will be implemented only when the user program “draws” them. Therefore, the offset directions can be pre-computed and be stored in an offset texture to be reused when the user changes the obstacles again.

From the view of computer graphics, the occasions that the pre-computation technique can be adopted corresponding to the applications in which the image on the screen changes slowly, that is, if divided into smaller intervals, the image is basically fixed or changed very little so that the change will not be aware of by human’s eye.

It is clear that there is no explicit branching operation in existing GPU instructions to control the stream flow in the forms of static branch or pre-computation. The compromised solutions were brought in by shielding branching through manual interference. For the simulations involving rapid particle movements such as collision, a Z-Cull solution was made available thanks to the hardware evaluation of modern GPUs.

3. Z-Cull

Z-cull is a technology employed by modern GPUs in the stage of pixel processing to determine a pixel’s visibility. The letter Z refers to the Z axis of the 3D viewing

space (Mitchell and Sander, 2004). The principle of Z-cull is of comparing the depth value (Z) of an input block of fragments with the depth values of the corresponding block of fragments stored in the Zbuffer (see Section 2.3.2). Only those fragments that pass through depth test will be further processed by pixel shader to form their pixel colour. In contrast, those fragments failed on the depth test will be discarded before the process of pixel shader. Therefore, the valuable GPU processing capacity saved.

Referring to the particle simulation discussed above, the current pressure status of a particle is first determined before the subsequent computation, which is then performed by pre-evaluating this particle's neighbours' pressing on it. If the particle receives all the press from its top, bottom, left-hand, and right-hand neighbours, then this particle will be treated as in the "balance" status. So that it can be ignored when computing all particle's movement direction in the next time slot. In this case, the "balance" status of this particle will be pre-marked as failing in the depth test in the Z buffer. This design has ensured "failed" fragments are directly discarded when the fragments are calculated by the fragment processor (Simon et al., 2007; Liu et al., 2008). Suppose the particle receiving the pressure from one direction can be represented by 1 (otherwise by 0), the process of pre-evaluation can be issued by a fragment program in the first pass, as depicted below:

```
Kernel Pre-evaluation()
{    Set a vector named as marker with four components – x, y ,z and w;
     marker.x = the pressure from the top neighbour;
     marker.y = the pressure from the bottom neighbour;
     marker.z = the pressure from the left-hand neighbour;
     marker.w = the pressure from the right-hand neighbour;
     Add the x, y, z, w value of marker up;
     If the sum is equal to 4
     then set the value of depth test in depth buffer as failure;
}
```

Figure 3.5 The configuration for Z-Cull in the first pass

After Z-Cull is set up, the whole process of particle simulation is simply demonstrated as follow:

```
Initialize the depth buffer and enable the depth test;
Run the kernel of Pre-evaluation in the first pass to set up Z-Cull;
Run the fragment program to compute the particles movement direction in the
subsequent pass;
```

Figure 3.6 The process of particle simulation using Z-Cull

In the particle simulation carried out by Li (Li, 2004), if the particles in the "balance" status are fairly large, then much work can be saved by passing processes for those particles. Therefore, the Z-Cull is a powerful method for skipping unnecessary work based on the hardware features of GPUs. Although the "if-then" style instruction has been introduced in vertex and fragment programs from the releasing of NVIDIA's GeForce 6 series and the shader model 3.0, there are researches demonstrate that the Z-Cull is more efficient than directly issuing conditional instructions in shader languages when implementing complex computations involving large computational ranges (Han et al., 2005; Xie et al., 2008).

3.4.3 Data Structure

In CPU programming, the basic data structure is based on multi-dimensional array. The memory address can be accessed easily by using pointer or directly indexing the array's coordinates along various dimension. In contrast, the 4-D vector style texture memory is the dominant form of local memory in GPU programming. Although there are various formats such as 1D and 3D textures, the physics memory of GPU is actually of the 2D texture. Restricted by the capacity of 1D textures and the number of slices in 3D textures that can be accessed when issuing a rendering pass, there are dimensional conversions such as 1D-to-2D and 3D-to-2D in GPGPU when storing various dimensional

array into a 2D texture. In GPGPU programming, data access is implemented through indexing the texture coordinates, therefore the key task for describing the data structure in GPGPU is to index a texel through the memory address translation process (Owens et al., 2007).

1. Address translation from 1D array to 2D texture

Data from a 1D array can be stored in a 2D texture by packing the data into that texture, as shown in Figure 3.7.

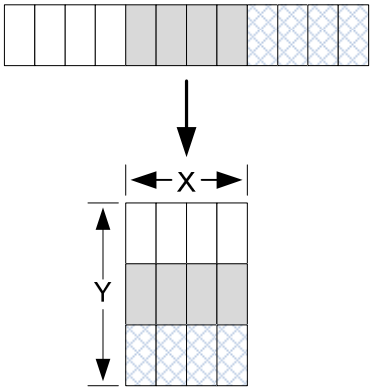


Figure 3.7 1D array packed into 2D textures

When the packed array is accessed from a vertex or fragment program, the 1D array address must be converted to the 2D texture coordinates, which can be implemented by a fragment program. For example, suppose an array size is *N*, which means the 1D array address denoted by an integer variable $1D\_Addr$ is within the range [0, *N*). If the texture size is *X* and *Y* along x- and y- direction, then the 2D texture coordinates, denoted by variable $tex\_cord.x$ and $tex\_cord.y$ can be computed by the following equations that are issued by a kernel embedded in a fragment program.

$$tex\_cord.x = 1D\_Addr \,\%\, X \qquad (3\text{-}1)$$

$$tex\_cord.y = 1D\_Addr \,/\, X \qquad (3\text{-}2)$$

2. Address translation from 3D array to 2D texture

According to modern GPU's data architecture design, a 3D array may be stored in one of the three ways: in a 3D texture (with each slice stored in a separate 2D texture), packed into a series of 2D texture, or packed into a single 2D texture (Lefohn et al., 2006).

In the first case, no address translation is required because the x, y, z components of texture coordinates can be directly indexed. It seems very convenient but the problem is that GPUs can only update limited slices of the volume per rendering pass – thus might requiring many passes to write to the entire array (Lefohn et al., 2004).

In the second case, as shown in Figure 3.8, multiple 2D textures can be updated through binding the textures with different colour buffers and depth buffers, as discussed in Section 3.3.

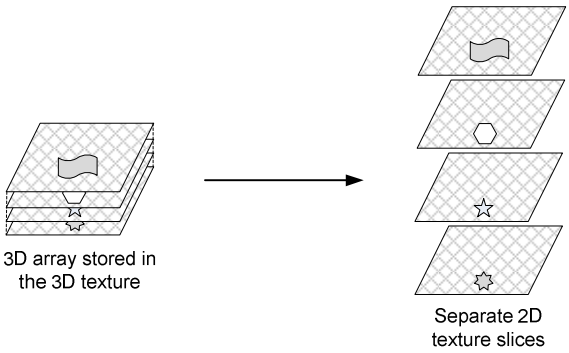


Figure 3.8 Storing a 3D array with separate 2D slices

Based on Owen's review (Owens et al., 2007), the problem of this approach is that the volume can no longer be truly randomly accessed because each slice is a separate texture stored at separate addresses. The programmer must know the exact slice numbers to access before the kernel execution since the fragment and vertex programs cannot dynamically compute which texture to access at runtime. As a result, the process of address translation between 3D array and 2D texture is needed to solve the problem, where two scenarios must be considered. It is known the common representation of a 3D array is $a[x][y][z]$ that

demonstrates the 3D array's coordinates on x-y plane and in z axis respectively. If the texture size along x- and y- dimension is same as the size of a 3D array on x-y plane, the address translation is comparatively simple. The programmer can acquire the indices of slices through the z component. However, the possibility that the size of texture in 2D texture slices is different from the 3D array's size on the x-y plane must be considered. Suppose the 3D array's size on x-y plane is denoted by X_{3D} and Y_{3D} respectively, the texture size is X_{2D} and Y_{2D}, for element $a[x][y][z]$ in 3D array, the address translation to one of the 2D texture slices can be implemented by a fragment program that demonstrate the algorithm as follow (Owens et al., 2007):

Step 1: Express the index of $a[x][y][z]$ in the 1D array's index form – $Index_{1D}$

$$Index_{1D} = z \cdot X_{3D} \cdot Y_{3D} + y \cdot X_{3D} + x$$

Step 2: Determine $a[x][y][z]$ located in which slice number that is denoted by $SliceNumber$

$$SliceNumber = Index_{1D} / (X_{3D} \cdot Y_{3D})$$

Step 3: Determine the index of $a[x][y][z]$ in the selected texture, the index is denoted by $Temp\_Index$

$$Temp\_Index = Index_{1D} \% (X_{3D} \cdot Y_{3D})$$

Step 4: Determine the x and y coordinates in the selected texture according to $Temp\_Index$

$$y\ coordinate = Temp\_Index / X_{2D}$$

$$x\ coordinate = Temp\_Index \% X_{2D}$$

Now consider the last case of 3D array being packed into a single 2D texture. It is actually a special case of the 3D array being stored in slices on separate 2D textures. In this case, once the texture size is large enough, just one texture is needed other than several slices for the storage. Therefore, the address translation of 3D array to slices of separate 2D textures is similar to that of 3D array being packed into a single 2D texture with an amendment of erasing the Step 2 of the aforementioned algorithm.

3.5 Optimization of GPGPU in Linear Arithmetic Operations

As stated in Section 3.1, the GPGPU programme is consisted of two key elements – stream and kernel that distinguish the CPU programming in the term of SIMD. Like in CPU programming, there also exists the demand of program optimization in GPGPU. The principle for optimizing the kernel is to alleviate unnecessary operations or instruction calls in vertex and fragment programs through using techniques such as the Z-Cull to issue the branch operations. The key for optimizing stream is to erase the non-essential data sets in stream to reduce the size of kernel's input and easing GPU's memory burden.

It is widely accepted that linear algebra is the basis of almost all mathematical applications, in which the data (streams) are often consisted of vectors that corresponds to various dimensional 1D arrays and matrices. The basic operations on the data (kernels) are normally composed of arithmetic and bit shifting operations. When there are a large number of zero components in the vectors and matrices involved in a multiplication operation, a matrix can be categorized as the dense matrix in which there are little zero value or the sparse matrix possessing a large number of zeros. How to efficiently store a sparse matrix in the GPU memory and to implement it on the optimized matrix or vector is a vital issue in linear algebraic based GPGPU applications.

3.5.1 Representation of Banded Sparse Matrices

The so-called banded sparse matrices is the sparse matrices that exhibit a regular pattern of nonzero elements, such as diagonal matrices, upper-triangle matrices, and lower-triangle matrices (Kincaid and Cheney, 2002). A banded sparse matrix can be depicted as follow.

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 \\ 0 & a_{32} & a_{33} & a_{34} & \cdots & 0 \\ 0 & 0 & a_{43} & a_{44} & \cdots & 0 \\ \vdots & & \ddots & & \ddots & \\ 0 & \cdots & 0 & a_{nn-1} & a_{nn} & \end{bmatrix}$$

Figure 3.9 A banded sparse matrix

When storing the large size diagonal matrices or the banded sparse matrices that have the analogous structure like the former on GPU, an effective style is to first transform the diagonal elements into several vector formats, and then packing the transformed vector into various 2D textures. Following this pattern, the storage of the matrices as shown in Figure 3.9 can be processed in the style as shown in Figure 3.10.

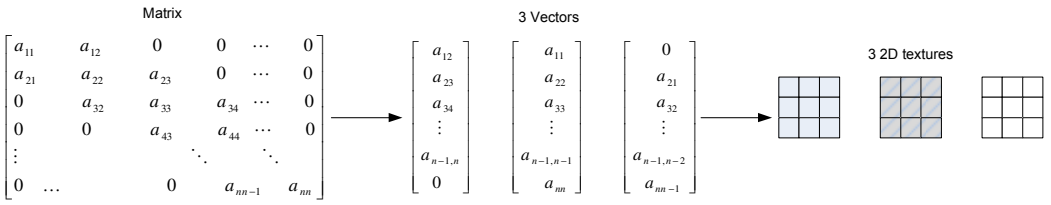


Figure 3.10 Store a banded sparse matrix on the GPU

Considering there are some zeros added in the diagonal vector to fit the texture size, the more efficient storage can then be implemented by combining the two opposing diagonals into one vector. It means a full matrix in the diagonal format can be stored in several textures without wasting a single byte of these textures' space. An example of this case is demonstrated by Krüger and Westermann (Krüger and Westermann, 2003) as shown in Figure 3.11.

As defined in classical linear algebra, the multiplication between a matrix and a vector can be expressed as follow

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & & & \vdots \\ \vdots & & \ddots & \\ a_{n1} & \cdots & & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad \Rightarrow \quad y_i = \sum_{j=1}^{n} a_{ij} b_j \qquad (3\text{-}3)$$

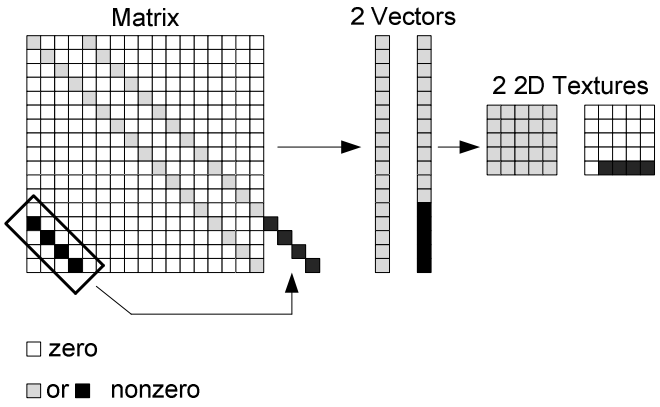


Figure 3.11 Pack more nonzero into diagonal vector

If a matrix is packed into a 2D texture directly, then the row and column indices of the matrix elements can directly correspond to the texel's coordinates, so there is no need for any coordinate transformation in the kernel. However, if using the diagonal format to represent a matrix, it is unavoidable to have the coordinate transformation in the kernel to guarantee the multiplication between the correct elements in the matrix and the vector. The coordinate transformation is comparatively simple in this case because it only need some regular shifts on the x and y texture coordinates to ensure the regular distribution of the diagonal vector in the banded sparse matrix.

3.5.2 Optimized Implementation on Random Sparse Matrix

For random sparse matrices, a small quantity of nonzero elements are scattered randomly in the matrices. When issuing the matrix-vector product as indicated by Equation (3-3), how to establish the relation of the row and column indices between the texture coordinates is much more sophisticated than that in the case

of the banded sparse matrix-vector product. Krüger and Westermann (Krüger and Westermann, 2003) have devised an efficient encoding method to solve this problem.

A close inspection on Equation (3-3) can reveal that the row index i influences the final position of the result y_i, while the column index j specifies what values of the vector X are to be combined with a_{ij}. This pattern has stimulated the thought to use the vertex to include the information of y_i, for example, employing the vertex position to encode the row index. When rendering a vertex, the indexed can be bound with multiple texture coordinates, then the column index is encoded in these texture coordinates. The $XYZW$ components of a texture coordinate are all float-point type and can be manually set by the programmer to indicate the coordinates, thus a special texture coordinate can be specified to contain the value of nonzero entries in the matrix by using its $XYZW$ components. Based on this encoding principle, a series of vertex array need to be created in which the row index is included in the vertex position, and the column index and the value of the nonzero elements are included in the several texture coordinates that are bound to a vertex, as sketched in Figure 3.12. Every four nonzero elements that are in the same row are grouped. If there are several groups, these groups are stored in various vertex arrays, therefore a series of vertex arrays are created to store the vertices that have the same position.

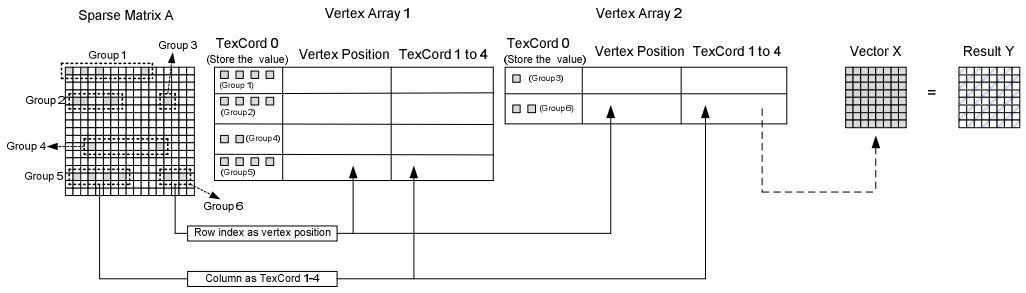


Figure 3.12 Encode to the nonzero element in the random sparse matrix (Courtesy to Krüger and Westermann)

After the encoding on the random sparse matrix, its production with a vector can be issued by rendering the vertices, which is actually implemented by a vertex or fragment program that involves multiple rendering passes. Each single pass is to render a vertex array and each pass actually does part of the multiplication as indicated in Equation (3-3), the results of each pass are stored in the vertex or frame buffers as the intermediate value. After all the rendering passes have been implemented, all the intermediate values can be added up to obtain the ultimate result, i.e., vector Y in Equation (3-3). It is noted that the number of rendering passes is determined by the row that has the most nonzero elements in its sparse matrix. Suppose the most nonzero elements in that row is m, then the number of rendering pass in the vertex/fragment pass is equivalent to $\lceil m/4 \rceil$.

3.5.3 Further Discussion

From the above discussions, it can be seen that the optimization measures of GPGPU-based linear algebraic operations have been focusing on the improvement of GPU's memory usage. Hence it is of great values for applications, such as meteorological data processing, in which some matrices have enormous sizes and are difficult to be directly packed into the GPU memory such as 2D textures. The above encoding techniques can enable and accelerate the product operations on those matrices performed on GPUs. In the mean time, it is also observed in the review that on the current solutions the GPGPU strategies and implementations actually require a researcher in an engineering field be able to map the domain knowledge onto graphics concepts, such as vertex array establishment, fragment specification, texture coordinates binding, intermediate data storage in vertex/frame buffer, as well as multi-passes and rendering.

Most of the above discussed "optimized" GPGPU strategies seem pointing to the direction of larger GPU memory size, however, the uncontrolled increase of GPU memory can also lead to the increased cost on rendering passes in a GPGPU

application. Since multiple rendering passes are implemented in serial, it unavoidably brings the negative impact on GPU's acceleration performance. From this point, it can be argued that a balance need to be struck on the efficient use of GPU's existing memory and the complexity of the vertex and fragment programs.

3.6 Process Decomposition in Parallel Computing

The aforementioned Flynn taxonomy (see Section 2.1) categorizes the computing architectures as of the SISD, MISD, SIMD, and MIMD models based on the relationships of instruction and data streams. No matter which architectural pattern is eventually being adopted for a parallel programming task, the first job of the development cycle is always to decompose the input and process specifications to achieve parallelism. In general, three methods for decomposing programs for potential parallelism are summarized based on a large spectrum of existing approaches, functional decomposition, domain decomposition, and activity decomposition (Foster, 1995; Carrieroand and Gelernter, 1988). These methods are conceived from the partitioning policy for data and/or algorithms through establishing three different forms to articulate parallelism, and, henceforth, to design parallel programs.

Each one of these methods can be categorized in terms of the characteristics of a parallel pattern as described below:

1. Functional decomposition. It is also known as task decomposition or specialist decomposition, which focuses on the decomposition of the algorithm (Foster, 1995; Carrieroand and Gelernter, 1988; Chandy and Taylor, 1992; Pancake, 1996). Its objective is to divide the algorithm into discrete tasks, which are capable of being executed simultaneously. Once being divided into separate tasks, the data requirements of each task (input data and output data) will be examined. If the data requirements for each task is also discrete, then process divisions can be formed. In contrast, if the data requirements overlap

significantly, then intensive communication becomes unavoidable to replicate data.

During the functional decomposition, all tasks will start simultaneously with most of them waiting for the arrival of data initially (Pancake, 1996). Different tasks may carry out different operations for accomplishing an algorithm as a whole. Once under way, different tasks will operate on different pieces of data in a discrete fashion. The main idea behind this is to allow the execution of tasks, even with overlapping, proceeding simultaneously (Carrieroand and Gelernter, 1988). Each task under this design is normally assigned to perform one specific type of operation, until the natural restrictions order and precedence imposed by the problem occur.

2. Domain decomposition. It is often referred as data decomposition attributing to its operations on decomposing the data associated with the problem (Foster, 1995; Carrieroand and Gelernter, 1988; Chandy and Taylor, 1992; Pancake, 1996). If applicable, the data will be divided into smaller pieces of approximately equal size. Then, the algorithm is divided through associating each task with the data it operates on. This divisional operation will yield a number of tasks, each comprised by some data and a set of operations on that data. At runtime, an operation may require data from several tasks and move data between tasks.
   Similar to functional decomposition, in domain decomposition, all tasks also start simultaneously up to the point until the work on a piece of data cannot proceed until another is finished (Carrieroand and Gelernter, 1988; Pancake, 1996).

3. Activity decomposition. Activity decomposition (also known as agenda decomposition) requires partitioning both the data and the algorithm (Carrieroand and Gelernter, 1988; Pancake, 1996). As a hybrid operation of the above two, different pieces of data are operated on by different tasks.

Each task can be considered as a “worker”, capable of grabbing some data and performing part of the algorithm on it before returning a result.

In activity parallelism, all tasks also start simultaneously as there is no special commitment to any part of the data. All tasks are able to operate independently (occasionally there can be a sequence of actions). However, tasks still need to coordinate when operating on a same piece of data to assemble a single and final result. In simple terms, each task will be assigned to pick a piece of data, operate on it, produce a result and repeat until the whole data has been processed.

The boundaries between these three models can sometime be blurred, and often, their elements are mixed in order to deal with a particular application. For example, a functional decomposition may use an activity decomposition operation at an intermediate phase. However, as pointed out by Carriero and Gelernter (Carrieroand and Gelernter, 1988), the above approaches represent distinctive way-of-thinking and problem-solving strategies. In the following sections, these three decomposition techniques will be re-assessed through classifying and selecting the architectural patterns for various parallel programming tasks.

3.7 Classification of Parallel Architectural Patterns

Except the decomposition criteria explained in Section 3.6, the nature of processing components can also be used for classifying parallel systems. It is clear that all components of a parallel system perform certain type of coordinating and processing activities. Parallel systems can therefore be classified as homogenous systems or heterogeneous systems according to the features of coordination among the processing components (Sinnen, 2007). A homogeneous system is consisted of components coordinated in the same rules based on the fact that they are processed in the same style. The operational switches among

these components need not any special communication mechanisms. In homogeneous systems, the communications among the components are issued through data exchange. In contrast, a heterogeneous system is consisted of multiple groups of components processed by various functions. The operations of the heterogeneous systems rely on the distinctions between different groups of components. Within the same group, the coordination between components is similar to homogeneous systems, i.e., employing data exchange. However, the coordination between components located in different groups must be through specialized communication mechanisms in the form of function calls.

Based on the two sets of classification criteria highlighted in Section 3.6 and Section 3.7, four general architectural patterns have been deduced by Goswami (Goswami et al., 2002) for parallel programming systems as: Divide-and-Conquer, Processor Farms, Pipes-and-Filters, and Communicating Sequential Elements. The relationship of those patterns and the aforementioned criteria can be in the form of Table 3.1.

Table 3.1 Architectural patterns classification

	Functional Decomposition	**Domain Decomposition**	**Activity Decomposition**
Heterogeneous processing	Pipes-and-Filters		
Homogeneous processing	Divide-and-Conquer	Communicating Sequential Elements	Processor Farms

These four architectural patterns have been deduced from the existing parallel systems that cover practices such as computer clusters, grid computing, GPU and game consoles, as well as pervasive and mobile applications.

Following subsections will briefly explain these architectural patterns, their relationship with the GPGPU idealism, and the perceived application domains.

3.7.1 Divide-and-Conquer

The Divide-and-Conquer pattern breaks a computational task into multiple sub-tasks that are similar to the original one but smaller in size. It then solves the sub-tasks recursively, and finally combines those solutions to create an overall solution to the original problem (Rabhi, 1995; Darlington et al., 1993). This architectural pattern is inherent in computer clusters in which pieces of computation snippets of a large application are assigned to the nodes through the middleware such as Message Passing Interface (MPI) or Parallel Virtual Machine (PVM).

The process tree of the Divide-and-Conquer pattern can be shown as in Figure 3.13, which represents a 3-level Binary Divide and Conquer pattern.

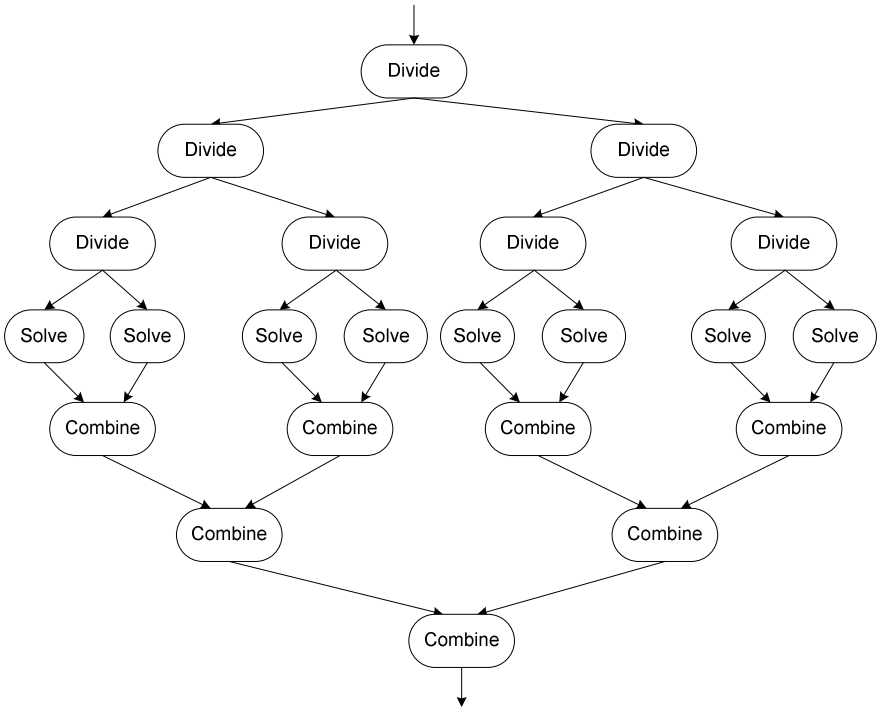


Figure 3.13 The process tree of Divide and Conquer pattern

A classical example of the Divide-and-Conquer pattern in parallel computing is the Merge-Sort algorithm that is an $O(n\log n)$ comparison-based sorting process

invented by John von Neumann -- the father of modern computer -- in 1945 (Cormen et al., 2001). At runtime, for a data list, the Merge-Sort algorithm employs Divide-and-Conquer approach and works as follows:

1. If the list is empty or has just one component, then no further sorting operation is needed for this list, otherwise:
2. Continue dividing this list into two sub-lists of half the original size.
3. Implement sorting function on each sub-list recursively by re-invoking the Merge-Sort core.
4. Merge the two sub-lists back into one sorted list.

The Merge-Sort algorithm integrates two different functions to improve the computational efficiency, which reflects the intrinsic principle of the Divide-and-Conquer pattern to decrease computational complexity:

1. A smaller list can take fewer steps for sorting comparing to a larger one.
2. If using two sorted lists, then fewer steps are taken to form a sorted list than using two unsorted lists, because each sorted list needs to be traversed just once if they are already sorted.

The following classic example from the renowned “The Art of Computer Programming” written by Donald (Donald, 1998) explains the principle of Divide-and-Conquer-based Merge-Sort algorithm in detail. Suppose there is an array $a[1\ldots n]$, the Merge-Sort algorithm splits the array into two sub-arrays, and then recursively implementing the sorting function on each sub-array. It then merges the two sorted sub-arrays to generate the result as shown in the following pseudo-codes which includes two programs *merge-sort*() and *merge*() employing various input variables:

Program merge-sort ($u[1\ldots s]$)
Input: $u[1\ldots s]$
Output: $u'[1\ldots s]$ that is the sorted $u[1\ldots s]$
if $s > 1$:

```
        return merge(merge-sort (u[1..⌊s/2⌋]), merge-sort (u[⌊s/2⌋+1...s]))
else:
        return u

function merge(p[1 … r], q[1 … t])
if r= 0: return q[1 … t]
if t = 0: return p[1 … r]
if p[1]≤ q[1]:
        return p[1] O merge(p[2 … r], q[1 … t]      // O denotes concatenation
else:
        return q[1] O merge(p[1 … r], q[2 … t])
```

From the above pseudo-codes, it is clear that function *merge-sort* () issues the divide process and function *merge* () issues the solve and combine process (the top and bottom half of the Figure 3.13). Suppose array $u[1…s]$ now corresponds to real number [9, 2, 11, 5, 8, 4, 3, 13], the flowchart of the Merge-Sort algorithm sorting can be depicted as in Figure 3.14.

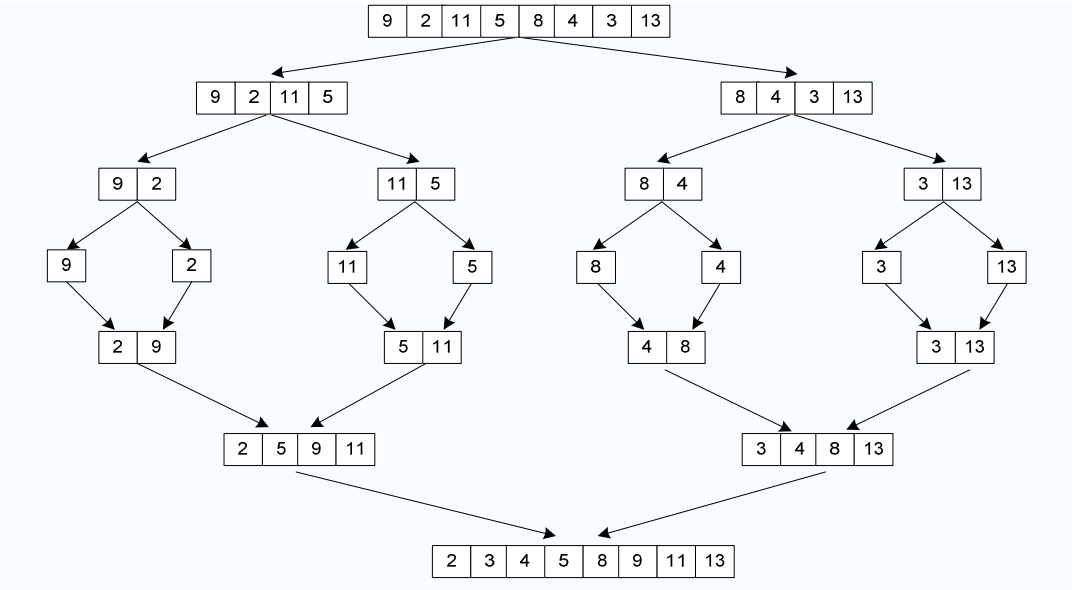


Figure 3.14 Demonstration of the Merge-Sort algorithm

3.7.2 Pipes-and-Filters

The Pipes-and-Filters pattern is a parallel processing structure in which a filter function is a process that pusher or pulls data stream to the adjacent processing unit through a data pipe (Darlington et al., 1993; Goodeve, 1994). Building families of related systems can be achieved by combining filters. By definition in this design, a filter is similar to a parallel processor that comprises of multiple processing steps. In contrast, the functions of a pipe are much simpler and mainly focusing on transferring data flow between filters. The tasks of a filter operating on the input data can include enriching, refining or transforming through adding information, collecting or distributing information, and transforming data by delivering it in certain specific representations.

The Pipes-and-Filters pattern originated from the applications in which a number of computational tasks are implemented orderly but independently, which is similar to a queue of time-step operations, on ordered data. In this case, the output stream of the first computational task becomes the input of the next task. The achievement of parallelism in this form can be obtained by overlapping operations on different pieces of data through time. A typical example of the Pipes-and-Filters in practice is to use it for managing the arithmetic units in a supercomputer (Meunier, 1995) where each arithmetic unit is equivalent to a filter.

To maintain consistent synchronization, the Pipes-and-Filters pattern sets the activity through triggering in between neighbouring filters with adaptable buffering mechanisms for storing intermediate data. A filter task can be activated by one of the following events as defined by Meunier (Meunier, 1995):

1. A request from the subsequent pipe indicating to pull data stream from the current filter as its output.
2. An instruction from the previous pipe requiring to push data stream to the current filter as its input.

3. Both the neighbouring two filters are activated where one filter sends out data stream to the pipeline, while the other receives data from the pipeline.

There are various communication styles between a pipe and a filter, corresponding to the three methods above, to coordinate the synchronization process. Figure 3.15 shows the case of the push method. In this case, filter A actively and continuously pushes data out to the adjacent pipe until an overflow indication from the pipe is received; once acknowledgements (ACKs) from the pipe is received, the pushing activity of filter A will be triggered again; the received data in pipe will also be sent to filter B where it just passively receives data stream and sends the notifications or the overflow signal to control the operation.

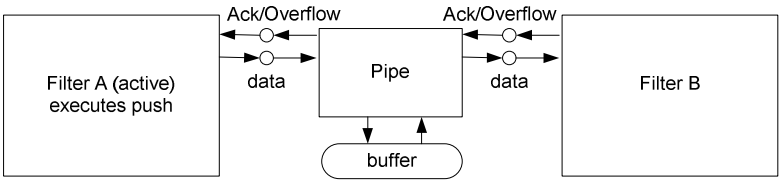


Figure 3.15 Coordination between Pipes-and-Filters in the push method (Courtesy to Meunier)

The pull method depicted in Figure 3.16 is analogous to the push method in Figure 3.15 except that notifications for data stream transferring are actively required by filter B, while in the case of push method, the same requests are originated from the pipe.

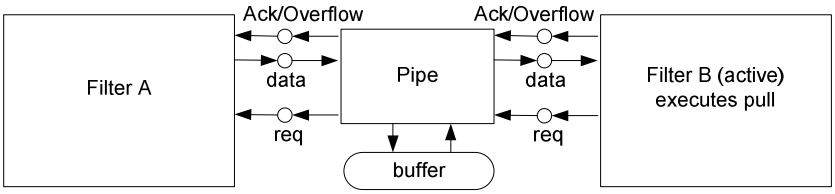


Figure 3.16 Coordination between Pipes-and-Filters in the pull method (Courtesy to Meunier)

Figure 3.17 shows the case in which both two neighbouring filters are active, which can be simply viewed as the synthesis of push and pull methods. This

hybrid structure provides maximum flexibility in ensuring the efficiency of the asynchronized communication.

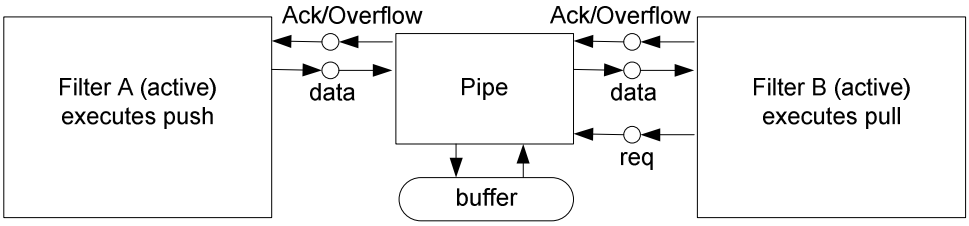


Figure 3.17 Coordination between Pipes-and-Filters where both two filers are active (Courtesy to Meunier)

3.7.3 Communicating Sequential Elements

As illustrated by Table 3.1, the Communicating Sequential Elements pattern belongs to the category of domain parallelism, therefore each processing element actually implements the same instructions on different pieces of data sets (Chandy and Taylor, 1992; Christopher et al., 1994). On the other hand, implementions in each processing element also need partial results from neighbouring elements. Commonly speaking, the communication or data exchange between the adjacent processing elements is based on internal buses or external networks, which is dependent on the hardware platform.

Communications between processing elements of this pattern utilize fixed and predictable paths. This feature can be illustrated more clearly by a dynamics problem typified by Christopher (Christopher et al., 1994): “the data represents a model of a real system, where any change or modification in one region influences areas above and below it, and perhaps to a different extent, those on either side. Over time, the effects propagate to other areas, extending in all directions; even the source area may experience reverberations or other changes from neighbouring regions. If this simulation was executed serially, it would

require that computations be performed across all the data to obtain some intermediate state, and then, a new iteration should begin".

Comparing to the features of domain parallelism, parallelism under communicating sequential elements pattern has introduced multiple participating concurrent elements with each of them capable of issuing a number of instructions to a data subset independently. An element can access the results processed by other elements, which is achieved by exchanging data through communication channels. An element can communicate in various formats, for instance, synchronised or asynchronised, single data set or multiple data objects in 1-to-1, 1-to-many, many-to-1 or many-to-many modes.

As shown in Figure 3.18 (Arjona, 2006), in communicating sequential elements pattern, the functions of each sequential element implement a set of instructions on its private data subset through sending or receiving messages across the unified interface, while communication channel stands for a medium between concurrent sequential elements for synchronization.

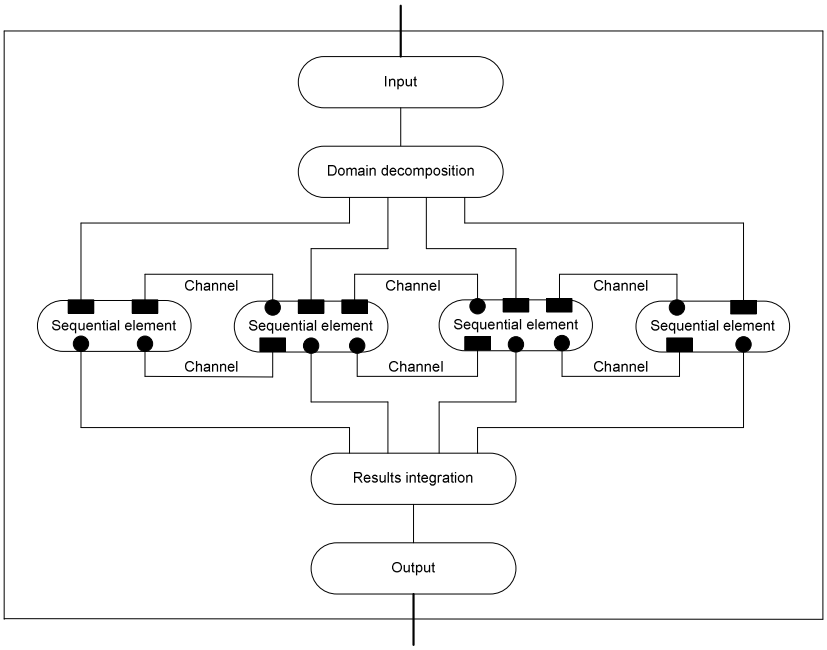


Figure 3.18 Communicating sequential elements pattern (Courtesy to Arjona)

3.7.4 Processor Farms

The Processor Farms pattern, sometimes also referred as Manager-Workers pattern or Master-Slave pattern (Buschmann et al., 1996), represents a simple strategy to parallelize problems consisting of one computation to be executed on a collection of initial data (tasks). In this pattern, a collection of processors that work together to process several specific pieces of data. Tasks are distributed, or "farmed out", by one "farmer" processor to several "worker" processors which then execute those tasks independently, and information and results are then sent from these "worker" processors back to the "farmer" processor (Shaw, 1995), as depicted in Figure 3.19.

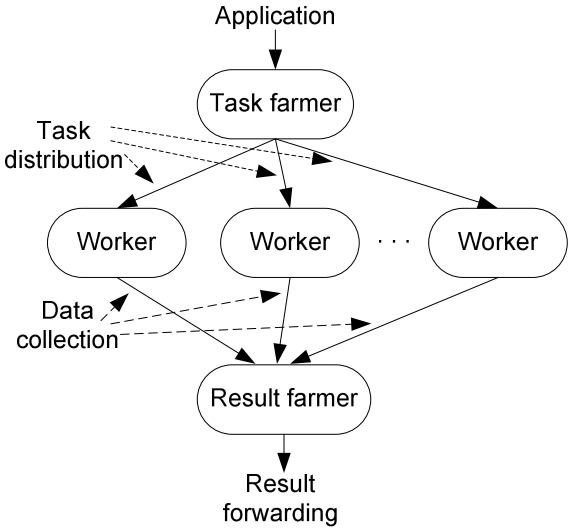


Figure 3.19 The Processor Farms pattern

The Processor Farms pattern is suitable for applications which can be partitioned into many separate and independent tasks. The parallelism is then activated by processing several tasks concurrently. In this case, each “worker” repeatedly seeks a task to perform till the program is finished. Each “worker” executes its own task independently. If tasks are distributed at run time, a crucial problem of the structure is to achieve load-balance (Goodeve, 1994). Another problem need attention is the communication costs between “farmers” and “workers”.

One of the first consumer-level “parallel” CPUs that compiles the Processor Farms pattern is the so-called Cell CPU released by the alliance of Sony Computer Entertainment, Toshiba Corporation, and IBM (Gschwind, 2007). The Cell CPU aims to bridge the gap in between the conventional CPUs and the more specialized high-performance processors such as GPUs. It is designed as a stream processor that consists of a controlling processor -- Power Processing Element (PPE), and multiple SIMD coprocessors -- Synergistic Processing Elements (SPEs) with independent program counters and instruction memory to form an innovative structure for applying multiple instructions on multiple data sets. In this architectural design of Cell CPU, the PPE, that acts the role of “farmer”, has control over the SPEs and can trigger, end, break off, and schedule subtasks implemented by the SPEs. The PPE can also access the main memory and the private memory of all SPEs through the standard load/store instructions. Each SPE is a RISC processor that is equipped with a 256 Mb embedded SRAM for instruction and data, called "Local Storage" which can be accessed directly by PPE. The PPE and SPEs are linked together by an internal high speed bus called "Element Interconnect Bus" (EIB) that is the internal communication system. Cell CPU can have a number of different configurations, the standard configuration is composed of 1 PPE and 8 SPEs (1 farmer and 8 workers) (Gschwind, 2007), which is shown in Figure 3.20.

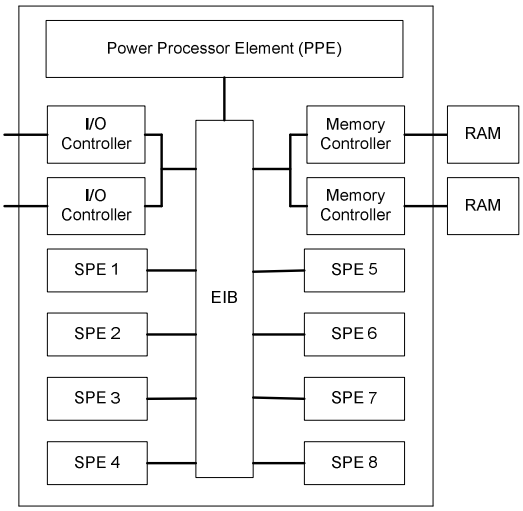


Figure 3.20 Cell CPU Architecture (Courtesy to Gschwind)

Referring back to the Figure 3.19 that shows the processor farms pattern in the form of a single layer, the practical applications are often come in the form of multiple layers which results in the fact that some components in the intermediate layers will act in the hybrid role of “farmer” and “worker” at the same time (Wagner et al., 1997). In addition, after task and data are distributed among all “workers”, these components can execute in the pattern of communicating sequential elements to provide domain parallelism. Based on these behavioural patterns, processor farms can be seen as a container for the pattern of communicating sequential elements.

3.8 Summary

This chapter has provided a broad review on GPU programming methods, techniques, and GPGPU concepts and practices. Since the early appearance of GPU was intrinsically a graphics device, the programming methods of GPGPU were purely based on the conceptions of graphics pipeline such as vertex shader, rasterizer, and fragment shader. This has largely determined that the core of GPGPU on the legacy GPUs was on how to map the implementation of a computational task for data parallelism to the operations and resources in the graphics pipelines. Stream and kernel are the basic elements to describe the concepts of data sets and operations on the data collection. Since the stream is the input and output of a GPU, for this reason, a GPU is also referred as a stream processor. Kernels are consisted of user-defined programs written in assembly or high level shading languages to enable the implementation of parallel operations on the data stream.

The activation of the implemented kernels on a GPU is normally accomplished by the instructions of graphics APIs such as OpenGL and Direct3DX. These instructions can trigger operations such as texture fetch, computational range setting and domain specification. The data storage and access are facilitated by

the introduction of the Pixel buffer and Frame Buffer Object that correspond to off-line rendering capacity in a graphics device.

In addition to the review on basic conceptions and methods in GPGPU, two key differences between GPU programming and CPU programming, flow control based on branching operations and data structures based on physical memory, have been discussed in detail. In a GPGPU practice, the ordinary branching instructions readily available in CPU programming, i.e., the “if-then” instruction, has to be implemented in great care due to the extra computational cost. Various techniques that include making use of GPU’s hardware feature have been developed with variant degree of success in solving the flow control problem. In contrast to CPU programming where data set or vector are often stored in various dimensional arrays, GPU adopts its basic physics memory in the form of 2D textures. Communication between these two distinctive data structures requires memory address translations between array and texture. A series of advanced GPGPU techniques, based on the example of optimization for GPGPU-based linear algebraic operations, have been analysed to reveal their merits and dilemma in facilitating GPU’s parallel processing effort.

The rapid evolution of GPU hardware and GPGPU development tools have exposed the need to better understand the parallel patterns on a PC-grade parallel processing system. Following the review of the GPGPU theories and practices, Chapter 3 also discussed the generalized architectural patterns of parallel programming, i.e., divide and conquer, Pipes-and-Filters, communicating sequential elements, and processor farms. Through careful analysing these four patterns, it becomes clear that the development trend for consumer-level parallel systems would ideally engaging efforts in devising and improving both the hardware designs and the supporting software models and programming tools. The roles of CPU and GPU in an integrative parallel architecture will be discussed in the next chapter with the aim to obtain general guidelines for GPGPU programming when facing different generation of GPUs and their development tools.

Chapter 4 General Programming Framework of GPGPU Applications

The evolution of architectural pattern of parallel processing, that is, from the early Divide-and-Conquer pattern to nowadays Processor Farms pattern, reflects the renovation trend of hardware design in parallel computing. This trend has also resulted in the dramatic change of software tools for parallel computing. As a kind of consumer-level parallel processor, the hardware structure of GPUs generally experienced the period from the traditional graphics accelerator which is still based on the conception of graphics adaptor to the latest multi-core processor which commonly employs the unified shader.

The rapid pace of development on GPU hardware and the corresponding programming languages within the last decade have also brought in confusions to researchers and application developers devoted in spreading GPGPU powers to wider areas when they are facing to different generations of GPUs in the real world. Therefore, it is essential to establish a general GPGPU programming framework which is capable of encompassing the variety of GPU's hard features and program instructions that can be readily utilised for individual GPGPU program design. Following the contents in previous chapter, this chapter analyses the general GPGPU programming framework and its detailed functions.

4.1 GPGPU's Parallel Architectural Pattern

Comparing with the basic concepts and methods in GPGPU programming, as introduced in Chapter 3, the procedures for implementing a conventional parallel programming system are more complex, which often involves constructing and encapsulating the integration of parallelism, communication, synchronisation and

embedment (Berrington et al., 1993). It is these procedures that characterized the various parallel architectural patterns.

Although the practical GPGPU applications are fairly difficult to develop, the flexibility of development can be achieved through establishing its general programming framework to guide the detailed programming model design. This GPGPU programming framework must rely on the GPGPU's parallel architectural pattern because the functions of its components are ultimately determined by the style of the employed architectural pattern. Based on a specific general programming framework, various GPGPU programming models which actually apply individual algorithms in applications can then be designed effectively. Broadly speaking, the relationship of the parallel architectural pattern, the GPGPU's programming framework, and the programming models can be depicted by Figure 4.1, which also represents a focalized approach of the researches carried out in this thesis.

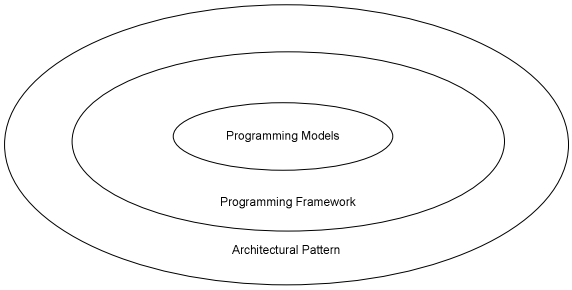


Figure 4.1 The relationships of GPGPU's parallel architectural pattern, programming framework and models.

From the introduction in Section 3.7.3, it can be concluded that, for GPU itself, its internal processing pattern follows the so-called Communication Sequential Elements. For the early generation of GPUs that uses vertex and pixel shader functions as the graphics accelerator, the multiple rendering pipelines existed in the vertex and pixel streams act as the sequential elements, and the registers act as the communication channel (as depicted in Figure 3.18) between these rendering pipelines. For the pipelines in a vertex shader, the inputs are vertices and the outputs are projected polygon assembles, while for the pipelines in a

pixel shader, the inputs are fragments produced by the rasterizer through fragmentation and the outputs are pixels. For the latest GPU that employs unified pipelines and function as real-meaning parallel processor, the unified shader, now also called “core”, is of the sequential element, while the shared memory acts as communication channels.

However, a complete GPGPU application includes CPU routines, API instructions, and kernels that actually run on the GPU. Although a GPU has substantial parallel processing ability, it is after all just a co-processor. As stated above, there is no doubt that GPU works in the pattern of communicating sequential elements. But considering the role of CPU and GPU in GPGPU applications, it is clear that a complete GPGPU application actually follows the Processors Farms pattern in which CPU schedules tasks and GPU acts as worker who has parallel processing capability. It means the GPGPU programming framework must follow the function specifications of “farmer” and “workers” pattern that have been introduced in Section 3.7.4. For the detailed analysis, any practical implementations should be referred to the actual parallel processing pattern and be standardized to deduce the confusion when deploying the GPGPU’s parallel programming framework. Four implementation stages - tasks and data streams, partitioning, communication, agglomeration and mapping - are to be analysed in the following section.

4.2 Implementations in Programming Framework for Parallel Systems

No matter which aforementioned parallel architecture patterns is to be adopted for a system, four detailed implementations -- partitioning, communication, agglomeration and mapping – need to be specified for the processing (Culler et al., 1997). Among them, the partitioning and communication are focused on scalability and concurrency characteristics, while the agglomeration and mapping aim to shift locality and other performance-related events.

In addition, these four implementations are overly determined by the specific hardware they are running on. Therefore, it is sensible to analyse the effects of these implementations in combination with the GPU's hardware structure.

1. Partitioning

Partitioning in all parallel programming assignments is consisted of two aspects: task specification and data partitioning. The responsibility of task specification is decomposing an application into a set of operations. These operations are hierarchically defined and related in accordance with their dependency. Whether an operation should be carried out on CPU or GPU is a typical responsibility of the task specification mechanism, which is often determined by the dependency and the complexity of the state in this operation. In simple terms, any tasks that can be described by a state machine through explaining the relationships among the states can be referred as an operation. Therefore, through some kind of computerized analysis on the states involved, a judgement can be made on whether an operation can be issued on GPU. For operations being issued on GPU, the communication cost in between CPU and GPU is another major factor to be assessed. The responsibility of the data partitioning process is to determine the size of streams based on the size of the original data set and the GPU's memory capacity. If the size of the data set is larger than the GPU's memory capacity, it will then be divided into ordered sections through partitioning.

2. Communication

The parallel architectural pattern adopted by most GPGPU applications follows the processor farms pattern with communications deployed to coordinate the operations between the farmer and worker. Messages are exchanged between the farmer and worker, which are often in the form of instructions from specific APIs (i.e., OpenGL and DirectX). The key of those messages are status parameters returned from each operation that indicates status of data consumption of the workers and the dynamic task allocation.

As the individual “worker” of GPGPU application, GPU works in the pattern of Communicating Sequential Elements, as explained in Section 4.1, the stream processor in GPU exchanges partial computational results with its neighbours through a set of registers or shared memories.

3. Agglomeration

Although there are task and data partitioning at the initial stage of most parallel systems, performance and implementation costs need to be carefully balanced throughout an application’s lifecycle, particularly when implementing agglomeration based on data partitioning. A good design will allow the size of data chunks to be changed when agglomeration takes place, which means data pieces can be integrated or partitioned into larger or smaller ones to promote computational efficiency or to reduce the overhead from the communication. In the Processor Farmers pattern, the granularity is adjusted with the aim of allocating the data set among the workers evenly to avoid the phenomenon that some workers are idle since small amount of data are received while the others remain busy trying to serve the farmer’s requests. Many parallel programming languages have the function of agglomeration, but often the programmer can also implement the agglomeration manually to achieve more optimal performance. For example, there are memory hierarchy that involves global memory, constant memory, texture memory, and shared memory in the now unified-pipeline of a GPU, the shared memory allows faster access rate than the rest. By transferring data from global memory to shared memory through optimization, higher GFLOPs can be obtained when executing certain algorithms than those implemented by the CUBLAS tools (a library released by CUDA for basic linear algebra routines) (Baskaran et al., 2008).

4. Mapping

For a GPGPU application in the past, the responsibility of task mapping resides with the application developer to segregate and map each part of an application to the GPU’s hardware structure such as the transformation and lightening (T&L) pipeline. For the job of data mapping, the key task is to transform data into sizes

that are suitable to the particular GPU's memory. This style has changed significantly since the arrival of CUDA and the unified-pipeline-equipped GPUs that will be the focus of next section and Chapter 7.

In fact, except GPGPU applications, as indicated in Section 3.7, the recent evolutional trend has seen many parallel systems moving toward the processor farms pattern due to its hybrid functional and domain decomposition features. In addition, the demand for fine-granularity parallel processing has stirred up the research into parallel hierarchies, i.e., nested parallelism, which is further evidenced by the appearance of the unified-pipeline-structure GPU in latest and the release of CUDA (Nvidia Corporation, 2009). Furthermore, the aforementioned Cell CPU from IBM aims to make the CPU functioned like a stream processor with software support. The concept and practices of the so-called field-programmable gate array (FPGA) further assists the programmable reconfiguration ability for electronic circuit design that will enable future video game consoles to maintain real-time and interactive rate (Kolks et al., 2009).

Although individual parallel processor might has different structure and supporting software, a framework of virtualized parallel system based on processor farms pattern is presented in this section, as depicted by Figure 4.2.

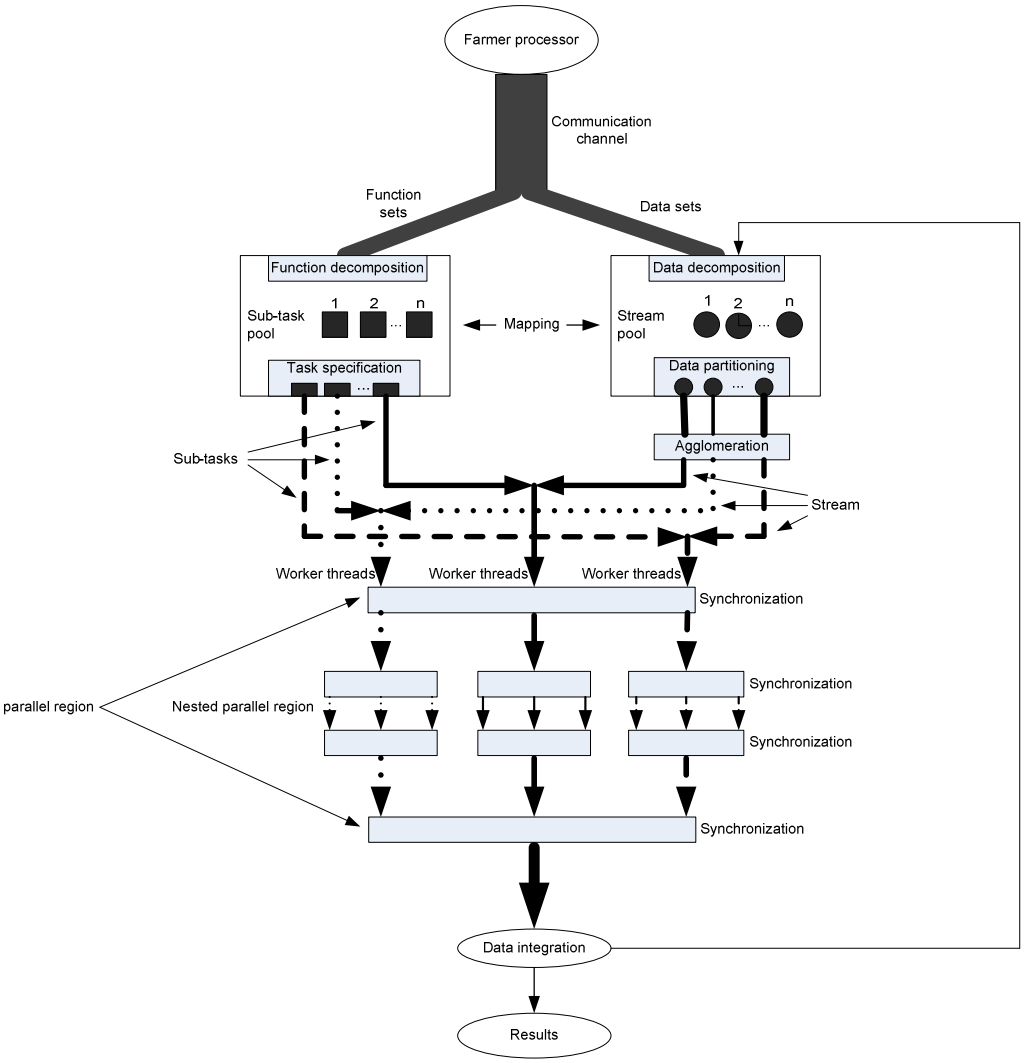


Figure 4.2 The framework of virtualized parallel systems

In Figure 4.2, a practical application is represented as the combination of functions and data sets. During functional decomposition, some independent operations will be assigned to the sub-task pools for parallel processing, while the other functions will have to be issued by the farmer processor in a serial mode due to their inherent correlation. For a SIMD processor such as the GPU, the tasks in the subtask memory will be operated upon by a single program. Similarly, any independent data will be sent to the stream pools that correspond to the memory of the co-processors such as the texture memory in GPU. The process of dynamically assigning data to the stream pools is sometimes referred

as data mapping that determines the size and format of data stored in the co-processor's memory. For example, in OpenGL, transferring data into a stream pool is issued by the instruction *glTexImage2D*( ), while in CUDA it is realized by the instruction *cudaMemcpy*() that copies an array stored in the host memory to a device memory. The function of synchronization in a parallel system is to control the pace of the execution of multiple threads to coordinate the memory access activities. Synchronization in real system implementation is commonly achieved by setting up a barrier at which all threads in a coprocessor must wait before any are allowed to proceed, as depicted in Figure 4.2. The output of the parallel region in Figure 4.2 is stored in a data repository through the process of integration that might either be an intermediate processing result that will be used for the next step of processing, or the final result that will be collected by the result farmer.

This virtualized parallel system aims to locate partitioning, communication, agglomeration, and mapping on different parts of the system through shielding the hardware distinctions of various parallel processors. If using Figure 4.2 as the blueprint and to integrate a specific GPU's hardware into its corresponding parts, then the general GPGPU programming framework that is built on these four implementations can be deduced.

4.3 GPGPU's Programming Framework

As explained in Section 4.1, a complete GPGPU programme is a hybrid application of instructions run on both the CPU and the GPU. From the stand of architecture pattern of parallel programming, the relationship between a CPU and a GPU is a farmer-and-worker pair. The CPU schedules the task and the GPU acts as coprocessor to operate in the SIMD mode. If only considering the operations implemented on the GPU, it works in the pattern of communicating sequential elements where sequential elements are pipelines in the vertex or pixel shader formats. Although the terminologies and programming methods for

GPGPU have been examined in detail from Section 3.3 to 3.5, it is yet to see a general guideline for the GPGPU's programming framework design based on the Processor Farm pattern explained in this chapter. As stated in the foreword of this chapter, a framework will be used to guide the design of corresponding GPGPU's programming model when various generations of GPU have to be involved in practical applications. For this sake, based on the blueprint depicted by Figure 4.2 and the feature of GPU's hardware architecture as introduced in Chapter 2, two conceptualized GPGPU's programming frameworks are devised at here, which address the application procedures by using the traditional GPUs equipped with distinctive vertex and pixel shaders, and the new generation GPU with unified-pipeline and shaders.

4.3.1 Programming Framework for Conventional Graphics Pipeline

The GPGPU's programming framework based on conventional GPU, that is, the GPU employing traditional vertex and pixel shader, is shown in Figure 4.3. Referring to the virtualized parallel system depicted in Figure 4.2, it is certain that the CPU acts as farmer processor and the GPU is the worker processor in this case. The worker's role is played by the vertex shader and/or pixel shader, while the GPU's memory such as the framebuffer or textures are the stream pools to the store data sets. Vertex or pixel shader, which is the kernel executor, is the sub-task pool to contain the sub-tasks that can be implemented in parallel. The vertex buffer or framebuffer, which corresponds to data integration, is the data repository to store the intermediate or ultimate results.

Except agglomeration that is controlled by GPU's own hardware mechanism, the partitioning, mapping, and communication operations are all issued by the corresponding graphics API instructions. which explains the the fact that for a single application there could be different GPGPU solutions in which different strategies might be employed on task allocation and data mapping.

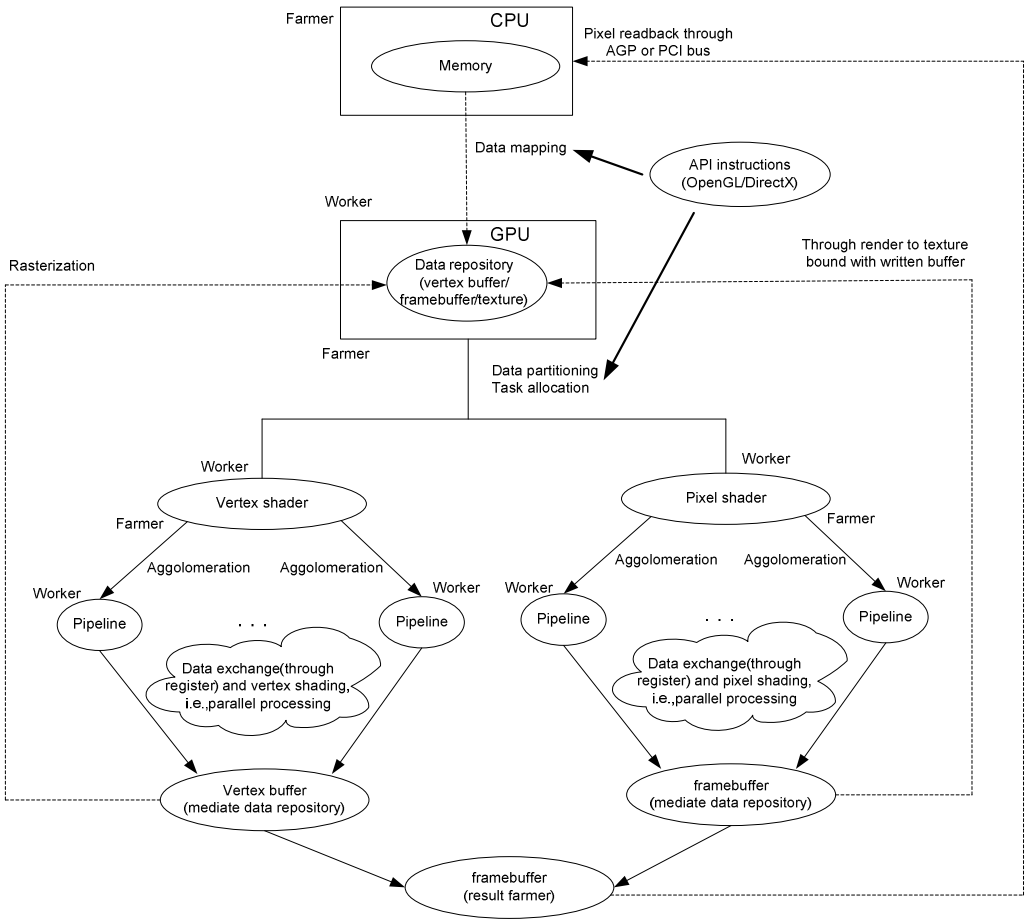


Figure 4.3 The conventional GPGPU architectural pattern

As the workers in GPU, both the vertex shader and the pixel shader have the ability of processing data in parallel. The challenge is that GPU can only allocate tasks to them in order with vertex shader at front. In addition, if only vertex shaders are utilised, the results will still need to be transmitted through the stage of pixel shader to be acquired since the vertex shader is only treated as a geometry transformer, while the pixel shader is designed for more intensive algebraic computation with a lot more pipelines. This legacy structure will result in the problem of workload balance as highlighted in Section 2.1.2. This is also the main motive of the major GPU vendors to propose and release the unified pipeline structure, which is regarded by the consumer-grade parallel processor researchers as a significant breakthrough to the traditional conception of PC computers. The distinctive advantage of unified pipeline over the old design is

that task can be distributed across all pipelines evenly, therefore, erasing the "unfairness" existing of free workers vs. heavy-burden workers.

4.3.2 Programming Framework for Unified Pipeline

The GPGPU's programming framework based on latest GPUs that employed the unified pipeline structure, is shown in Figure 4.4. GPU memories such as the global memory or textures are the stream pools to store data sets in this case. Comparing with the legacy GPUs using vertex and pixel shaders, the new stream pools can directly store arrays, which is more convenient for data mapping. The unified shader units, or "cores" in GPU, are combined together to be used as the sub-task pool in which a task is further decomposed into multiple thread blocks. The implemented result of each thread is integrated to the global memory that is equivalent to the data repository in Figure 4.2.

Similarly, task or data partitioning, mapping, and communication between CPU and GPU are controlled by developers through API instructions such as CUDA functions. The data agglomeration can also be controlled by developers through specifying the location of synchronization in programs.

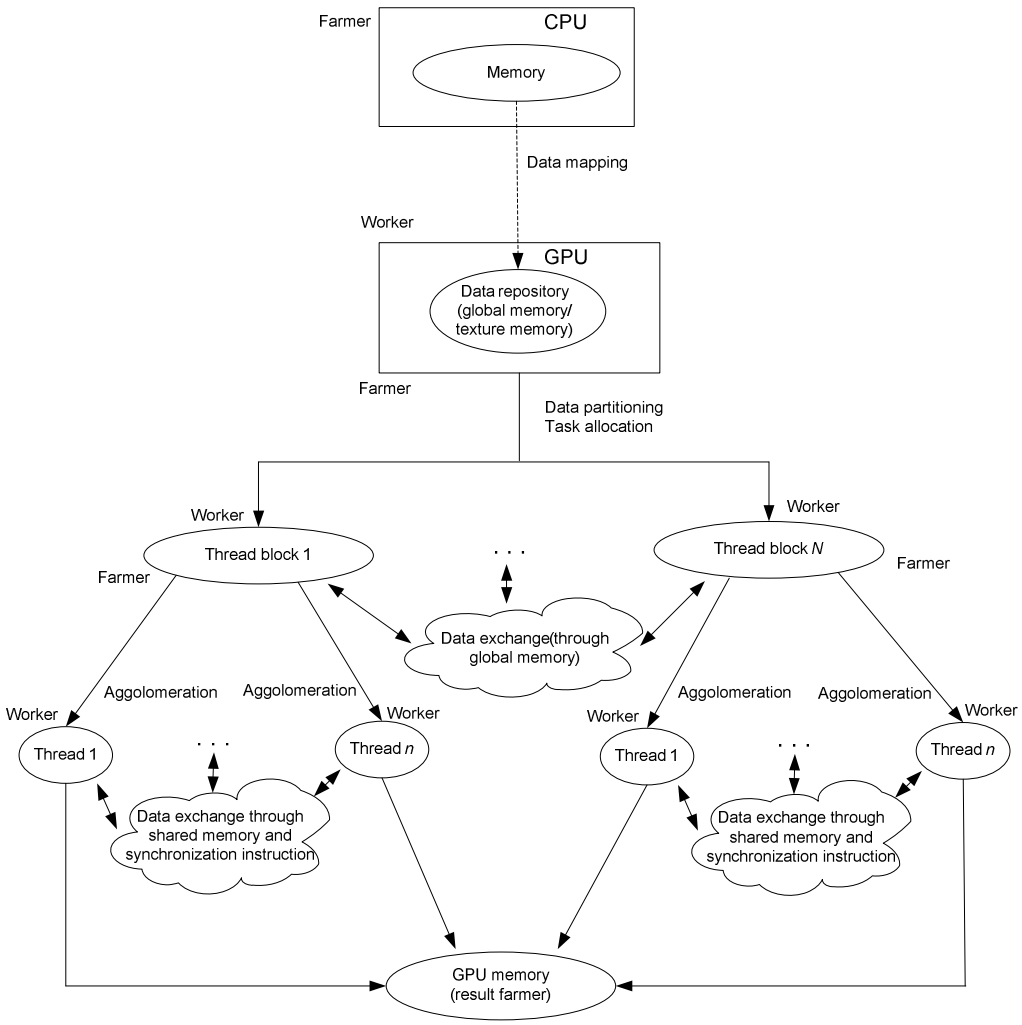


Figure 4.4 The new GPGPU architectural pattern with embedded unified pipeline

Corresponding to the above programming framework, the Compute Unified Device Architecture (CUDA) and its corresponding programming APIs have been released in accompany of the invention of the unified pipeline architecture as its supporting software. In contrast to the legacy programming framework depicted in Figure 4.3, the key difference of the new one is the blur of the vertex and pixel shader border and the availability of the commonly accessible local memory by all “thread blocks”.

4.3.3 Programming Model Design

In this thesis, the concept of architectural patterns is used for categorizing the parallel processing, while the GPGPU programming framework which is based on a specific architectural pattern - Processor Farms - is used as a general guideline to develop GPGPU applications where GPGPU is viewed as a branch of parallel computing. For a single GPGPU application, there should be a programming model to devise its solution. Basically, the programming model is originated from the programming framework, which has been depicted by Figure 4.1 that also represents a gradually more focusing research pipeline involved in this thesis. After the identifying the architectural patterns and the programming frameworks, the GPGPU programming model design is becoming the next focus of this study.

Referring to Figure 4.2, it is clear that any application can ultimately be divided into the combination of tasks and streams which are contained in its own container, task pool and stream pool, respectively. The four aforementioned implementations, partitioning, communication, agglomeration, and mapping, are then used for each components in the task and stream pools. Through the analysis on GPGPU's programming framework depicted by Figure 4.3 and 4.4, it is concluded that communication, agglomeration are generally fixed, that is, they have common instructions and constant implementation orders for most GPGPU applications. For example, data agglomeration is normally achieved by GPU's hardware, it is not transparent and programmable to developers, i.e., it can't be directly controlled by developers. Although issued by API instructions, the communications between CPU and GPU are also relatively straightforward as depicted in Figure 4.2 to 4.4.

The objects involved in partitioning and mapping include task and data, while data partitioning and mapping are ultimately determined by the task specification. In another term, the complexity of a parallel processing system increases with the amount of parallel tasks employed and the agreements on "handshaking" protocols to maintain the synchronisation (or asynchronisation). Since GPU is a SIMD parallel processor, the maintenance of synchronisation or asynchronisation

is not as sophisticated as that one in MIMD systems. Therefore, for a specific GPGPU application, its programming model design will have to subject to strict task “re-specification” based on the GPGPU framework and the software-to-hardware mapping to transform the serially implemented functions into the kernels that can be operated in parallel, which will be tested in the case studies of the models in the following chapters.

The task re-specification and mapping are based on the analysis of the principles, rules and algorithms involved in the application. Therefore, it is essential to carry out pre-analysis on an application before trying to establish the programming model for compiling programs. The project in this thesis is to be applied to surface metrology operations, so the methods or algorithms in surface metrological data processing will be analysed in the following chapters.

Another essential task in the project is to validate the effectiveness of the designed model. The evaluation strategy has focused the computational efficiency and practicability of the devised program models. For the validation of computational efficiency, the speed up factor of GPGPU programs is treated as the key to evaluate the GPU’s acceleration performance, which is tested through comparing the run-time of GPGPU programs and its CPU-based solutions. For the validation of the practicability, the data accuracy of GPGPU programs has been analysed through comparing the deficiencies of the results of the GPGPU programs and the CPU-based solutions. Both the absolute discrepancies and the relative errors have been evaluated at the numeric level.

4.4 Summary

Following the review of the GPGPU theories, practices, and architectural patterns of parallel processing in Chapter 3, this chapter has analysed the architectural pattern which GPGPU relies on. By integrating the discussed GPU’s hardware and software factors, a general programming framework for GPGPU applications is provided with the aim to obtain a guideline for the detailed GPGPU

programming model design. The framework covers the legacy GPU with traditional graphics pipeline and the latest products with unified pipeline. The work has also focused on the common operations that need to be considered as essential elements in the design of a parallel processing system and/or application such as task and data partitioning, communications in between CPU and GPU, task and data mapping, and data agglomeration.

Based on the proposed programming framework, the development of a GPGPU programming model for a specific application is discussed. Generally speaking, for a detailed GPGPU application, the development will experience through phases such as principles/algorithms analysis, programming model design, solution implementation, and result evaluation. This development route will be examined in Chapter 5, 6, and 7 through 3 real application case studies encompassing from surface metrology to image processing.

Chapter 5 Accelerated Filtering Algorithms for Surface Profiling

Based on the devised GPGPU frameworks discussed in Chapter 4, this chapter focuses on a practical problem-solving case study in accelerating the processes involved in surface characterization. The proposed solution employs legacy GPUs with vertex and pixel shaders. The main algorithm acceleration was accomplished by developing and adjusting fragment programs. It is well known that before the era of unified-pipeline-based GPUs, direct data "scatter" operations were not supported for GPU's memory, which had to be issued through the mechanism of rendering-to-texture. This investigated case illustrates how to improve the usage of textures by binding them with the Framebuffer Object (FBO) that is an off-screen rendering technique introduced by OpenGL. Past GPGPU attempts suffered from massive data transportation from GPU to CPU, which is a bottleneck that seriously undermined the acceleration performance of GPGPU applications due to the limited bandwidth of earlier AGP and PCI buses. Data splitting is realized in this case study by following the designed framework to efficiently overcome the shortcoming. The performance of the proposed GPGPU programming model is validated through the implementation of a classical 2D Gaussian filter that are extensively used in surface metrology and then comparing it with the performance from a CPU-only MATLAB program for the same function.

5.1 Filtering Algorithms for Stylus-based Surface Metrology

According to which kind of measurement instrument is employed, surface metrology can be classified into two categories as stylus-based measurement and non-touchable systems (Blunt and Jiang, 2003). For stylus-based

measurement system, form, waviness and roughness are three main factors. The task of stylus-based surface metrology is to extract these factors to characterize a surface. It is noted these three factors correspond to different frequency segment if they are transformed into the frequency domain for analysis. For example, the roughness corresponds to high frequency components, the waviness corresponds to medium frequencies, and the form corresponds to low frequencies (Raja et al., 2002). Therefore, filtering technologies are extensively used in stylus-based measurement system.

The earliest filter used for surface characterization is a 2RC network which is a series of two RC filters that are built from a resistor and a capacitor (RC). The 2RC analogue filter was formally recommended a “standard wave filter” in ISO 3274-1996 (ISO 3274, 1996), which has the following impulse response:

$$h(t) = \frac{1}{RC}(2 - \frac{t}{RC})\exp(-\frac{t}{RC}) \qquad (5.1)$$

where t is a time axis.

The main disadvantage of 2RC filter is its nonlinear phase which causes some waveform distortion due to phase shift. To overcome the problem of phase distortion, 2RC phase-corrected digital filters were developed by adding weighting factors in Equation (5.1) to correct the phase offset.

In 1994, the Gaussian filter was made into the standard filter used for stylus-based surface measurement, which is described in ASME B46.1(ASME B46.1, 1995) and ISO 11562 (ISO 11562, 1996) respectively. The recommended Gaussian filter includes two types, 1D and 2D filters. The 1D Gaussian filter is used for establishing a mean line while 2D Gaussian filter is used for establishing a mean surface. ISO11562 defines the impulse response of 1D Gaussian filter in the time domain, and the corresponding amplitude-frequency response function in frequency domain as follow:

$$h(t) = \frac{1}{\alpha\lambda_c}\exp[-\pi(\frac{t}{\alpha\lambda_c})^2] \qquad (5.2)$$

$$G(\lambda) = \exp[-\pi(\frac{\alpha\lambda_c}{\lambda})^2] \tag{5.3}$$

where λ_c is the cut-off wavelength, ISO11562 regulates that when $\lambda = \lambda_c$, the value of $G(\lambda)$ should be 0.5 so that $\alpha = \sqrt{\ln 2/\pi} = 0.4697$.

A 2D Gaussian filter is the integration of two 1D Gaussian filters which are implemented in the x and y directions respectively. Its impulse response in spatial domain is defined as

$$h(x, y) = \frac{1}{\beta\lambda_{xc}\lambda_{yc}} \exp\{-\frac{\pi}{\beta}[(\frac{x}{\lambda_{xc}})^2 + (\frac{y}{\lambda_{yc}})^2]\} \tag{5.4}$$

Its corresponding amplitude-frequency response function is

$$G(\lambda_x, \lambda_y) = \exp\{-\pi\beta[(\frac{\lambda_{xc}}{\lambda_x})^2 + (\frac{\lambda_{yc}}{\lambda_y})^2] \tag{5.5}$$

where $\lambda_{xc}, \lambda_{yc}$ are the cut-off wavelengths along x and y directions respectively, and $\beta = \alpha^2 = \ln 2/\pi$, which guarantees the filter has an attenuation ratio of 50% at $\lambda_x = \lambda_{xc}$ with $\lambda_y = \infty$ or $\lambda_y = \lambda_{yc}$ with $\lambda_x = \infty$.

The distinctive characteristic of Gaussian filter is its feature on linear phase which is a great promotion for surface metrological data processing in contrast to 2RC filter. The Gaussian filter is therefore extensively used for establishing the mean surface in stylus-based surface metrology. A problem of Gaussian filter is the boundary distortion, which also exists in 2RC filter. The boundary distortion results in the consequence that the mean line or surface at the boundary region can not be correctly evaluated. Therefore, the Gaussian regression filter (GRF) (Brinkmann et al., 2000) was developed as an enhanced version of the Gaussian filter for more precise evaluation on a whole surface profile. For example, the 1D GRF is defined as a series of recursive steps, which can be written in the following discrete form

$$\sum_{l=1}^{n}(z_l - w_k)^2 s_{kl}\Delta x \to \underset{w_k}{Min} \tag{5.6}$$

where z is the profile ordinate; w the mean line ordinate; n the number of sample points; k the index for the location of the impulse function; l the index for profile points and s is the impulse function as given by

$$S_{kl} = \frac{1}{\lambda_c\sqrt{\ln 2}}\exp\{-\frac{\pi^2}{\ln 2}[\frac{(k-l)\Delta x}{\lambda_c}]^2\} \tag{5.7}$$

In the regression phase, the impulse function is calculated for every sampled point on the surface and a minimal objective function is employed to locate the ordinate on the mean line w. Generally, GRF can be further classified as zero-order GRF, second-order GRF, and robust GRF, which can be referred in Raja's publications (Raja et al., 2002).

As well as the 2RC and Gaussian filters, the R_k filter is another commonly used filter for surface evaluation, which was recommended by DIN standards and has the following amplitude-frequency response (DIN 4776, 1990).

$$G(\lambda) = 1 - (\frac{\lambda}{\pi\alpha\lambda_c})^2 \sin^2(\frac{\pi\alpha\lambda_c}{\lambda}) \tag{5.8}$$

where λ represents the wavelength and $\alpha = 0.44294647$.

It is noted that although various filtering algorithms have been designed for surface characterization and formed a series standard, the serious problem of efficiency of data processing has become more stringent, a feature due to the development of measurement instruments that can sample more point on an surface and the requirement of high accuracy (Yanagi and Hara, 2003). How to efficiently process the enormous measurement data, as a result, accelerating the filtering algorithms is becoming a problem for stylus-based measurement systems.

5.2 Filtering Algorithm Analysis

If an input signal $u(t)$ passes through a filter with impulse function $h(t)$, its output $O(t)$ can be computed either in the spatial domain or the frequency domain. Suppose the amplitude-frequency responses of $u(t)$, $h(t)$, and $O(t)$ are $W(\lambda)$, $H(\lambda)$, and $V(\lambda)$ which can be obtained through Fourier transform, then

$$V(\lambda)= W(\lambda)\bullet H(\lambda) \tag{5.9}$$

In spatial domain, $O(t)$ is equivalent to the convolution between $u(t)$ and $h(t)$, which is expressed as follow.

$$O(t) = \int_{-\infty}^{+\infty} u(t-\xi)h(\xi)d\xi \tag{5.10}$$

For signal processing system, convolution is the common operation used for computing the filtered signals. Equation (10) describes the continuous and infinite form of filtering. Its discrete and finite form is written as

$$O(j) = \sum_{k=-m}^{m} u_{i-k}h_k\Delta\xi \qquad j=-m,...,n+m \tag{5.11}$$

where n is the number of sample point on $u(t)$, $2m+1$ is the number of sample point on $h(t)$, and $\Delta\xi$ is the sample interval on $h(t)$.

If $u(t)$ and $h(t)$ are two-dimensional signal, then the discrete form of $O(t)$ is

$$O(x_i, y_s) = \sum_{k=-m}^{m}\sum_{j=-n}^{n} u(x_{i-k}, y_{s-j})\cdot h(x_k, y_j)\Delta\xi_x\Delta\xi_y \qquad i=-m,..., p+m \qquad s=-n,...q+n \tag{5.12}$$

Where p and q are the number of sample point on $u(x, y)$ along x and y direction, $2m+1$ and $2n+1$ are the number of sample point on $h(x,y)$ along x and y direction respectively. The discrete form of $u(x,y)$ and $h(x,y)$ thus can be represented by two matrices with the sizes of $p\times q$ and $(2m+1)\times(2n+1)$. Sometimes they are also called as *data window* and *filter window*. m and n are called as the filter radius of *filter window* in x and y directions. In a mathematical view, convolution is a scalar

product of two functions u and h, producing a third function O that is typically viewed as a modified version of one of the original functions. This is demonstrated below in Figure 5.1.

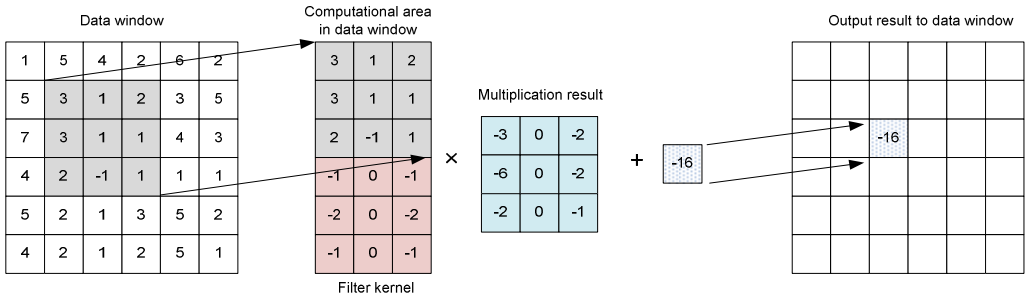


Figure 5.1 The convolution operation

In Figure 5.1, we can see that the sequential program that produces the output $O(x_i, y_s)$ from Equation (5.12) can be written in the following way, it is noted that the area of filter window is expressed in the form from (-m, -n) to (m, n) in Equation (5.12), so that the coordinates of the centre of the filter window is (0,0). But in actual computer systems, the area of window must be expressed in the form from (0, 0) to (2m, 2n). In this case the coordinates of the centre of the filter window becomes (m, n), so there should be –m and –n offset along x- and y-direction in the program.

```
for(i=0; i<p+2m+1; i++) {
    for(int s=0; s<q+2n +1; s++) {
        O(i,s)=0;
        for(x=0; x<2m+1; x++) {
            for(y=0; y<2n+1; y++) {
            O(i,s) += u(i+x-m,s+y-n)*h(x,y);
            }
        }
    }
}
```

the same operation on each element in O(t)

Figure 5.2 Sequential program for the convolution operation

Since all elements $O(x_i, y_s)$ in the output O(t) must be processed in the same way in the above program, we can see that the scalar product in the convolution operation is actually a SIMD computing operation, ideally suited to computation

on SIMD processors such as GPUs. In addition, although increasing the number of sample points increases the accuracy, it also produces much more data for analysis, and hence more time is required to implement the convolution operation. Improved accuracy therefore raises the problem of the computational efficiency of filtering algorithms, which was stated in previous section. For example, when a surface that was measured with 1024×1024 sampled data points is filtered by a 2D Gaussian filter window with a radius of 50×50, it takes the MATLAB-based multithreaded program about 5 seconds to complete the process on a 2.6GHz PC with 2GB memory. Based on the research findings detailed in Chapter 3 and Chapter 4, the following sections will implement the GPGPU framework targeting on accelerating the common filtering algorithms used in the stylus-based surface metrology and to evaluate the acceleration performance.

5.3 Hardware Acceleration for Filtering Algorithms

The hardware used for GPGPU programming is based on the conventional GPU's structure prior to 2007 (see Section 2.3.2) that are equipped with vertex and pixel shaders. Since the convolution operation is a form of numerical processing, the pixel shader is selected for the extra parallel processing ability over the vertex shader on the convolution computation. Generally, the whole application can be divided into 4 tasks:

1) Reading the original metrological data;
2) Constructing filtering window;
3) Issuing filtering algorithms on GPU;
4) Outputting the filtered data for visualization

Referring to the farmer-worker model for general GPU programming that is shown in Figure 4.3 in Chapter 4, it can be seen that the first step of the application is to allocate tasks to the "farmer" – CPU and the "worker" – GPU. Among the above four tasks, the first two, data reading and filtering window

establishing will be carried out on the CPU, the filtering operation is carried out on the GPU, and the data outputting will be carried out jointly by CPU and GPU.

5.3.1 The GPGPU Programming Model

According to the task allocation on CPU and GPU, the GPGPU programming model for filtering algorithms in surface metrology is described in Figure 5.3.

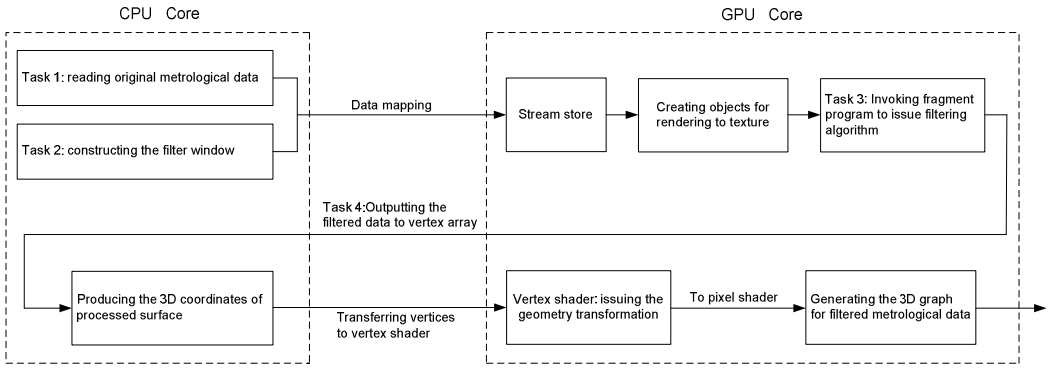


Figure 5.3 GPGPU programming model for filtering algorithms

There are two data sets involving in the computation on GPU, the original measurement data and filter window, which means at least two texture memory units are used to store these two data streams. In addition, GPUs based on traditional vertex and pixel shaders lack efficient “scatter” memory operations. Hence storing the results of convolution operation in a dynamic store unit and transferring them back to CPU are the problems for the presented framework. These details will be discussed in the following section.

5.3.2 Implementation Details

1. Data mapping

Data mapping is directed by the OpenGL instructions, the code in Figure 5.4 illustrate how to map the data in CPU’s memory to a texture.

```
glGenTextures( 1, &Data );
glBindTexture( GL_TEXTURE_RECTANGLE_NV, Data );
glTexImage2D( GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_R_NV, x_width,
              y_height, 0, GL_RED, GL_FLOAT, original_Data);
```

Figure 5.4 The codes for data mapping

The code in Figure 5.4 is used to create a texture named as *Data* on a GPU and transfer data from CPU memory, the data being stored in the array *original_Data* as a texture. A texture has four channels available, red, green, blue and alpha to store data. Here we just use the channel red to store the original data, which is specified by parameter *GL_RED* in the instruction of *glTexImage2D*().

2. Task allocation on GPU

For the vertex and fragment shaders, since just one can be chosen for running a parallel programme, it is necessary to specify which kind of shader will be used. There are API instructions to specify the worker on a GPU. For example, the parameter *profile* in OpenGL's instruction *cgCreateProgramFromFile*(*context*, *type*, *file*, *profile*, *entry*, *args*) indicates whether the programme will run on a vertex shader or a pixel shader.

3. Fragment program for convolution operation

Referring to the sequential program for the convolution operation shown in Figure 5.2, the fragment program written in *Cg*, which is a kind of high level shading language released by Nvidia corporation, is shown in Figure 5.5.

4. Data scatter

The last instruction, return *v* in Fig. 5.5, is a scatter operation that is similar to C-like code *x*[*i*]=*j*. Unfortunately, scatter is not as straightforward to implement in a GPU fragment program, since fragment processors are incapable of memory scatter (Owens et al., 2007). Various tricks are resorted to achieve the data scatter, the most common method in GPU programming is to bind a dynamic texture to a Pixel buffer (PBuffer) or Framebuffer Object (FBO), and then change

the value of pixel in Pbuffer or FBO through render-to-texture. In contrast to PBuffer, FBO is a more cost-effective solution. Each Pbuffer requires a unique GL context that includes both the device context of graphics device interface (GDI) and the rendering context (Oat, 2005). The problem of recording the states of all the GL contexts is a tedious work for programmers when facing a large-scale application. FBO requires no extra GL contexts and allows depth, stencil and color buffers to be shared among framebuffers which is impossible for a PBuffer-based approach. Based on these advantages, FBO was chosen as the solution to data scatter. Another reason for choosing FBO was that FBOs have a set of attachment points to which various textures can be attached. This is convenient for partitioning the result, which is vital for efficiently transferring massive data from GPU to CPU, a problem that will be discussed later. The attachment points are COLOR[n], DEPTH and STENCIL (Persson, 2007). To receive the results of a convolution operation, a dynamic texture is created and attached to an established FBO. The corresponding OpenGL instructions are shown in Figure 5.6.

```
float Filtering (uniform samplerRECT data :   TEXUNIT0,      // the metrology data
                 uniform samplerRECT filter:     TEXUNIT1,     // the filter window
                 uniform int window_width,              // filter window width in x- direction
                 uniform int window _height,          //  filter window height in y-direction
                  uniform float2 offset,                 //  offset in x- and y-direction
                  float2 pos : TEXCOORD0           //   texel position in TEXUNIT0
                  ) : COLOR
{
          float v = 0;
          for(int y=0; y< window _height; y++) {
                    for(int x=0; x< window _width; x++) {
                              float weight=texRECT(filter, float2(x, y)).r;
                              v +=  texRECT( data, pos + float2(x, y)  + offset).r * weight;
                    }
          }
           return v;
}
```

Figure 5.5 Fragment program to implement convolution operation

```
// Create a FBO
glGenFramebuffersEXT (1, &fbo);

// Create texture to store the results of convolution operation
glGenTextures( 1, &result );
glBindTexture( GL_TEXTURE_RECTANGLE_NV, result);
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_R_NV, Width/4,
                    Heigh/4, 0, GL_RGB, GL_FLOAT, NULL);

// Bind the FBO and attach result texture to it
glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbo);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                    GL_COLOR_ATTACHMENT0_EXT,
                                    GL_TEXTURE_RECTANGLE_ARB, result, 0 );
```

Figure 5.6 Data scatter through render-to-texture

5 Data splitting

For visualizing the filtered data, the computational results need to be transferred from the GPU's memory back to the CPU's memory, and stored in vertex array to form the 3D coordinate of each vertex, which was illustrated in Fig. 5.3. Although GPU has powerful parallel processing ability, this process possibly creates a bottleneck for applications and produces negative effects on GPU's acceleration performance (Geys and Gool, 2007). For example, it has been tested in our research that to transfer a 1124×1124 single precision floating-point data block from Nvidia's 7900 GPU back to CPU took nearly 5 seconds.

To partially resolve this problem, the framework shown in Fig. 5.3 splits that splits the result of filtering algorithm into several smaller blocks to speed up data transfer to CPU. The idea is mainly based on the principle of the Divide and Conquer pattern, described in Section 4.2.1. According to this principle, better performance will be produced when a large problem or data set is divided into several problems or several blocks of data set smaller in size. Thus data splitting works as follows: firstly, the primitive metrological data is split into n parts, where n is constrained to square of an integer to guarantee the normal texture lookup in

fragment program; Then n parts of data are convolved with the filter window respectively and the filtered data is stored in n dynamic textures, which are all bounded with a same Framebuffer object. Finally, the data in n dynamic textures will be read back to vertex array in CPU. The detailed process of splitting on the primitive data with the original size of *W*×*H* and storing mechanism for the filtered data in the form of n dynamic textures is shown in Fig.5.7.

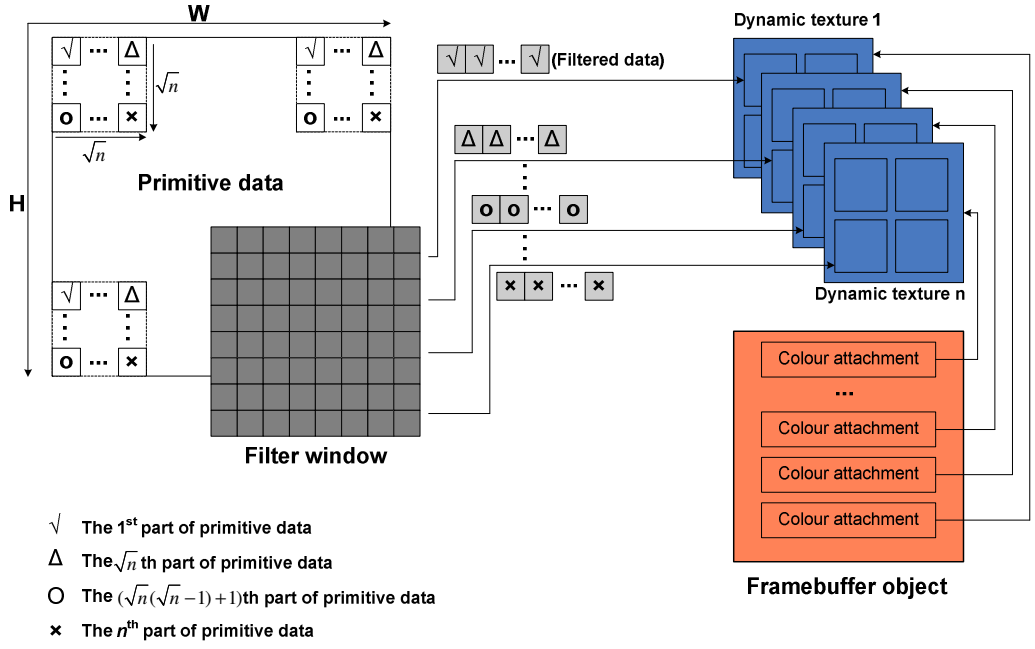


Figure 5.7 Data splitting and storage in Framebuffer object

6 Invoking GPU's task

How to carry out task computing on a GPU has been described in Section 3.2, also a process of data partitioning by specifying the vertex and texture coordinates when drawing a quad has been described. Considering the aforementioned data splitting illustrated in Fig.5.7. The computation on the first part of metrological data shown in Fig5.7 is implemented as shown in Fig. 5.8.

Processing the $\sqrt{n}$ th, …, $(\sqrt{n}(\sqrt{n}-1)+1)$ th, and n^{th} part of metrological data is similar with that on the first part of metrological data, only the texture coordinates of the four corner of the quad need to be adjusted. For example, the configuration

of the texture coordinate for computing $(\sqrt{n}(\sqrt{n}-1)+1)$ th part of metrological data is written as shown in Fig. 5.9.

```
glBegin(GL_QUADS);
        glTexCoord2f(0, 0);
        glVertex2f(0, 0);

        glTexCoord2f(RENDERBUFFER_WIDTH, 0);
        glVertex2f(RENDERBUFFER_WIDTH/ n, 0);

        glTexCoord2f(RENDERBUFFER_WIDTH, RENDERBUFFER_HEIGHT);
        glVertex2f(RENDERBUFFER_WIDTH/n, RENDERBUFFER_HEIGHT/n);

        glTexCoord2f(0, RENDERBUFFER_HEIGHT);
        glVertex2f(0, RENDERBUFFER_HEIGHT/ n);
glEnd();
```

Figure 5.8 Convolution operation on the first part of metrological data shown in Fig5.7

```
glBegin(GL_QUADS);
        glTexCoord2f(0, √n(√n − 1) + 1 );
        glVertex2f(0, 0);

        glTexCoord2f(RENDERBUFFER_WIDTH, √n(√n − 1) + 1 );
        glVertex2f(RENDERBUFFER_WIDTH/ n, 0);

        glTexCoord2f(RENDERBUFFER_WIDTH, RENDERBUFFER_HEIGHT+
                √n(√n − 1) + 1 );
        glVertex2f(RENDERBUFFER_WIDTH/n, RENDERBUFFER_HEIGHT/n);

        glTexCoord2f(0, RENDERBUFFER_HEIGHT+√n(√n − 1) + 1 );
        glVertex2f(0, RENDERBUFFER_HEIGHT/ n);
glEnd();
```

Figure 5.9 Convolution operation on the $(\sqrt{n}(\sqrt{n}-1)+1)$ th part of metrological data

7 Data transferring back to CPU

Since the result has been split into n parts and stored in n textures that are attached to the n colour attachments in the FBO, these split results will be read back to CPU in sequential. If the data in n^{th} texture need to be transferred back,

the instruction of *glReadBuffer*(GL_COLOR_ATTACHMENTn_EXT) is firstly invoked to point to the nth colour attachment in a FBO as shown in Fig.5.7. Then invoking the instriction of *glReadPixels*() will transfer the data in n^{th} colour attachment to an array in CPU's memory.

5.4 Test and Performance Evaluation

A primitive surface that is sampled at 1024×1024 points, and has 2D Gaussian filtering applied was used to validate the acceleration performance of the proposed GPGPU framework. The window radius of 2D Gaussian filter along x and y directions are both 50, so, the width and height of Gaussian filtering window is 101×101. The graphics card used for test is Nvidia's GeForce 7900 GTX. Figure 5.10 shows the primitive measured surface.

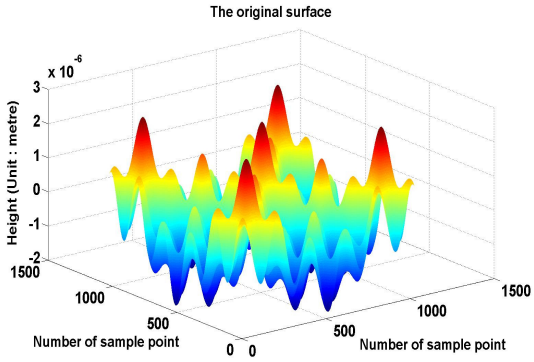


Figure 5.10 A primitive surface profile

5.4.1 Test Results

Fig.5.11 and Fig.5.12 show the results obtained by MATLAB simulation kit and the developed GPGPU programming respectively, from which it can be seen that GPGPU program obtains the same surface profile obtained from MATLAB simulations. It is noted that the filtered surface has the size of 1124×1124.

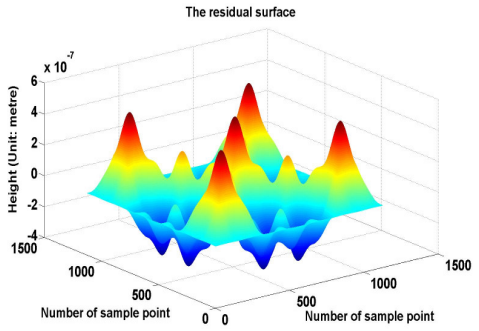


Figure 5.11 Result of Gaussian filtering issued by MATLAB simulations

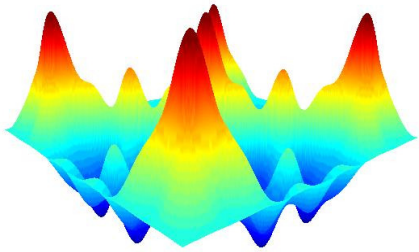


Figure 5.12 Result of GPGPU-based Gaussian filtering

To verify computational efficiency of the developed GPGPU programming framework, the run time of the main stages of the GPGPU framework and the run time of the MATLAB simulation programs, running on the same computer are listed in Table 5.1. It is noted since GeForce 7900 GPU supports the maximum of 4 colour attachments in a FBO, the measured data of primitive surface is therefore split into 4 blocks to be processed by Gaussian filtering when the 4 data blocks are transferred back to CPU sequentially.

Table 5.1 Processing time of GPGPU program and MATLAB simulation

	GPGPU	MATLAB
Data from CPU to GPU	426ms	Not specified
Convolving with Gaussian filter	410 ms	4940ms
Data from GPU to CPU	973ms	Not specified

Table 5.2 lists the comparison of run time between the solutions using no data splitting and when dividing the measured data into 4 blocks for processing. The solution of no data splitting means just a dynamic texture is used to store the whole result of filtering, so, just one of the colour attachments in a FBO is used.

Table 5.2 Processing time of solutions with data dividing and without dividing

	Data splitting	No data splitting
Data from CPU to GPU	426ms	426ms
Convolving with Gaussian filter	410 ms	403 ms
Data from GPU to CPU	973ms	4986ms

5.4.2 Performance Evaluation

Although there is often a degree of latency for data transfer between the CPU and GPU, the proposed GPGPU solution is still proven an effective computing platform for accurately profiling a filtered surface defined in stylus-based surface metrology. If just considering the convolution operation carried out on GPU, the proposed GPGPU framework can achieve a 12× speed-up factor. As a whole application, the data transfer between CPU and GPU should also be considered and the cost of these operations should also be taken into account for performance evaluation. It can be seen from Table 5.2 that transferring massive data (e.g., data in a texture with size of 1124×1124) from GPU back to CPU to visualize the filtered data is a bottleneck in the proposed GPGPU framework. This is caused by the limited bandwidth of PCI/AGP bus. This problem is partially solved by dividing the data into several blocks to compute and transfer. Its feasibility is validated from the test results shown in table 5.2. It can be seen that the time for transferring data back to CPU has decreased from 4986ms to 973ms since the strategy of data splitting is employed, which therefore improves the speed-up factor of the whole GPGPU framework. For example, it is observed in table 5.1 that the GPGPU framework, that includes the stages of data transferring between CPU and GPU, achieves a speed-up factor of 2.73 comparing to the

Matlab-based simulation software. It is believable that with the increase of bandwidth of PCI/AGP bus (the bandwidth of new generation of PCI-Express bus is up to 6Gb/s), the proposed GPGPU framework can achieve a greater acceleration performance.

5.4.3 Accuracy Analysis

Although Fig.5.11 and Fig.5.12 prove that the developed GPGPU programs can obtain the same surface profile as the one obtained by MATLAB simulations, it is essential to evaluate the accuracy gap between the computation results of CPU programs and GPU programs. For the Matlab programs, the minimum value of filtered data is -3.0152e-007 while the maximum value of filtered data is 5.3329e-007. If expressed in the form of absolute value, then the minimum and maximum values are 3.2637e-013 and 5.3329e-007.

The accuracy evaluation is issued by the following two forms:

- The maximum difference of the results obtained by Matlab program and the proposed GPU program in the term of absolute value.
- The maximum difference in the term of percentage that is expressed by

$$Gap = \frac{\left|\mathrm{Re}\,sult\ of\ \ GPU\ \ program - \mathrm{Re}\,sult\ of\ \ Matlab\ \ program\right|}{\left|\mathrm{Re}\,sult\ of\ \ Matlab\ \ program\right|} \times 100\% \qquad (5.13)$$

The latter is more meaningful because the results in this case study is up to the level e-013, thus the maximum difference in the term of absolute value can't efficiently evaluate the accuracy of the results of GPU's program. It has been proved that the maximum difference in the term of absolute value is 3.2305e-012 and the maximum difference in the term of percentage is 8.58%. The latter gap shows that the proposed GPU program can obtain a satisfactory accuracy when processing the data with much small values.

5.5 Summary

The devised GPGPU framework for legacy graphics cards and shader models have been tested and evaluated in this chapter. The application comes from a real-world demand on faster processing speed for issuing the filtering algorithms in stylus-based surface metrology. The developed solution is built on the basis of the analysis over the characteristics of filtering algorithms and their breakdown components. The implementation details such as using fragment programs for filtering, data scattering, and massive data splitting were discussed thoroughly in terms of functions, routines, syntax and semantics. The developed solution has been specifically tested on the 2D Gaussian filtering operation which is a classical process used in 3D surface topography analysis. The test results show that comparing with the runtime of the MATLAB simulation program on a fixed sized of data, the entire GPGPU solution (that is including data loading and CPU to GPU cross-border operations), still achieves a near 300% speed-up factor. If only measures the actual computation part, the speed-up factor will reach 15x. At the same time, the mean errors between the GPGPU program and its "C-language" counterpart is well within the data accuracy specifications. Therefore, the compiled GPGPU program speeds up the filtering process substantially while maintaining the filtering quality, which proved the practicability and validity of the proposed programming model for filtering algorithms used in surface metrological data processing. Chapter 6 will tackle another important application area, image denoising, using the GPGPU framework to assess its flexibility.

Chapter 6 Parallel Implementation on Wavelet-based Image Denoising

The discrete wavelet transform (DWT) has been extensively used for image compression and denoising in image processing and computer vision. However, the intensive computation required by the DWT due to its inherent multilevel data decomposition and reconstruction operations, brings a bottleneck that drastically reduces its performance and implementations for real-time applications when processing large size digital images and/or high-definition videos. Although various software-based acceleration solutions, such as the lifting scheme, have been devised and achieved a higher performance in general, the pure software accelerated DWT still struggle to cope with the demands from real-time and interactive applications.

Following the previous case study, this chapter presents another application of the devised GPGPU programming framework to obtain a parallel computing solution for the wavelet-based image denoising operation. The proposed solution can be readily incorporated with different forms of DWT by customising the parameter of the wavelet kernel in the form of pixel shaders. Experiment results show that the developed GPGPU solution gains applicability in data parallelism and satisfaction performance in acceleration.

6.1 Wavelet-based Denoising

Fourier transform expresses a signal as the sum of a series of sines and cosines of the so-called Fourier expansion, so that the amplitude of different sines and cosines represents the signal's energy distribution in the frequency domain. This is the fundamental reason that Fourier transform is predominantly used to analyse a signal in frequency domain, i.e, to obtain frequency resolution for signal analysis and processing. The filtering algorithms investigated in Chapter 5 are

actually based on the Fourier transform. However, the main limitation of the Fourier transform is that it only retrieves the solutions of signal processing in the frequency domain but not in the time domain. As a result, although one solution based on Fourier transform can generate the information of all the frequencies existed in a signal, it will be impossible to tell when they are incurred. To overcome this problem, wavelet transforms have been discovered and improved in the past 5 decades, which are capable of representing a signal in both the time and the frequency domain at the same time.

The wavelet transforms are usually classified into two categories, continuous wavelet transform (CWT) and discrete wavelet transform (DWT). Due to its feature of obtaining multi-resolution analysis results both in the frequency and the time domain, wavelet transforms, especially the DWT, have become an important tool in image processing such as image denoising. When the DWT is applied in image denoising, implementation involves the following three processing phases (Bovik, 2005):

1) Decomposition

Select a suitable base wavelet and a decomposition level to generate the approximation and detail coefficients of a noisy image at the chosen level.

2) Thresholding

For each level, to generate a threshold and implement it through hard/soft thresholding on the detail coefficients.

3) Reconstruction

Re-calculate for reconstructions using the modified coefficients of various levels.

6.1.1 Analysis of the Wavelet Transform

For a continuous, square-integrable function $f(t)$, its *continuous wavelet transform* (CWT) is defined as the sum over all time of the signal multiplied by scaled, shifted versions of the wavelet function ψ (Ocak, 2008):

$$C(scale, translation) = \int_{-\infty}^{+\infty} f(t)\psi(scale, translation, t)dt \qquad (6.1)$$

CWTs operate on every possible scale and translation over a signal spectrum. However, the calculation of coefficients at every scale and translation is a substantial body of work that often generates a huge amount of data. In addition, the signal processing instructions implemented in computer programs must divide the continuous signals into a series of discrete signals for the digitalized processing. The discrete wavelet transform (DWT) uses a specific subset of scale and translation values where the chosen scale and translation are based on the powers of two, which is the so-called *dyadic* scales and translations. In this case, the wavelet analysis is much more efficient but just as accurate. When it is implemented, the DWT of $f(t)$ is calculated by passing $f(k)$ that is the discrete expression of $f(t)$ through a series of low-pass (LP) and high-pass (HP) filters respectively. Following the low-pass and high-pass filtering, the output signal of each filter will then be downsampled according to the ratio of 2, which produces the approximation and the detail coefficients of the input signal respectively. The approximation coefficients represent the high-scale and low-frequency element in a signal, and the detail coefficients represent the low-scale and high-frequency element (Ocak, 2008). The decomposition process can be iterated, with successive approximation coefficients being generated in turn so that a signal can be decomposed into many lower resolution elements.

The inverse discrete wavelet transform (IDWT) is used for reconstructing the original signal. It involves two distinctive operations of upsampling and filtering. Upsampling is the process of lengthening a signal component by inserting zeros between samples. The filtering part of the reconstruction process also consists of a series of LP and HP filters which are associated with the decomposition filters in DWT. These form a system of what is called the *quadrature mirror filters* to guarantee reproducing the original signal accurately. Fig.6.1 illustrates a multi-level DWT and IDWT of a signal with bandwidth F (Ocak, 2008).

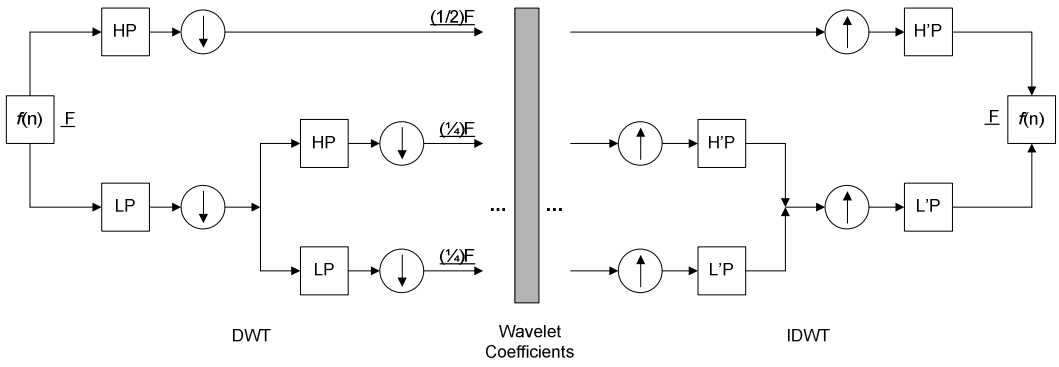


Figure 6.1 Multi-level DWT and IDWT

6.1.2 Thresholding Strategy

A noisy signal $f(k)$ is commonly modeled as the following form:

$$f(k) = s(k) + e(k) \qquad (6.2)$$

where $s(k)$ is the true signal which is often a low frequency or stationary component in the practical implementation. $e(k)$ is the actual noise, which is usually a high frequency term that contains many high frequency details. As stated by Bovik (Bovik, 2005), the general wavelet denoising procedure consists of three steps: forward transformation of the signal to the wavelet domain; modifying the wavelet coefficients; and inverse transformation to the native domain. The wavelet coefficients modification is determined by a thresholding strategy that has been extensively researched. The most practical thresholding methods were mainly initiated by the work of Birgé and Massart (Birgé and Massart, 1997; Barron et al, 1999), and Donoho and Johnstone (Donoho and Johnstone, 1995; Donoho and Johnstone, 1998; Donoho et al, 1995).

Based on the work of Birgé and Massart, the thresholding methods used in practice can be classified into the following two categories:

- Scarce High, Medium, and Low (SHML)
- Penalized High, Medium, and Low (PHML)

The SHML methods work as the follows: for a noisy signal that is decomposed to a level J, the approximation coefficients at level J are kept; for a random level i from 1 to J, the n_i largest coefficients are kept in the form stated as formula (6.3).

$$n_i = \frac{M}{(J+2-i)^a} \qquad (6.3)$$

In the above equation, the value of the parameters a and M are determined by the practical applications. The SHML methods can be further classified by the value of parameter a.

For the PHML, a threshold T applied to the detail coefficients for the wavelet case can be generalized as:

$$T = \left| c(t^*) \right| \qquad (6.4)$$

with

$$t^* = \arg\min[-sum\{c^2(k), k < t\} + 2vt(a + \log(\frac{n}{t})); t = 1,...,n] \quad (6.5)$$

In equation (6.4) and (6.5), $c(.)$ is all the detail coefficients of DWT, the coefficients $c(k)$ are sorted in decreasing order of their absolute values, where v is the noise variance. The value of a that corresponds to PHML are in the range of $2.5 \leq a < 10$, $1.5 < a < 2.5$, and $1 < a < 2$ respectively.

Regarding the issue of denoising, Donoho and Johnstone have devised four different thresholding options (Donoho and Johnstone, 1995; Donoho and Johnstone, 1998; Donoho et al, 1995; Hector et al, 2002):

1. **Rigrsure**

   Rigrsure is an adaptive threshold selection approach using the Stein's unbiased risk estimate criterion. The Rigrsure method defines the threshold level T *by*

   $$T = \sigma\sqrt{2\log_e(N\log_2 N)} \qquad (6.6)$$

   Where N is the number of signal samples; and σ is the standard deviation of the noise.

2. **Sqtwolog**

   The Sqtwolog method defines the universal threshold slightly different from the Rigrsure method in a fixed form

$$T = \sigma\sqrt{2\log_e(N)} \qquad (6.7)$$

3. **Heursure**

   Heursure is a synthesis version of the aforementioned two rules resulting in an optimal forecasting variable threshold.

4. **Minimaxi**

   Minimaxi is a threshold selection scheme using the minimax principle, in which a fixed threshold is selected to obtain the minimum of the maximum mean square error, that is obtained for the worst function in a given set, when compared against an ideal procedure.

All the above thresholding criteria is based on a simplified model that suppose a noise is a Gaussian white noise with standard deviation $\sigma = 1$. For the general cases that noises are unscaled or nonwhite ones, the threshold level should be rescaled according to the aforementioned thresholding criteria. The actual level is commonly obtained by multiplying a rescaling factor by the thresholding value found by the $Sqtwolog$ method. Two rescaling options have been proposed. The first one is to rescale the noise based on coefficients in the first level of the wavelet decomposition. In this option, the Daubechies (Db) 1 wavelet is used to obtain the detail coefficients of decomposition level 1, then the rescaling factor is made to equal to the median values of all absolute values of the detail coefficients. If the median absolute value is equal to 0, the actual threshold value T_s is expressed as:

$$T_s = 0.05 \times max(abs(c)) \qquad (6.8)$$

where $abs(c)$ represents a set of absolute values of detail coefficients at decomposition level 1 of the Db1 wavelet. The first rescaling option then treats the T_s as a global rescaling factor for the whole reconstruction. The second rescaling option, which is best used for nonwhite noise, determines different rescaling factors at various reconstruction levels.

In fact, there are a variety of noises in practical engineering and computer science applications. It is almost impossible to adopt a uniform thresholding strategy to achieve the best performance of denoising for all applications when facing noises with various characteristics. Actually, there are many other thresholding methods specially designed to deal with various forms of noise in specific fields. The performance evaluation of different denoising methods are often carried out by means of Mean Square Error(MSE), Signal to Noise Ratio (SNR), and Peak Signal-to-Noise Ratio (PSNR) (Chicken and Cai, 2005; Azzalini et al, 2005) with many past publications being focused on.

Except the aforementioned precision performance evaluation measures, another vital but often omitted factor also determines the perspective of successful implementation – computational cost. Extremely high computational cost (slow process and long delay to users) will constrain the application of denoising methods that demand a large pool of computer resources. This problem can become very serious when wavelet-based denoising are used for large size noisy images or high-definition videos, for example, satellite image processing and real-time surveillance video processing, or even Augmented Reality applications, in which enormous number of pixels need to be processed in a fraction of a second. In this research, a hardware accelerated solution for wavelet-based denoising has been proposed for alleviating the problem of computational cost and process speed.

6.2 Wavelet-based Denoising on GPU

The amount of computation of wavelet-based denoising are mainly originated from the recursive operations of wavelet decomposition and reconstruction. With the constant increasing power of commodity GPUs, extensive researches on implementing DWT on GPU have been carried out. The most relevant contributions are works from Hopf and Ertl (Hopf and Ertl, 2000) at the University of Stuttgart in Germany, and Wong at the Chinese University of Hong Kong (Wong et al., 2007). Hopf and Ertl developed an OpenGL-based model of the

filter bank scheme (FBS) for implementing DWT on a Silicon Graphics workstation by using high-level OpenGL routines, such as the OpenGL convolution filters. The project had experienced a degree of success on process acceleration. However, the solution has no direct mapping on hardware, which limits the efficiency of the implementation with some of the GPU resources left to spare. For the works of Wong, the convolution, downsampling, and upsampling operations were performed in sequence on a GPU's fragment processors (FPs). Due to the restrictions on GPU programmability at the time and coding facilities, the texture mapping prior to the convolution process was issued by establishing texture lookup tables in which every single texture coordinate is pre-defined in advance by separate CPU programs. The potential benefit of hardware-driven acceleration by using the GPU's hardware interpolators for generating texture coordinates and texture fetch were not fully exploited. This in turn hampers the performance of the consequent FP programs.

Based on the existing research on wavelet-based denoising, the GPGPU programming model proposed in this chapter aimed at seeking further hardware empowered process acceleration for wavelet-based denoising. This was achieved by directly implementing texture fetching using hardware interpolators, which was based on the general programming framework as shown in Fig.4.3. When issuing filtering, kernels for downsampling and upsampling in the stages of decomposition and reconstruction, there is no need to employ any pre-defined values issued by separate CPU routines in advance. Furthermore, filtering and down-sampling operations can be carried out on GPU simultaneously, for instance, to implement the two operations on a single FP to exploit the performance gain from GPU's intrinsic functions.

Considering the fact that GPGPU is hardly a computational panacea and there are still many issues regarding the hardware structure and programming paradigm to be tackled before a proper match against its CPU counterparts becoming a reality. Task partitioning in the proposed programming model that decides which part of the work will be conducted on the GPU and which part should be left to the CPU for the current generation of hardware will be discussed

in detail to promote the acceleration performance of the application developed from the proposed GPGPU programming model for wavelet-based image denoising.

The GPGPU solution for wavelet-based denoising developed in this chapter employs 2D-DWT and 2D-IDWT respectively. 2D-DWT and 2D-IDWT were implemented by applying separate 1D-DWTs and 1D-IDWTs along the horizontal and vertical directions respectively. The decomposition process in 2D-DWT has adopted the common square decomposition method which is depicted as in Fig.6.2, where cA_j, cH_j, cV_j , and cD_j represent approximation coefficients (cA_0 represents original 2D signal), and the detail coefficients along horizontal, vertical, and diagonal orientations (Tenllado et al, 2008).

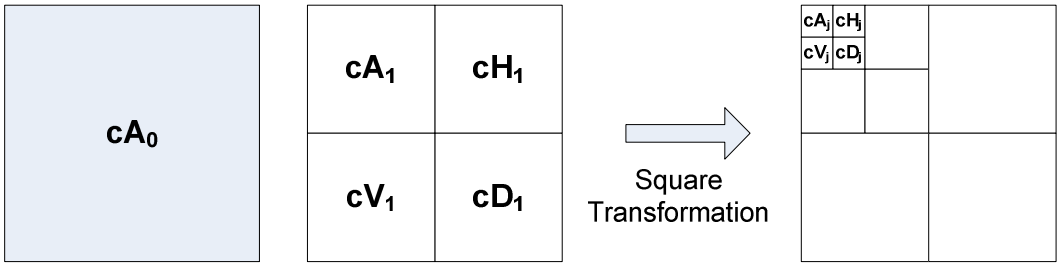


Figure 6.2 The square decomposition scheme

The thresholding approach chosen for denoising has employed the *Sqtwolog* method introduced in Section 6.1.2 to integrate with the global rescale options. As discussed earlier, the global rescaling factor is normally determined by the median absolute values of the detail coefficients obtained by the Db1 wavelet process, in which a sort operation on the absolute values of detail coefficients is essential. The sort operation requires random memory write accessibility, which is often not available from fragment processors on today's GPU in the so-called "scatter" memory operations which have been described in Chapter 5. The GPGPU solution devised in this chapter then assigned the task of thresholding to a CPU, while concentrating GPU resources on issuing the operations of decomposition and reconstruction. The entire programming model corresponding to the solution can be summarized as in Fig.6.3.

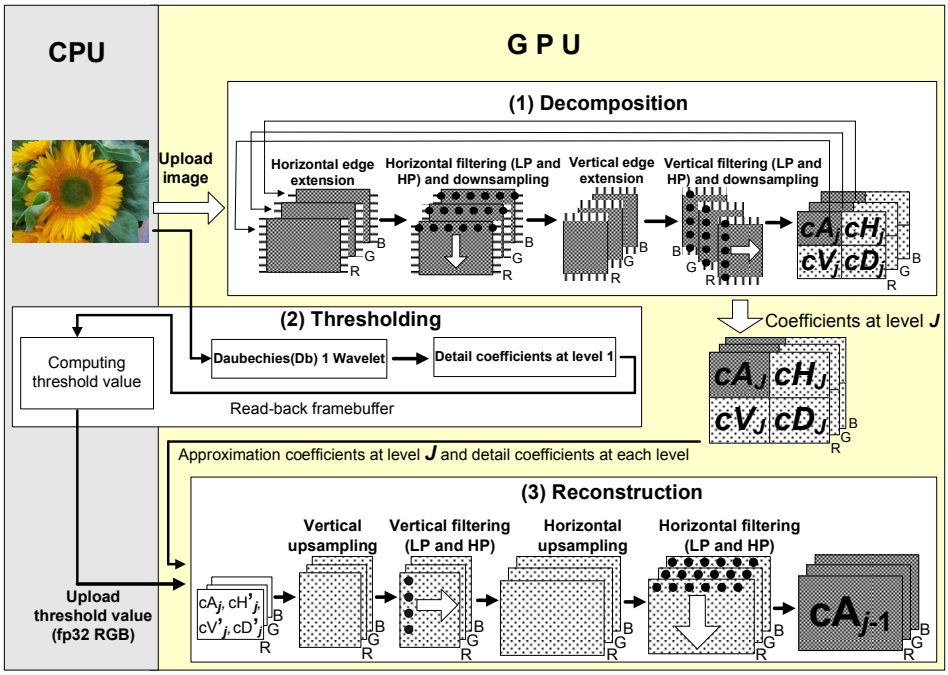


Figure 6.3 The operational model of the GPGPU and wavelet-based denoising

6.3 Technical Specifications of the GPU Implementation

A texture consists of a vector Red-Green-Blue-Alpha (RGBA) floating point values to be stored on a GPU. As a standard practice for image processing, these 4 floating point vectors were used for storing pixels of an image. All the approximation coefficients (cA_j) and the detail coefficients (cH_j, cV_j, cD_j) obtained by deploying the same base wavelet were also stored in the same texture with RGBA four channels. The Framebuffer Objects (FBOs) which has been used for data scatter in Gaussian filtering in Chapter 4 – was employed as an off-screen rendering mechanism for storing intermediate computation results.

6.3.1 Decomposition

There are three main steps concerning the integration of decomposition into the programming model including image edge extension, filtering and sampling. After

investigating common extension schemes that include periodic padding, symmetric padding, and zero padding, as summarized by Strang and Nguyen (Strang and Nguyen, 1996), the proposed GPGPU solution has applied the symmetrical periodic extension for its simplicity as shown in Fig.6.4. In the diagram the extension length L is determined by the kernel length of a filter employed in decomposition.

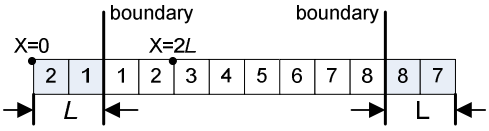


Figure 6.4 The symmetrical periodic extension scheme

Fig.6.5 shows a GPU program snippet for extending the left edge of an image on a GPU. The extended edge consists of the part outside the left boundary as indicated in Fig.6.4. The computational area is specified by an intrinsic OpenGL instruction *glBegin*(*GL_Quads*) for defining an off-screen quad canvas with specified vertex coordinates. The left edge extension was then issued with the following fragment program (FP).

```
fragout_float main(vf30 IN,
                   uniform samplerRECT image,  //image texture
                   uniform float L             //extension length)
{
      fragout_float OUT;
      OUT.col =f4texRECT(image, float2(2L-IN.TEX0.x, IN.TEX0.y));
      OUT.col.a=0.0;
      return OUT;
}
```

Figure 6.5 FP for edge extension

Two separable 1D-DWTs were issued following the edge extension, to enable convolutions between the image texture and the filter kernel for downsampling along the horizontal and vertical dimensions. In this project, the downsampling

was issued by using functions from OpenGL library to control the actual sample intervals in the texture fetching operations. For example, if using the variables *tex_width* and *tex_height* to represent the width and height of an image texture, the convolution between the image texture and the filter kernel along the horizontal dimension for dowsampling can be combined into the following OpenGL instruction sets and FP process, as shown in Fig.6.6 and Fig.6.7.

```
glBegin(GL_QUADS);
{
    glTexCoord2f(                0.0f,                    0.0f);
    glVertex2f    (              0.0f,                    0.0f);
    glTexCoord2f((float)tex_width,                      0.0f);
    glVertex2f  ((float)tex_width/2,                    0.0f);
    glTexCoord2f((float)tex_width,    (float)tex_height);
    glVertex2f  ((float)tex_width/2,  (float)tex_height);
    glTexCoord2f(                0.0f,  (float)tex_height);
    glVertex2f    (              0.0f,  (float)tex_height);
}
glEnd();
```

Horizontal down-sampling according to ratio 2:1

Figure 6.6 OpenGL instructions for controlling filtering and downsampling

When implemented in the proposed GPGPU denoising programs, the filter kernel was stored in the *R* channel of a texture. As shown in Fig. 6.7, a factor of 0.5 for addressing the pixel center when fetching a texture has been adopted.

The operation of filtering and down-sampling along the vertical direction is an analogue to the horizontal ones.

```
fragout_float main(vf30 IN,
                   uniform samplerRECT image,     //image texture
                   uniform samplerRECT filter,      //texture for filter kernel
                   uniform float L          //kernel length
)
{
   float3 sum=float3(0,0,0);

  // Implementing convolution

  for (int i=0; i<L; i++)
  {
    sum += f3texRECT(filter , float2(i+0.5,0.5)).r *
           f3texRECT(image , float2((IN.TEX0.x+i, IN.TEX0.y));
  }

 fragout_float OUT;
 OUT.col = float4(sum, 0.0);
 return OUT;
```

Figure 6.7 Corresponding fragment program for filtering in horizontal dimension

6.3.2 Thresholding

As highlighted in Fig.6.3, a critical step in the thresholding stage is to implement a Db1 wavelet on a GPU, and to retrieve the corresponding coefficients from the GPU's framebuffer transferring them to the CPU's memory to generate a rescaling factor. The task performed on the CPU is the sorting operation. This back-and-forward process is the most time-consuming step in the entire process for the reasons stated in Section 6.2.1.

Although some researchers claimed to have developed GPU-based sorting libraries for implementing the sorting algorithms at 16-bit and 32-bit floating precision with a performance comparable to a CPU, it was noticed that the implementations still struggle to sort arrays with non power-of-two image sizes (

Govindaraju et al., 2008). To ensure adaptability, sorting operations in the devised programming model in this project were still performed on the CPU. After the threshold values were computed, they were downloaded to the GPU to modify the detail coefficients obtained in the previous stage of decomposition.

6.3.3 Reconstruction

The reconstruction phase in the model is an inverse of the decomposition, which is achieved by applying 1D inverse DWT vertically and horizontally in turn. For reconstruction, the process started from the lowest decomposition level – referred as J; and then the approximation coefficients cA_j, and the modified detail coefficients $(cH_j^{'}, cV_j^{'}, cD_j^{'})$ would be upsampled and filtered by corresponding reconstruction filters along vertical and horizontal dimensions respectively. The four computational results originated from cA_j, $cH_j^{'}, cV_j^{'}, cD_j^{'}$ would then be synthesized to form the approximation coefficients of the upper level j-1. After a series of recursive computation, the ultimate denoised image can be obtained. Fig.6.8 and Fig.6.9 illustrate the upsampling operations at the image size of *tex_width* and *tex_height*.

```
glBegin(GL_QUADS);
{
        glTexCoord2f(                0.0f,                0.0f);
          glVertex2f  (              0.0f,                0.0f);
        glTexCoord2f((float)tex_width,                   0.0f);
          glVertex2f  ((float)tex_width,                 0.0f);
        glTexCoord2f((float)tex_width,  (float) tex_height);
          glVertex2f  ((float)tex_width,  (float) tex_height);
        glTexCoord2f(                0.0f,  (float) tex_height);
          glVertex2f  (              0.0f,  (float) tex_height);
}
```

Figure 6.8 OpenGL commands that implement upsampling along the vertical dimension

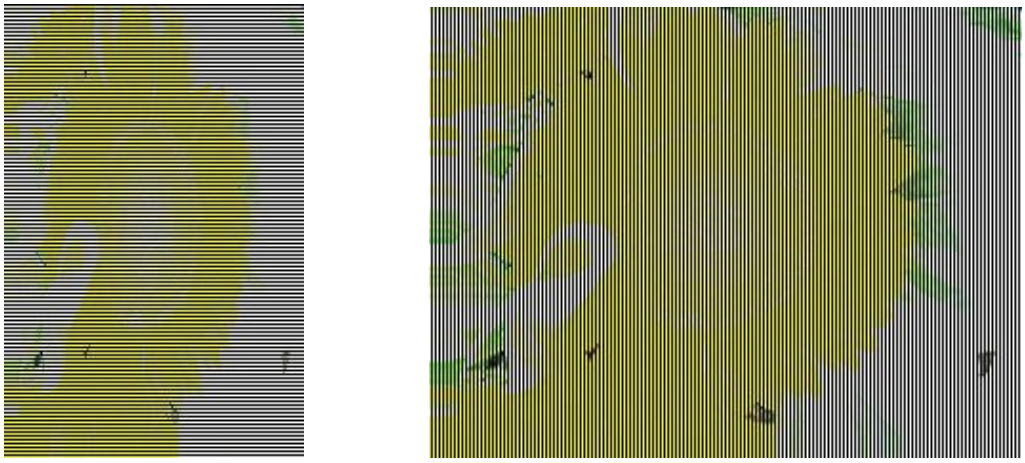
```
fragout_float main(vf30 IN,
                   uniform samplerRECT image        //image texture
)
{
    float3 sum=float3(0,0,0);
    int y=floor(IN.TEX0.y);

    if (y%2==0)
    {
        sum=f3texRECT(image, float2(IN.TEX0.x, floor(IN.TEX0.y/2)+0.5);
    }

    fragout_float OUT;
    OUT.col = float4(sum, 0.0);
    return OUT;
}
```

Figure 6.9 Fragment program for upsampling along vertical direction

The effects of vertical upsampling and horizontal upsampling are displayed in Fig. 6.10.



(a) Vertical upsampling (b) Horizontal upsampling

Figure 6.10 The effect of upsampling

6.4 Test and Performance Evaluation

Our proposed GPGPU solution for wavelet-based denoising was tested on a PC equipped with Nvidia's GeForce 7900 GTX. Among the various base wavelets, the Db4 base wavelet was tested as the denoising wavelets in the programs to validate the programming model's functionality.

First the results of DWT at various decomposition level are shown to illustrate the coefficients, i.e. cA_j, cH_j, cV_j, and cD_j. The denoising effects on noisy images at various reconstruction level are then demonstrated. In addition to the denoising effect to be benchmarked, another key performance is the computational efficiency that is often exponentially linked to a specified base wavelet and image size. For various base wavelets, the kernel length of the low-pass or high-pass filter is normally less than 20, for example, the kernel length of Db4 base wavelet is 8, therefore the image size becomes the dominant factor that influences the computational efficiency of the wavelet-based denoising.

6.4.1 Results of Decomposition

Fig. 6.11 shows a noisy image of night-sky cityscape in the size of 1280×1024. This image consists of a large number of white light dots in the background of blue night-sky, which is helpful to clearly illustrate the coefficients at different decomposition level when using the DWT.

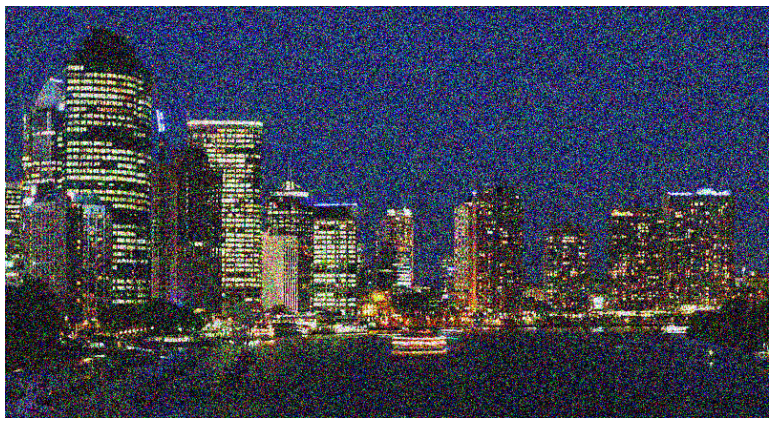


Figure 6.11 Noisy night-sky cityscape

Fig.6.12-6.14 show the coefficients at decomposition level 1, 2, 3 by employing the proposed GPGPU programs to issue the DWT.

Figure 6.12 Coefficients at decomposition level 1

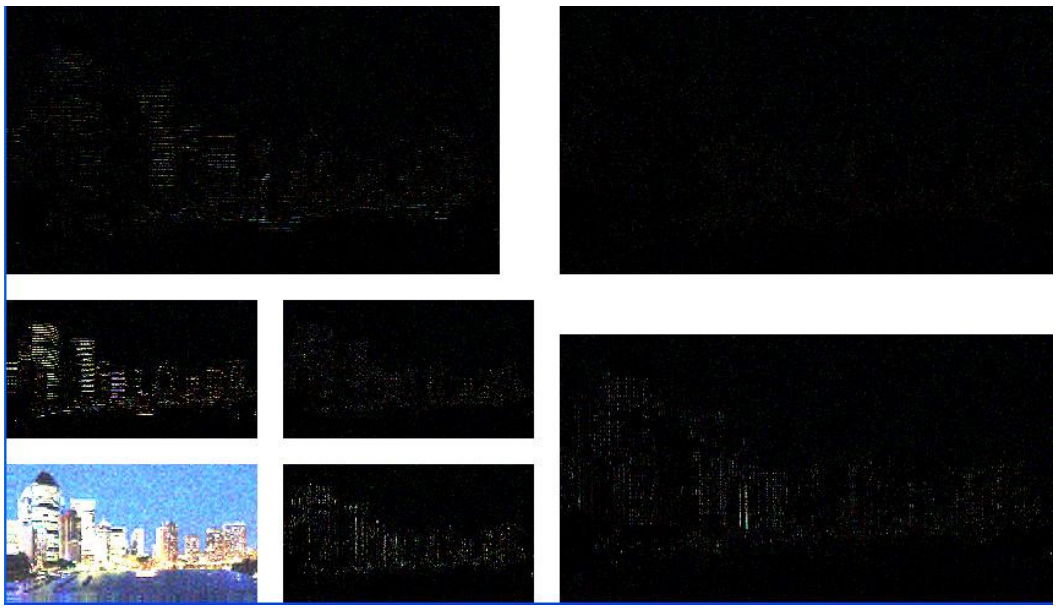


Figure 6.13 Coefficients at decomposition level 2

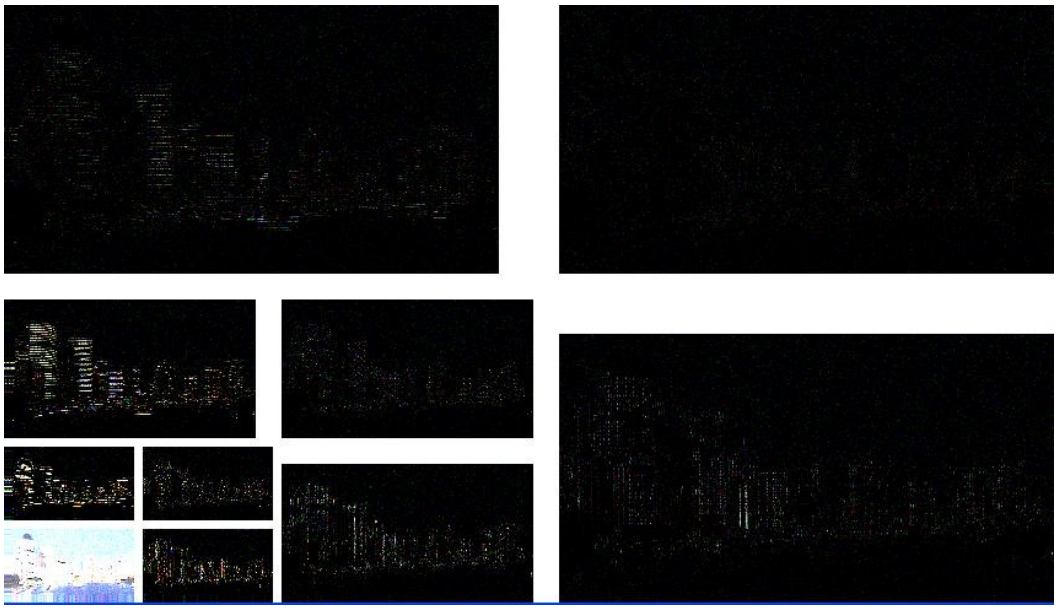


Figure 6.14 Coefficients at decomposition level 3

6.4.2 Quality Analysis

Three noisy image samples that contain nonzero-mean white noise were tested sequentially. Fig.6.15 shows the noisy image with the size of 1024×960.

Figure 6.15 Noisy image (1024×960)

For the noisy image, shown in Figure 6.15, the maximum number of wavelet decompositions chosen was 4. The synthesized images at different reconstruction level corresponding to the approximation coefficients (cAs) at the

reconstruction according to the modified detailed coefficients are illustrated in Fig. 6.16.

(a) cA_3

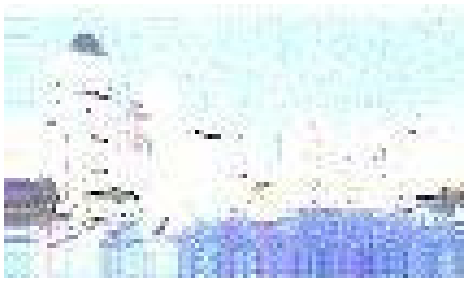
(b) cA_2

(c) cA_1

(d) The ultimate denoised image (cA_o)

Figure 6.16 Denoising effects using the Db4 wavelet

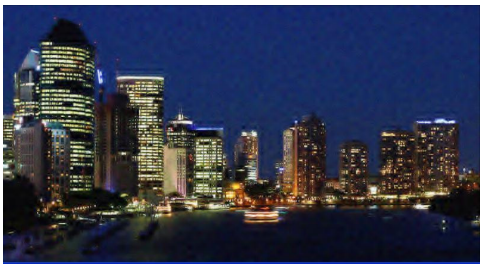
Another noisy image is the night-sky cityscape, with a size of 1280×1024, which is depicted by Fig. 6.12. The maximum number of wavelet decompositions chosen for this image was also 4. Fig. 6.17 shows the modified approximation at each reconstruction level and the ultimate denoising image.



(a) cA_3

(b) cA_2

(c) cA_1



(d) The ultimate denoised image (cA_o)

Figure 6.17 Denoising effects on the image of night-sky cityscape

Fig. 6.18 shows a noisy image of sunflower with arbitrary curves and edges and with a size of 2048×2048.

Figure 6.18 The noisy image of a sunflower

The maximum number of wavelet decompositions chosen for this image was 3. Fig. 6.19 shows the modified approximation at each reconstruction level and the ultimate denoising image.

(a) cA_2

(b) cA_1

(c) The ultimate denoised image (cA_o)

Figure 6.19 Denoising effects on the image of sunflower

It was observed that during the process of reconstruction, much of the useful image details were resorted with the noisy signals in the background region reduced. In real applications, noise rejection and oversmoothing are often a dilemma which sometimes causes unsatisfactory effects such as edge blurring. There exists a trade off between these two factors when choosing and balancing a donoising approach. In general, as indicated in Fig.6.16, Fig.6.17 and Fig.6.19 that the wavelet-based denoising achieved a good performance on GPU and restored a substantial percent of strong edges which can be seen from the reconstructed images, which further approves the effectiveness of wavelet for image denoising.

6.4.3 Evaluation on Computational Efficiency

- **Comparison with the software-based wavelet denoising**

The computational efficiency of the developed GPGPU programming model for image denoising was evaluated against the acceleration factor by comparing with software-based wavelet implementations on a Pentium IV 2.6 GHz PC equipped

with Nvidia's GeForce 7900 GTX graphics card. Except the aforementioned three noisy images, two noisy images with sizes of 512×512 and 800×600 were processed to evaluate the acceleration performance of the developed GPGPU solution. Table 6.1 lists the comparison results regarding the overall operational time on software-based wavelet denoising and on the GPGPU denoising programs with the accelerating factors computed.

Table 6.1 Runtime comparisons on different image size (in ms)

Image size	512×512	800×600	1024×960	1280×1024	2048×2048
Software-based	2125ms	2703ms	6094ms	7562ms	26234ms
GPGPU-based	222ms	348ms	725ms	1275ms	3324ms
Accelerating factor	9.6	7.8	8.4	5.9	7.9

To evaluate the acceleration performance of the whole GPGPU programs on the distinctive decomposition and reconstruction stages, a further breakdown of computational time with regard to each stage is listed in Table 6.2 with a Db4 wavelet as a chosen target. It was envisaged that the GPGPU programming model would have a satisfactory performance especially in the decomposition stage. On the other hand, the accelerating factor for the reconstruction is much lower than the decomposition. The reason for that is the need to obtain the approximation coefficients at level j (cA_j). The approximation and detail coefficients at level j+1 (cA_{j+1}, cH_{j+1}, cV_{j+1}, and cD_{j+1}) here to be upsampled and filtered in sequence in the solution, which increases the computational cost and results in the reduced acceleration performance comparing to the decomposition. In fact, the operations on all coefficients in the reconstruction stage are the same. Therefore a better mechanism for texture mapping in the programming model, in order to enable all coefficients in the stage of reconstruction to be processed in parallel, needs to be researched in the future.

Table 6.2 Breakdown of computational time (in ms)

Image size	512×512	800×600	1024×960	1280×1024	2048×2048
Software-based decomposition	423ms	658ms	1596ms	1923ms	5862ms
GPGPU-based decomposition	15ms	16ms	31ms	94ms	158ms
Accelerating factor	28.2	41.1	51.5	20.5	37.1
Software-based reconstruction	516ms	798ms	2112ms	2670ms	10968ms
GPGPU-based reconstruction	125ms	171ms	391ms	593ms	2000ms
Accelerating factor	4.1	4.7	5.4	4.5	5.5

Since most of the tasks in the stage of thresholding are actually carried out by the CPU, the impact of this workload distribution on the GPGPU programs has also been evaluated. Table 6.3 lists the runtime of key steps in thresholding operation, which includes issuing the Db1 wavelet decomposition on a GPU, transferring coefficients of the Db1 decomposition at level 1, and sorting the coefficients to compute the median absolute values for generating the rescale factor. It can be observed that most of the run-time latency was caused by the reading of coefficients back from GPU's framebuffer and the sorting operation on CPU. Table 6.4 lists the proportion of the runtime of these two tasks in the entire GPGPU solution. It can be seen that the runtime of these two tasks dramatically increases along with the image size.

Table 6.3 Runtime of key steps in thresholding (in ms)

Image size	512×512	800×600	1024×960	1280×1024	2048×2048
Issue Db1 Decomp.	3ms	5ms	9ms	11ms	36ms
Read-back framebuffer	31ms	47ms	109ms	359ms	500ms
Sort operation	31ms	62ms	125ms	156ms	562ms

Table 6.4 Proportional benchmarking of GPU-CPU data transfer latency

Image size	512×512	800×600	1024×960	1280×1024	2048×2048
Latency of GPU-CPU uploading	62ms	109ms	234ms	515ms	1062ms
Total time cost	222ms	348ms	725ms	1275ms	3324ms
Proportion of the cross border delay	27.9%	31.3%	32.3%	40.4%	31.9%

- **Comparison with Wong's solution**

The performance of the developed GPGPU solution was also compared with another GPU-based solution devised by Wong's group at the Chinese University of Hong Kong. The core of Wong's solution is to establish lookup tables along both the horizontal and vertical directions, to store the texture coordinates for texture fetching used in the fragment programs for DWT and IDWT at different level. The lookup tables were initialized by a program running on CPU.

Adopting the same approach for thresholding operations as was explained in Section 6.3.2, a series of experiments for image decomposition and reconstruction that employed Wong's method were also carried out. Table 6.5 lists the runtime performances regarding the sub-stages of decomposition, reconstruction and lookup table initialization.

Table 6.5 Runtime of sub-stages on various image sizes using Wong's method (in ms)

Image size	512×512	800×600	1024×960	1280×1024	2048×2048
Decomposition	13ms	25ms	56ms	149ms	248ms
Reconstruction	16ms	31ms	59ms	154ms	251ms
Lookup table initialization	235ms	360ms	901ms	1479ms	3034ms

Comparing the results shown in Table 6.5 with those in Table 6.2, it is observed that for the GPGPU solution devised in this project, the runtime of image decomposition was less than that of the Wong's method. However, using the Wong's solution, the runtime of image reconstruction is faster than the proposed GPGPU solution. Based on the processing flow of image reconstruction depicted in Fig.6.3, it can be seen that the processes of upsampling and filtering in IDWT are actually issued by different fragment programs running in multiple passes on GPU. Snippets of the fragment programs have been shown in Fig. 6.7 and Fig.6.9 respectively. In comparison, by using Wong's method, the upsampling and filtering can be issued by the same fragment program based on the pre-built texture coordinates lookup tables. In Wong's method, these two processes can be implemented simultaneously. This is the reason why the runtime of image reconstruction using Wong's method is faster than the proposed solution. However, the Wong's approach requires the constant construction of processing phase related lookup tables, which can be a time-consuming process to implement. Table 6.5 also lists the runtimes for establishing the texture coordinates lookup tables when using Wong's method, which dominates the application's runtime.

Table 6.6 lists the comparison results regarding the overall runtime performances of the GPGPU solution developed in this project. It can be seen that the overall processing time of the proposed solution is less than that of Wong's. Another advantage of this solution is that it only allocates textures for image and filter kernels which are essential for the GPU operation. The additional textures to store the lookup tables are unnecessary during the operation cycle; hence spare the hosting CPU program's involvement completely. This design further improves the GPU's memory usage when issuing wavelet-based denoising on large size digital images and/or high-definition videos.

Table 6.6 Runtime comparisons on different image size (in ms)

Image size	512×512	800×600	1024×960	1280×1024	2048×2048
Wong's solution	284ms	466ms	1103ms	1877ms	4231ms
The new method	222ms	348ms	725ms	1275ms	3324ms

6.5 Summary

In this chapter, a GPGPU solution based on the proposed programming model has been developed and evaluated for wavelet-based denoising. A number of popular signal denoising algorithms and techniques have been implemented in this chapter. The overall performance of the proposed GPGPU solution has been assessed in terms of the visual quality and computational efficiency against the set criteria. Through the quantitative tests on noisy images scaled from 512×512 to 2048×2048, the solution has achieved speed up factors in the range between 5.9 to 9.6 as shown in Table 6.1. It harnesses the parallel processing ability and programmability of modern consume-level graphics hardware for accelerating image processing. At the same time, the acceleration performance of the proposed GPGPU solution was also compared with the other researcher's GPGPU solution such as Wong's GPGPU approach. It is proved the newly presented GPGPU solution can obtain a shorter runtime, and the solution is particularly effective when the denoising approach is issued on a large volume of noisy data, as depicted by Table 6.6.

On the other hand, it has been observed during the experiments that although modern GPUs are fast co-processors, they are not designed to implement all the tasks and to replace the CPU, some tasks such as sort operation on random-size array aren't suitable to be issued on the legacy GPUs. But through careful balancing of the allocation of computational tasks between CPU and GPU, the computation efficiency can still be greatly improved. The following chapter will examine the latest GPU structure and its programming tools through implementing another popular data processing technique for surface metrology.

Chapter 7 Unified Pipeline Model-based Parallel Processing for Spectral Scanning Interferometry

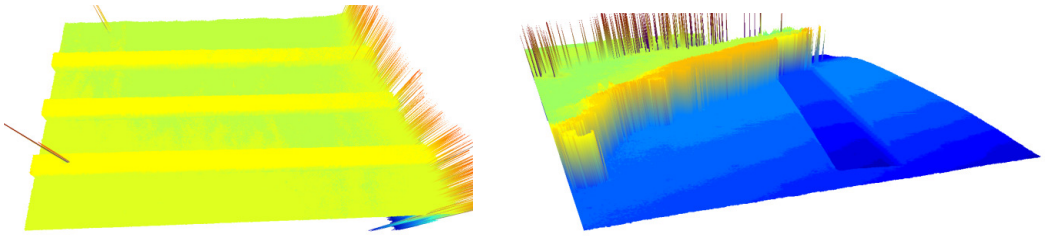
In this chapter, the unified pipeline model-based parallel processing for measurements obtained from the optical interferometry is discussed. Different from the approach discussed in the last two chapters, which heavily relied on the legacy pipeline structures, CUDA is used in this experiment for performing Fast Fourier Transform (FFT) and data analysis through employing the latest unified pipeline structure.

7.1 Surface Metrology Using Optical Spectral Scanning Interferometry

Traditional surface metrology mainly focuses on the abstraction of roughness and waviness from a rough surface, which is achieved by distinguishing these components in different frequency segments in frequency domain, hence various filtering algorithms, such as Gaussian filtering, Gaussian regressive filtering, and Spline filtering, are employed to obtain the roughness and waviness. Except rough surface, there is also a special kind of surface called a structured surface that is characterized by various step and grooves (Reilly et al., 2006; Singleton et al., 2002). For measuring this kind of surface, optical interferometry has been widely explored due to the advantages of non-contact and high accuracy.

The use of ultra precision structured surfaces, which are now measured at the nano scale, is rapidly increasing applications rang across optics, Si wafers, hard disks, MEMS/NEMS, micro fluidics and the micro moulding industries (Reilly et

al., 2006; Singleton et al., 2002). Figure 7.1 (a) and (b) shows 3D profiles of structured surfaces which also involve some environmental noises in measure process.



(a) (b)

Figure 7.1 Profiles of structured surface characterized by step and grooves

Although so many industries critically rely on ultra precision structured surfaces, there is however a fundamental limiting factor to the manufacture of such surfaces, namely the ability to measure the product accurately and rapidly in the manufacturing environment. Traditional mechanical scanning of the probe head or the specimen stage limits the accuracy and causes invalid results. As a result, non-contact optical interferometry was introduced to measure the structured surface, which is commonly called optical spectral scanning interferometry (OSSI) (Su and Shu, 1991; Yamamoto and Yamaguchi, 2000).

7.1.1 The Principle of Surface Metrology Using Monochromatic Interferometry

The essence of OSSI is using various monochromatic lights to generate interference signal at each scanned point on a measured surface. The principle of surface metrology using monochromatic interferometry will be briefly introduced before the introduction to OSSI.

Surface metrology using menochromatic interferometry makes use of optical path difference (OPD) to profile a structured surface (Huang et al., 1988). OPD's

conception can be explained using Figure 7.2 that illustrates a classical Michelson interferometer.

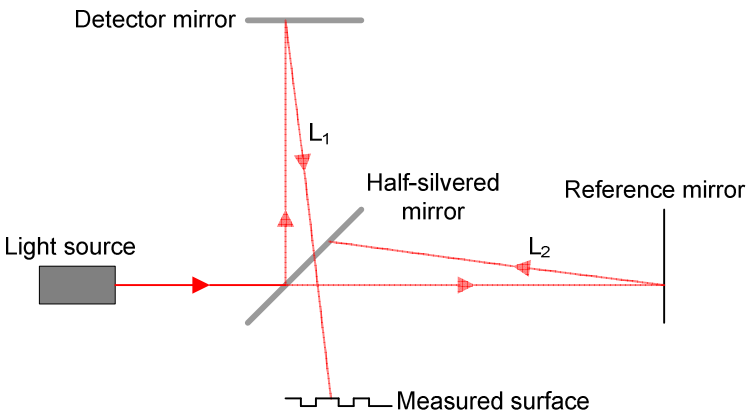


Figure 7.2 The optical path in a interferometer

The interferometer in Figure 7.2 is composed of a half-silvered mirror, detector mirror, and reference mirror. The half-silvered mirror reflects part of the light from light source to the detector mirror and directly transmits part of the light to reference mirror. The light reflected by detector mirror then transmits through half-silvered mirror to measured surface, which forms optical path L_1 with length l_1. The light reflected by reference mirror transmits to half-silvered mirror and forms optical path L_2 with length l_2. In the interferometer, l_2 is constant while l_1 is shifted with the topography of measured surface. The length difference of L_1 and L_2, which is the optical path difference represented by l_1- l_2, determines the phase of the light radiant on the measured point at the surface by the following equation, where Φ represents phase and λ represents wavelength of the light (Huang et al., 1988).

$$\Phi = 4\pi(l_1 - l_2)/\lambda \qquad (7.1)$$

λ in Eq.(7.1) can be constant by using a monochromatic interferometer, OPD can be therefore acquired by measuring the phase of the light at different scanned point. But monochromatic interferometry brings the problem of $2pi$ phase

ambiguity (Schwider and Zhou, 1994; Schnell et al., 1996), which is illustrated in Figure 7.3.

Figure 7.3 Illustration of 2pi phase ambiguity

In Figure 7.3, it is possible that the respective OPD of the interferometer at scanned point A and B will result in the phenomenon that the phase of the light transmitted to point A, which is represented by Φ_A, differs from the phase of the light transmitted to point B, which is represented by Φ_B, by an integer multiple of 2π. In this case, Φ_A and Φ_B are evaluated to be same, hence the OPD of the interferometer at scanned point A and B are also evaluated to be same. Points A and B are therefore viewed as being locating at the same height in the height map of the surface, which results in the step or grave where point B is located being ignored.

To solve the problem of 2*pi* phase ambiguity, optical spectral scanning interferometry (OSSI) was employed for structured surface metrology (Dai and Katuo, 1998; Hayes, 2002; Joo and Kim, 2006). Compared with monochromatic interferometry using a unique wavelength, OSSI uses light beam including with various wavelength, such as white light, to measure surfaces. Surface topography measurements are based on the phase and wavelength shift, which therefore overcomes 2*pi* phase ambiguity.

7.1.2 The Principle of Optical Spectral Scanning Interferometry

Optical spectral scanning interferometry (OSSI) uses light beam such as white light to measure a surface (Sandoz et al., 1996; Hirai et al., 1999). Suppose the wavelength segment of the light beam is [λ_1, λ_2], the relationship between the

intensity of the interference light at a measure point, for wavelength λ, and the OPD of the measured point can be expressed as (Hlubina, 2002):

$$I(\lambda) = I_1(\lambda) + I_2(\lambda)\cos(2\pi\lambda h) \qquad \lambda \in [\lambda_1, \lambda_2] \qquad (7.2)$$

where I_1 and I_2 are the background intensity and fringe visibility respectively, h is the OPD of the scanned point. In Eq. (7.2) I_1 and I_2 can be viewed as direct current components which vary slowly with wavelength, thus the phase of the interference signal is actually reflected by the cosine function in Eq. (7.2), which is expressed as $\Phi = 2\pi\lambda h$. Also due to the slow variation of I_1 and I_2, there are periodic peak points on the curve of $I(\lambda)$ and the phase shifts of these peaks $\Delta\phi$ satisfy $\Delta\phi = n \cdot 2\pi$. Since OPD of each scanned point on the measured surface is actually constant, the phase shift is intrinsically caused by the wavelength shift which is expressed as (Hlubina, 2002):

$$\Delta\phi = \Delta\lambda \cdot 2\pi h \qquad (7.3)$$

This case is illustrated in Figure 7.4, in which the peaks on curve of $I(\lambda)$ correspond to a constant wavelength shift that causes a 2π phase shift. If using $\Delta\phi_{21}, \Delta\phi_{31}$ and $\Delta\phi_{41}$ to represent the phase shift of peak point 2, 3, and 4 to peak point 1 in Figure 7.4, then $\Delta\phi_{21}, \Delta\phi_{31}$ and $\Delta\phi_{41}$ satisfy the following equation:

$$\Delta\phi_{21} = 2\pi \qquad (7.4)$$

$$\Delta\phi_{31} = 4\pi \qquad (7.5)$$

$$\Delta\phi_{41} = 6\pi \qquad (7.6)$$

For two scanned points with different OPD, their wavelength shift that cause 2π phase shift are also different.

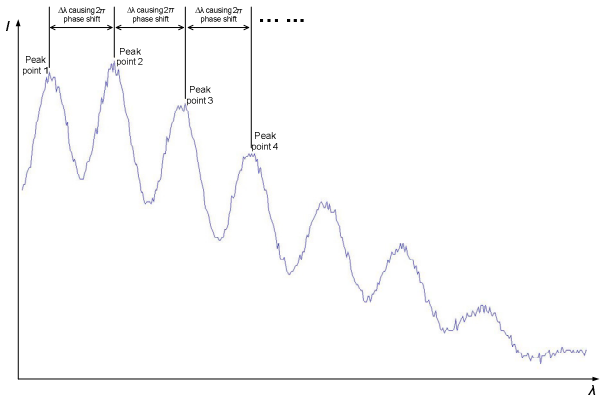
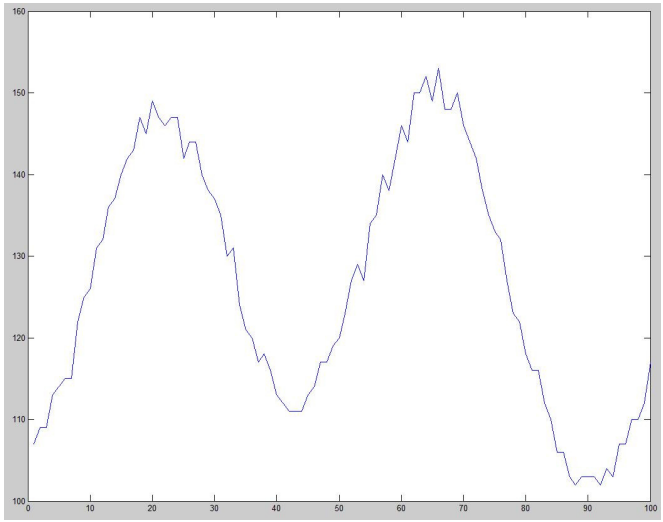
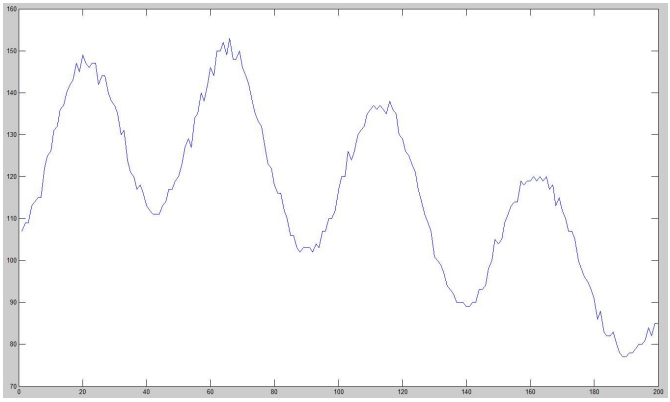


Figure 7.4 Intensity curve of interference signal at a scanned point

OSSI uses a spectrum and generates a series of interference signal at each scanned point on a surface. By analysing the phase shift and wavelength shift of each scanned point at a chosen wavelength segment, the OPD of each scanned point will be acquired, thus the tomography of a surface is profiled. Wavelength number determines the length of wavelength segment in light spectrum used for measurement. Figure 7.4 is the intensity curve corresponding to using 400 lasers with different wavelengths within the light spectrum for scanning, while Figure 7.5 shows the intensity curve corresponding to using 100 and 200 lasers respectively. The curve in Figure 7.5 is ¼ and ½ part of the curve in Figure 7.4.



(a) Intensity curve of wavelength=100



(b) Intensity curve of wavelength=200

Figure 7.5 Intensity curve with different length of wavelength segment

Commonly, the more wavelength that are used, the wider the wavelength segment, and so the more accurate the OPD measurement. But the wide wavelength segment also limits the scanning speed and brings massive increasing in data processing. In fact, monochromatic interferometry can be viewed as an extreme case in OSSI where the wavelength number is equal to 1 and the light spectrum is compressed to a discrete point (Jiang et al., 2006).

7.2 Data Processing in Optical Spectral Scanning Interferometry

The key to calculating a OPD is to calculate the phase shift at a chosen wavelength segment, as stated in Eq.(7.3). Therefore the first step is to calculate the phase distribution within the chosen wavelength segment.

Now rewrite equation (7.2) as (Takeda and Yamamoto, 1994;James et al., 2004):

$$I(\lambda) = I_1(\lambda) + \frac{1}{2} I_2(\lambda)[\exp(i2\pi\lambda h) + \exp(-i2\pi\lambda h)] \qquad (7.7)$$

The Fourier transform of variable λ in this equation can be written as (Takeda and Yamamoto, 1994; James et al., 2004):

$$\hat{I}(f) = \hat{I}_1(f) + \hat{I}_2(f-h) + \hat{I}_2(f+h) \qquad (7.8)$$

where $\hat{I}(f), \hat{I}_1(f)\ and\ \hat{I}_2(f)$ are Fourier transform of $I(\lambda), I_1(\lambda)\ and\ I_2(\lambda)$ respectively. In a practical engineering application, the OPD of the scanned point, i.e. the value of *h* is large enough to separate the three frequency spectrum in Eq.(7.8) from one another, so that the second spectrum $\hat{I}_2(f-h)$ can be filtered out and processed by an inverse Fourier transform. The inverse Fourier transform of $\hat{I}_2(f-h)$ is (Takeda and Yamamoto, 1994; James et al., 2004):

$$IFFT[\hat{I}_2(f-h)] = \frac{1}{2} I_2(\lambda) \exp(i2\pi\lambda h) \qquad (7.9)$$

The logarithm of this signal is:

$$\log_e[\frac{1}{2} I_2(\lambda) \exp(i2\pi\lambda h)] = \log_e[\frac{1}{2} I_2(\lambda)] + i2\pi\lambda h \qquad (7.10)$$

The imaginary part of Eq.(7.10) is precisely the phase distribution of a scanned point within the chosen wavelength segment.

In OSSI, the surface measurement is implemented by a frame grabber and a CCD. The CCD is used for detecting a series of interferograms of different wavelengths within the chosen wavelength segment (Yamaguchi et al, 2000; North-Morris et al, 2002). A frame grabber is then implemented to transfer the interferograms from the CCD to a personal computer. Each grabbed frame forms a grayscale image in which the intensity of each pixel corresponds to the intensity of a scanned point at different wavelength, as stated in Eq.(7.2). When issuing data processing, all grabbed frames are packed in sequence and form a 3D data set as shown in Figure 7.6.

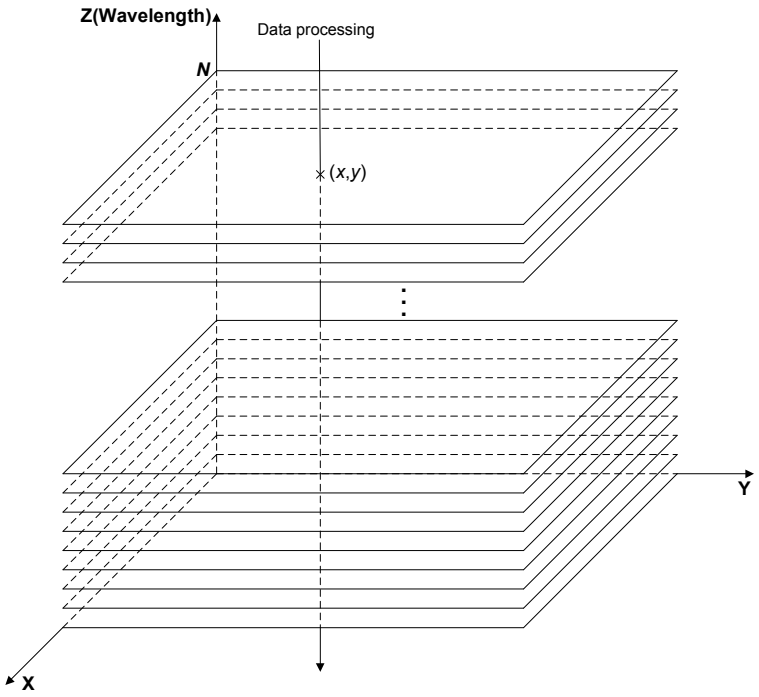


Figure 7.6 Pack of grayscale image at various wavelength

Intensities of the same pixel (x,y) in each grayscale image will be abstracted and formed into a vector $\vec{I}(x,y)=[I^1_{(x,y)},I^2_{(x,y)},...,I^N_{(x,y)}]$. For these discrete intensity signal, fast Fourier transform will be implemented to process them in frequency domain, which aims at filtering out $\hat{I}_2(f-h)$ in Eq.(7.8). After implementing inverse Fourier transform on $\hat{I}_2(f-h)$ and issuing logarithm computation that is illustrated by Eq.(7.10), the phase distribution of pixel (x,y)-- $\vec{\varphi}(x,y)$ will be acquired where $\vec{\varphi}(x,y)=[\varphi^1_{(x,y)},\varphi^2_{(x,y)},...,\varphi^N_{(x,y)}]$. However, the value of phase that is obtained by the existing function or library in any data processing tool is actually limited to within $[-\pi, +\pi]$ or $[0, 2\pi]$, which is shown by Figure 7.7 in which the number of wavelength used for scanning is 128.

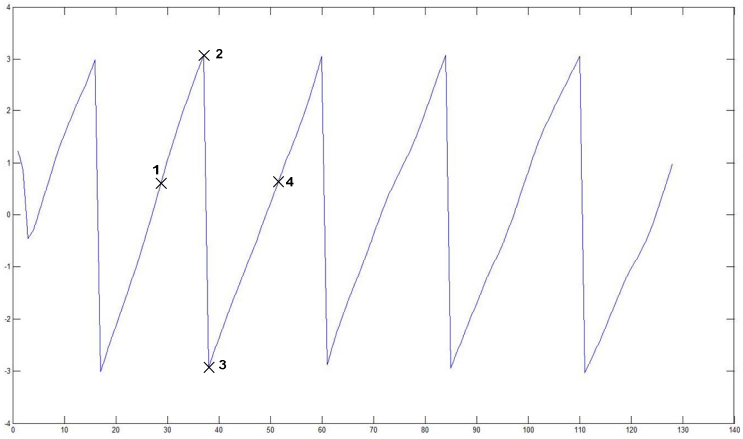


Figure 7.7 Phase distribution in the wavelength segment

Comparing the phase value of point 1, 2, 3, and 4, it is obvious that the phase shift between point 1 and 2 is within $[0, 2\pi]$. There is a 2π shift between point 2 and 3, so that the actual phase shifts between point 1 and 3, and between point 1 and 4 both exceed 2π. By comparing the phase shift of neighbouring points with 2π, the exact point where 2π phase shift occurs can be found out, the curve of 2π phase shift therefore can be obtained as shown in Figure 7.8, and the actual phase shift within the chosen wavelength segment can be also obtained as shown in Figure 7.9.

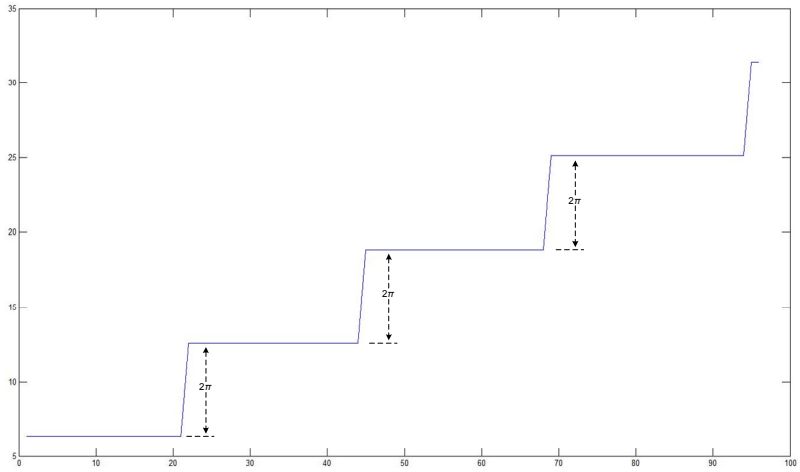


Figure 7.8 The curve of 2π phase shift

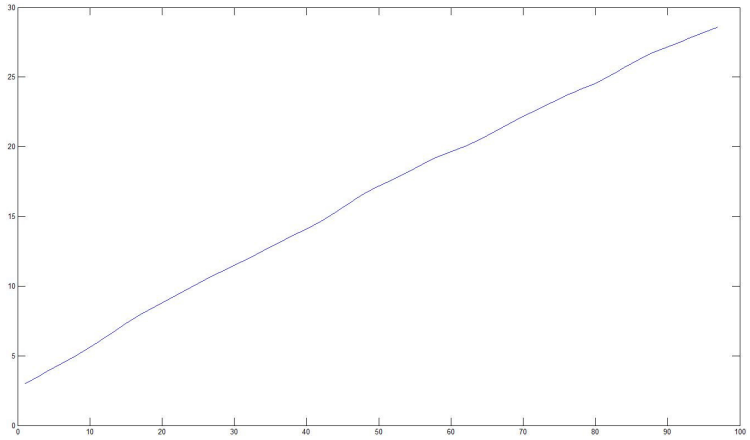


Figure 7.9 The curve of phase shift within chosen wavelength segment

Once the phase shift within the chosen wavelength segment is determined, the OPD of each scanned point can be acquired by converting Eq.(7.3) to $h = \Delta\Phi / 2\pi \cdot \Delta\lambda$.

It is clear based on the above illustration that each scanned point actually follows the same data processing pattern to obtain the topography of the measured surface. The multi-level parallel processing therefore can be realized by using the proposed programming frameworks in Chapter 4. In the following sections, the Compute Unified Device Architecture (CUDA), a new generation of parallel programming model, will be introduced. The earliest version of CUDA was released by Nvidia in November 2006. In this experiment, CUDA 2.1 (released in October 2008) was used for the test and evaluation.

7.3 Compute Unified Device Architecture (CUDA)

Briefly mentioned in Chapter 2, CUDA was designed from the ground-up for efficient general purpose computation on GPUs around 2008. In contrast to graphics-based GPGPUs, programmers can develop and compile GPGPU programs by using CUDA's C-like syntax and semantics, which is much simpler to deploy than the previous GPGPU platform characterized by the graphics APIs

and shading languages. Developing shader programs for GPGPU applications require skills and expertise in computer graphics, which had brought substantial difficulties to researchers in the past to harness the parallel processing power in PC-grade hardware. Empowered by the hard characteristics of the unified-pipeline-based graphics card, CUDA has exposed important features that are inherited and encapsulated from the conventional graphics APIs and shading languages. The most significant of those is the shared memory and the support for double precision floating point in arithmetic operations. In this section, new features of CUDA will be briefly introduced from the aspects of thread structure, memory hierarchy, host, and device. In CUDA, host commonly refers to CPU, while device refers to not only GPU, but also Cell or FPGA if CUDA is used by those processors.

Since the devised GPGPU programming framework follows the parallel pattern of the Processor Farms, a program written in CUDA is therefore comprised of two types of code, the host code implemented on CPU in serial, and device code implemented on GPU in parallel. The device code is further composed of a series of kernels that are instructions issued on GPU. The conception of thread is also introduced in CUDA's device code, for example, if a data stream involves *N* elements, then CUDA will establish a thread for the operation on each element, and *N* threads will run concurrently when a kernel is called.

7.3.1 Thread Hierarchy

CUDA programs use **__global__ or __device__** to label a kernel that is consisted of device code which runs on GPU. The computational range of GPGPU, i.e., the range of threads in a kernel, is defined by the syntax of **<<<…>>>,** which is different from the conventional shading language using vertex coordinate to specify a kernel's computational range. Each thread is automatically configured with a unique identifier that is labelled by a built-in variable in CUDA- **ThreadIdx.** Multiple threads make use of shared memory that is eauipped on each processing core and acts as L1 cache to achieve fast data

exchange. However, the capacity of shared memory is limited, which means not all threads in a kernel can access a single core's shared memory, so that the thread must be organized efficiently to balance the workload of GPU's core and its corresponding shared memory. This requirement results in the hierarchical structure in CUDA's thread management, which is characterized by thread grid and thread block, as depicted by Figure 7.10. All threads within a thread block can make use of the same shared memory for data exchange, therefore, the number of threads in a block is determined by the capacity of the shared memory, for example, a thread block on NVIDIA Tesla architecture can contain up to 512 threads (Nvidia Corporation, 2009).

The **<<<…>>>** syntax is normally initialized as **<<<**gridSize**,** blockSize**>>>** in which two parameters, gridSize and blockSize, are both 3-components vector which is denoted by *dim3* data type in CUDA. Parameter gridSize indicates the total number of thread blocks along x, y and z directions respectively. Each block is also automatically configured with a unique identifier that is labelled by a built-in variable in CUDA- **BlockIdx,** which is similar to **ThreadIdx,** while parameter blockSize indicates the number of threads along x, y and z directions in a block. Each thread is indexed by **ThreadIdx** that has been explained above.

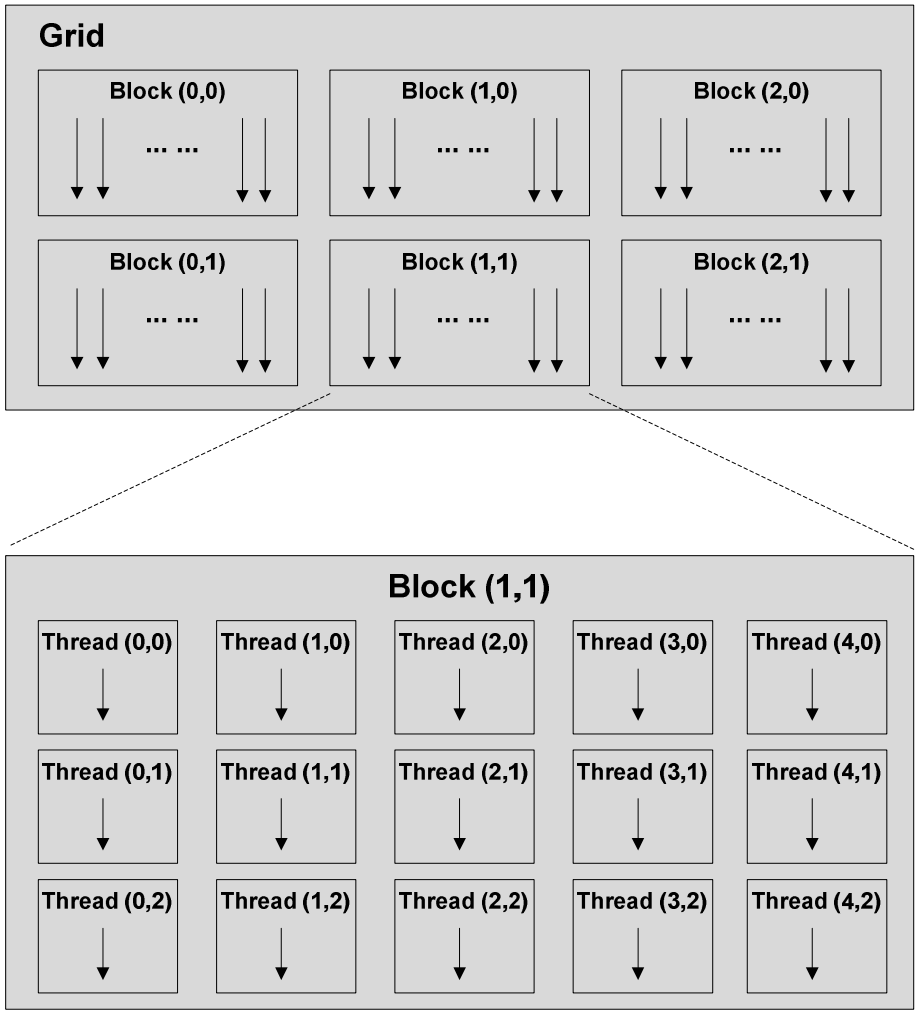


Figure 7.10 Grid of thread blocks

As an illustration, the following sample demonstrates how to do matrix addition using grid and thread blocks. In this example, $M \times M$ threads are encapsulated in multiple blocks, and each block consists of 32 threads along x and y directions.

```
__global__ void Addition(float X[M][M], float Y[M][M], float Z[M][M])
{
        int u =blockIdx.x * blockDim.x + threadIdx.x;
        int v =blockIdx.y * blockDim.y + threadIdx.y;
        if (u< M && v < M)
                Z[u] [v] = X[u] [v] + Y[u] [v];
}
```

```
int main()
{

    Dim3  blockSize(32,32)      // block size and the area of threadIdx is
                                   predefined

  //Specify the grid size, thus pre-define the area of  blockIdx
    dim3 gridSize((M + blockSize.x – 1) / blockSize.x,
                  (M + blockSize.y – 1) / blockSize.y);

  // Kernel invocation
Addition<<<gridSize, blockSize>>> (X, Y, Z);
}
```

From the above code snippet, it can be seen that the dimension of grid and block are specified by the first and second parameter of the <<<...>>>syntax respectively. In graphics-based GPGPU, the index of a data in data stream is specified by coordinate of the texture that stores data stream. While in CUDA, the index of data corresponds to the thread index in the structure of thread hierarchy that is featured by the thread block and grid.

7.3.2 Memory Hierarchy

CUDA names the GPU memory as the device memory, and it can be up to 1.5Gb in Nvidia GeForce GTX280 (Nvidia Corporation, 2009). A device memory is classified into various memory spaces that have different characteristics and performance. These memory spaces include global memory, constant memory, texture memory, local memory, registers and shared memory which are all illustrated in Figure 7.11.

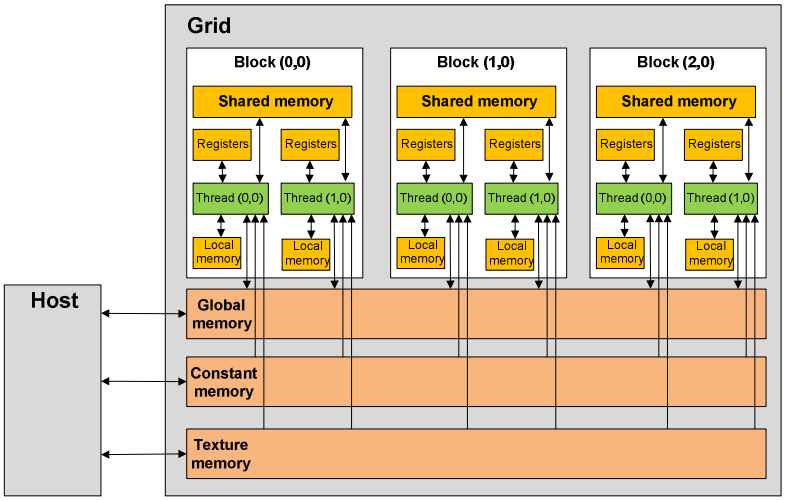


Figure 7.11 The memory spaces in device memory and their relationships between threads (Courtesy to Che et al.)

It can be seen from Figure 7.11 that the memory type defined in CUDA also present the hierarchical structure that precisely corresponds to the thread hierarchy. Each thread has a private local memory, while for thread block, its private memory is shared memory. As the basic unit in CUDA coding, thread is able to access all kinds of memory spaces. CUDA is also compatible with texture memory which is convenient for the graphics applications.

The memory spaces for CUDA application have different levels of accessibility. Generally, both registers and local memory can be read or written by a single thread, the distinction between them is that registers obtain a fast access speed while the access speed of local memory is relatively slow. Besides registers and local memory, shared memory and global memory can also be read and written, while constant and texture memory can only be read with a slow access speed, which is similar to the “gather” operation in the previous graphics-based GPGPU development tools. The diversity of memory space and the ability of “scatter” to memory are two advantages of CUDA comparing with the traditional graphics-based GPGPU development tools.

7.3.3 Host and Device

A GPGPU program written in CUDA includes *host* code and *device* code where *host* is specialised as CPU and *device* is referred to GPU. CUDA is a C-like language, either the *host* or *device* code is similar to the common C-language functions except the device code must use the label global_or device_for classification. Host code is implemented in serial, while device code run on GPU in parallel. A CUDA program comprises with a series of host-based functions and device-based kernels, as illustrated by Figure 7.12. CUDA's host code also includes API instructions that are similar to instructions of OpenGL or DirectX to invoke the device-based kernels, manage the hierarchical device memory, and transfer data between host memory and device memory.

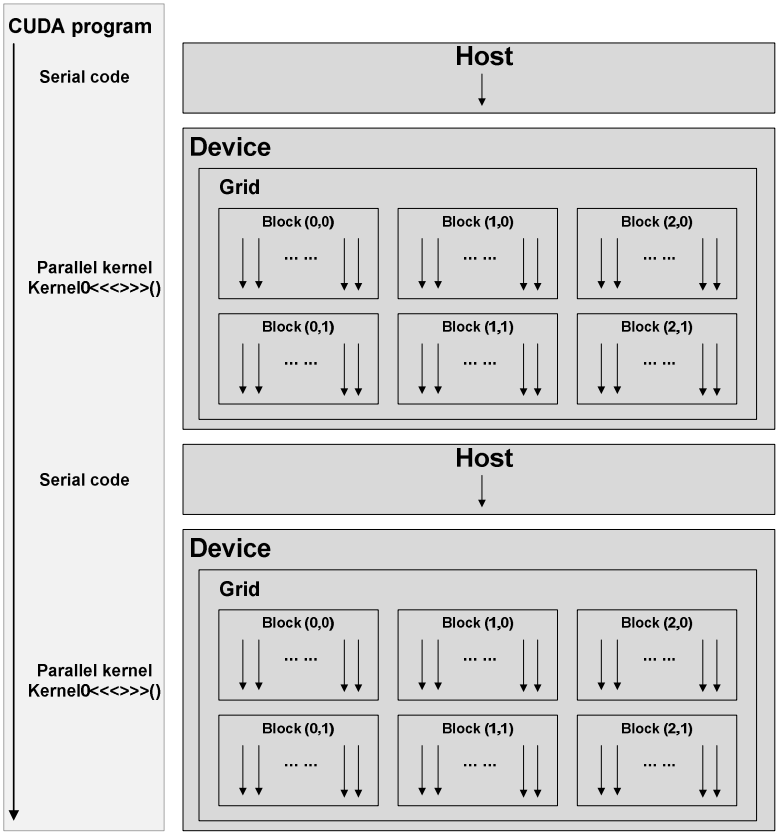


Figure 7.12 Heterogeneous programming in CUDA applications (Courtesy to Nvidia Corporation)

7.3.4 The programming API -- CUFFT

The Fast Fourier Transform (FFT) is intrinsically an iterative process which provides efficient solutions for computing discrete Fourier transforms that involve both complex or real-value data sets, and it is one of the most important and widely used numerical algorithms, with applications that include computational physics and general signal processing. Based on the product release announcements made by the Nvidia, CUDA provides the CUFFT library as a convenient interface for program developers. By employing CUFFT library, a developer need not to care the algorithm details issued in each step in FFT's iteration.

The CUFFT library in CUDA supports one-, two-, and three-dimensional transforms of complex and real-valued data. Multiple 1D FFT can be implemented in parallel by CUFFT library through batch execution, while for 2D and 3D transforms, the size of data array along each dimension can be up to 16384 (Nvidia Corporation, 2009).

In this project, batch execution for multiple 1D transforms will be issued by corresponding CUFFT functions as explained in Section 7.2. The main data types defined in CUFFT are listed in Table 7.1 (Nvidia Corporation, 2009).

Table 7.1 Data types in CUFFT

Type	Description
cufftHandle	A handle type used to store and access CUFFT plans. The user receives a handle after creating a CUFFT plan and uses this handle to execute the FFT plan.
cufftResult	An enumeration of values used exclusively as API function return values (The possible return values can be referred in the CUFFT documentation).
cufftReal	A single-precision, floating-point real data type.

cufftComplex	A single-precision, floating-point complex data type that consists of interleaved real and imaginary components.
cufftType	An enumeration of the types of transform data supported by CUFFT.

The relevant CUFFT functions for this experiment are listed in Table 7.2 (Nvidia Corporation, 2009).

Table 7.2 API functions in CUFFT

Functions	Description
cufftPlan1d()	Creates a 1D FFT plan configuration for a specified signal size and data type.
cufftPlan2d()	Creates a 1D FFT plan configuration for a specified signal size and data type.
cufftPlan3d()	Creates a 3D FFT plan configuration for a specified signal size and data type.
cufftExecC2C()	Executes a CUFFT complex-to-complex transform plan.
cufftExecR2C()	Executes a CUFFT real-to-complex (implicitly forward) transform plan.
cufftExecC2R()	Executes a CUFFT complex-to-real (implicitly inverse) transform plan.

The CUFFT library supports complex- and real-data transforms. The cufftType data type listed in Table 7.1 has the following values:

```
typedef enum cufftType_t {
        CUFFT_R2C = 0x2a, // Real to complex (interleaved)
        CUFFT_C2R = 0x2c, // Complex (interleaved) to real
        CUFFT_C2C = 0x29 // Complex to complex, interleaved
    } cufftType;
```

The detailed configuration of *cufftPlan1d*() is expressed as:

cufftResult cufftPlan1d(cufftHandle *plan, int nx, cufftType type,int batch);

Among all the input parameters, ***plan*** is a pointer to a cufftHandle object, ***nx*** is the transform size (e.g., 256 for a 256-point FFT), ***type*** is the transform data type (e.g., CUFFT_C2C for complex to complex), and ***batch*** is number of transforms of size ***nx***. The output of *cufftPlan1d*() is also a pointer ***plan*** that contains a CUFFT 1D plan handle value. The return value of *cufftPlan1d*() is a ***cufftResult*** data type which indicates whether the cufftHandle was allocated successfully.

cufftPlan2d() and *cufftPlan3d*() have analogous configuration with *cufftPlan1d*(), which are expressed as:

cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type);
cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz, cufftType type);

cufftPlan2d() and *cufftPlan3d*() create 2D and 3D FFT plan configurations respectively, according to specified signal sizes and data type. *cufftPlan2d*() is the same as *cufftPlan1d*() except that it takes a second size parameter, ***ny***, and does not support batching, and *cufftPlan3d*() takes the third size parameter ***nz*** except ***ny***. Both the output of *cufftPlan2d*() and *cufftPlan3d*() are in the parameter ***plan*** that contains a CUFFT 2D or 3D plan handle value, and the return value of these two functions are also ***cufftResult*** data type that has the same indication with that of *cufftPlan1d*().

cufftExecC2C(),cufftExecR2C(), and cufftExecC2R() have analogous parameter configurations as following:

cufftResult cufftExecC2C(cufftHandle plan, cufftComplex *idata, cufftComplex *odata, int direction);
cufftResult cufftExecR2C(cufftHandle plan, cufftReal *idata, cufftComplex *odata);
cufftResult cufftExecC2R(cufftHandle plan, cufftComplex *idata, cufftReal *odata);

The parameter ***plan*** is the cufftHandle object for the executing FFT plan, ***idata*** is the pointer to the input data (in GPU memory) to transform, and ***odata*** is the

pointer to the output data (in GPU memory) after transformation. Both ***idata*** and ***odata*** can be parameterized with cufftReal and cufftComplex data type. cufftExecC2C() can execute the forward and inverse FFT, which is determined by the fourth parameter *direction* in cufftExecC2C(). cufftExecR2C() executes an implicitly forward FFT while cufftExecC2R() executes an implicitly inverse FFT.

In this development, the function cufftPlan1d() was employed to create a 1D FFT plan and the function cufftExecC2C() was used to execute the forward and inverse FFT that calculates the pulses of the interference signals.

7.4 CUDA-based Data Processing in OSSI

Referring to the principle of data processing in OSSI that has been explained in Section 7.2, the developed CUDA-based programme for parallel processing includes the following sub-tasks:

- Loading original measured data (implemented on host);
- Issuing FFT and inverse FFT(implemented on device);
- Computing the absolute phase shift(implemented on device);
- Visualizing the results of data processing(implemented on device);

The details of these sub-tasks are demonstrated in the following sub-sections.

7.4.1 Initialization

The task of initialization mainly consists of loading measured data which is stored as a series of 8-bit grayscale bitmap images, allocating device memory on GPU, and transferring data from host memory to device memory. The number of images is determined by the number of wavelengths that was used for generating interference signals. A bitmap image of an interference signal generated at a specific wavelength is shown in Figure 7.13.

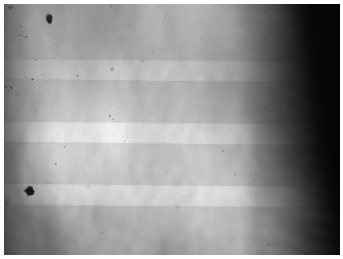


Figure 7.13 The intensity of interference signal at a specific wavelength

The example code of memory allocation is

cudaMalloc((*void***)*&d_yt, sizeof*(*Complex*) * *ImageSlices* * *ImageWidth* * *ImageHeight*);

which is very analogous to the *malloc*() function in C language. *d_yt* is a pointer to the allocated GPU memory segment which stores *float2* data type that is represented by *Complex* through the instruction of "*typedef float2 Complex*". Thus *(*d_yt*).*x* and *(*d_yt*).*y* are used to store the real and imaginary parts of the results of both the FFT and the inverse FFT. *ImageSlices* is the number of images, then *ImageSlices* * *ImageWidth* * *ImageHeight* is the total number of pixels in all the bitmap images. After the device memory allocation, the loaded data stored in array *h_yt* in host memory can be transferred to device memory through the following CUDA instruction:

cudaMemcpy(*d_yt,h_yt,sizeof*(*Complex*)**ImageSlices*ImageWidth*ImageHeight, *cudaMemcpyHostToDevice*);

7.4.2 FFT and Inverse FFT

The FFT is issued by the following code:

cufftHandle plan;
cufftPlan1d(*&plan, ImageSlices, CUFFT_C2C, ImageWidth * ImageHeight*);
cufftExecC2C(*plan,* (*cufftComplex **)*d_yt,* (*cufftComplex **)*d_yt, CUFFT_FORWARD*);

The total number of pixels in the FFT will be *ImageWidth* * *ImageHeight*. To implement these FFTs, all pixels in device memory are arranged as following order:

Figure 7.14 FFT on different pixels

The results of FFT were stored in *d_yt*. The next step is to implement filtering in parallel to get component $\hat{I}_2(f-h)$ as illustrated in Eq.(7.8) before the inverse FFT is applied, the inverse being involved by the kernel function *filtering_d_yt*(). The intrinsic operation of this kernel is to set some components in *d_yt* to 0, and just reserve the value of components that corresponts to the frequency segment of $\hat{I}_2(f-h)$. A snippet of this kernel is shown as follow:

```
static _global_ void filtering_d_yt(Complex *d_yt, int ImageWidth, int ImageHeight, int ImageSlices)
{
        const int x = blockIdx.x * blockDim.x + threadIdx.x;
        const int y = blockIdx.y * blockDim.y + threadIdx.y;

        //The following instructions are used for filtering out Î2(f−h) in Eq.(7.8)

        if ((x < ImageHeight) && (y < ImageWidth))  //Specifying computation area
        {
                for (int i=0; i<2; i++)
                {
                        d_yt[((x * ImageWidth)+y) * ImageSlices)+i].x=0;
                        d_yt[((x * ImageWidth)+y) * ImageSlices)+i].y=0;
                }

                for (int i=20; i<ImageSlices; i++)
                {
                        d_yt[((x * ImageWidth)+y) * ImageSlices)+i].x=0;
                        d_yt[((x * ImageWidth)+y) * ImageSlices)+i].y=0;
                }
        }
}
```

Based on kernel *filtering_d_yt*(), the parallel filtering of the results of the FFT is implemented by the CUDA API instruction that calls kernel *filtering_d_yt*() by

specifying the area of threads through the parameters *gridSize* and *blockSize* in the <<<...>>> syntax, which is expressed as:

filtering_d_yt <<<*gridSize, blockSize*>>> (*d_yt*, *ImageWidth*, *ImageHeight*, *ImageSlices*);

gridSize and *blockSize* are pre-defined as:

```
const dim3 blockSize(16, 16, 1);
const dim3 gridSize((ImageHeight + dimBlock.x – 1) / blockSize.x,
                    (ImageWidth + dimBlock.y – 1) / blockSize.y);
```

The inverse FFT is then issued by the *cufftExecC2C*() in CUFFT library as:

cufftExecC2C(*plan,* (*cufftComplex* *)*d_yt,* (*cufftComplex* *)*d_yt, CUFFT_INVERSE*);

7.4.3 Computing the Absolute Phase Shift

For each scanned point on the measured surface, the phases of interference signals at different wavelengths are acquired by issuing the logarithm on the results of the inverse FFT, and then abstracting the imaginary part of the logarithm computation. This procedure is issued by the kernel *Obtain_phase*() as follow:

```
static _global_ void Obtain_phase(Complex *d_yt, Complex *phase,int ImageWidth, int
                                  ImageHeight, int ImageSlices)
{
    const int x = blockIdx.x * blockDim.x + threadIdx.x;
    const int y = blockIdx.y * blockDim.y + threadIdx.y;

    if ((x < ImageHeight) && (y < ImageWidth))    //Specifying computation area
    {
        for (int i=0; i<ImageSlices; i++)
        {
            phase[((x * ImageWidth)+y) * ImageSlices)+i].x   //Computing the phase
          = atan2(d_yt[((x * ImageWidth)+y) * ImageSlices)+i].y, d_yt[((x *
                ImageWidth)+y) * ImageSlices)+i].x);
        }
    }
}
```

The CUDA API instruction calling the kernel *Obtain_phase*() is then expressed as *Obtain_phase <<<gridSize, blockSize>>> (d_yt, d_yt, ImageWidth, ImageHeight, ImageSlices)*, which guarantees this kernel is used by each thread in parallel on GPU. The phase shift between neighbouring wavelengths can be computed by the kernel *Phase_shift*() which is written as:

```
static _global_ void Phase_shift (Complex *phase, Complex *Diff_phase, int ImageWidth, int
                                  ImageHeight, int ImageSlices)
{
        const int x = blockIdx.x * blockDim.x + threadIdx.x;
        const int y = blockIdx.y * blockDim.y + threadIdx.y;

        if ((x < ImageHeight) && (y < ImageWidth))    //Specifying computation area
        {
                //Computing the phase shift between neighbouring wavelengths
                for (int i=0; i<ImageSlices-1; i++)
                {
                        Diff_phase[((x * ImageWidth)+y) * ImageSlices)+i].y
                    = phase[((x * ImageWidth)+y) * ImageSlices)+i+1].x
                    - phase[((x * ImageWidth)+y) * ImageSlices)+i].x;
                }
        }
}
```

The CUDA API instruction calling the kernel *Phase_shift*() is *Phase_shift <<<gridSize, blockSize>>> (d_yt, d_yt, ImageWidth, ImageHeight, ImageSlices)*. In the above snippet, *Diff_phase* is an array variable in device memory to store the phase shift. In fact, it can be seen that the x components of array variable *d_yt* are used for storing the phase while the y components are used for storing the phase shift.

The aim of computing phase shift between neighbouring wavelengths is to find out where 2π phase shift occurs, which is achieved by comparing the shift amplitude as illustrated by Figure 7.7 and 7.8. The corresponding kernel *2pi_PhaseLeap*() is then written as:

```
static _global_ void 2pi_PhaseLeap (Complex *Diff_phase, Complex * PhaseLeap, int
                                    ImageWidth, int ImageHeight, int ImageSlices)
```

```
{
        const int x = blockIdx.x * blockDim.x + threadIdx.x;
        const int y = blockIdx.y * blockDim.y + threadIdx.y;

        if ((x < ImageHeight) && (y < ImageWidth))    //Specifying computation area
        {

            PhaseLeap[(x*PhotoWidth+y)*PhotoSlices+0].y=0;  //Reset the initial value to 0

            for (int i=1; i<ImageSlices-1; i++)
            {
                if (Diff_phase [(x*PhotoWidth+y)*PhotoSlices+i].y>0)
                    Diff_phase [(x*PhotoWidth+y)*PhotoSlices+i].y=0;


                //The following programe is to acquire 2π phase leap
                Diff_phase [(x*PhotoWidth+y)*PhotoSlices+i].y
               =round(-Diff_phase [(x*PhotoWidth+y)*PhotoSlices+i].y/5)*2π;


                PhaseLeap[(x*PhotoWidth+y)*PhotoSlices+i].y
              =  PhaseLeap[(x*PhotoWidth+y)*PhotoSlices+i-1].y
              + Diff_phase [(x*PhotoWidth+y)*PhotoSlices+i].y;
            }
        }
}
```

The CUDA API instruction calling kernel *2pi_PhaseLeap*() is *2pi_PhaseLeap <<<gridSize, blockSize>>>* (*d_yt*, *d_yt*, *ImageWidth*, *ImageHeight*, *ImageSlices*). If the 2π phase shift along the whole wavelength segment used for the entire surface measurement is acquired, then the absolute phase shift within the wavelength segment, which is illustrated by Figure 7.9, can be obtained by the kernel *Abso_Phaseshift*() as demonstrated below:

```
static _global_ void Abso_Phaseshift (Complex *Phaseshift, Complex *PhaseLeap,int
                                      ImageWidth, int ImageHeight, int ImageSlices)
{
        const int x = blockIdx.x * blockDim.x + threadIdx.x;
        const int y = blockIdx.y * blockDim.y + threadIdx.y;

        if ((x < ImageHeight) && (y < ImageWidth))    //Specifying computation area
        {
```

```
            //Calculating the absolute phase shift through adding the 2π phase leap
            for (int i=0; i<ImageSlices; i++)
            {
                Phaseshift [((x * ImageWidth)+y) * ImageSlices)+i].x
              = Phaseshift [((x * ImageWidth)+y) * ImageSlices)+i].x
              + PhaseLeap[(x* ImageWidth +y) * ImageSlices +i-1].y;
            }
        }
}
```

After the absolute phase shift within the wavelength segment is calculated, the optical path difference (OPD) of each scanned point can be computed by the kernel *Obtain_OPD*(), thus enables the tomography of the structured surface being profiled according to the OPDs. The snippet of the *Obtain_OPD*() function is listed below:

```
static _global_ void Obtain_OPD (float *OPD, Complex * Phaseshift, float Δλ, int ImageWidth,
                                 int ImageHeight, int ImageSlices)
{
        const int x = blockIdx.x * blockDim.x + threadIdx.x;
        const int y = blockIdx.y * blockDim.y + threadIdx.y;

        //Calculating the OPD according to Eq.(7.3)
        if ((x < ImageHeight) && (y < ImageWidth))   //Specifying computation area
        {
            OPD [ (x * ImageWidth)+y] =
          = (Phaseshift [((x * ImageWidth)+y) * ImageSlices)+ ImageSlices-1].x
            - Phaseshift [((x * ImageWidth)+y) * ImageSlices)].x)/ Δλ;
        }
}
```

In the two functions above, the parameter *Phaseshift* corresponds to the array in device memory to store the absolute phase shift; $\Delta\lambda$ is a constant representing the length of wavelength segment; and the parameter *OPD* is the array to store the OPD of each scanned point. Since both kernels, *Abso_Phaseshift*() and *Obtain_OPD*(), are device instructions, there are corresponding CUDA API

instructions for calling them with specifications of threading areas, which are analogous with other aforementioned CUDA API instructions.

Based on the above discussions, the flows of CUDA-based parallel processing in OSSI can be summarized as in Figure 7.15.

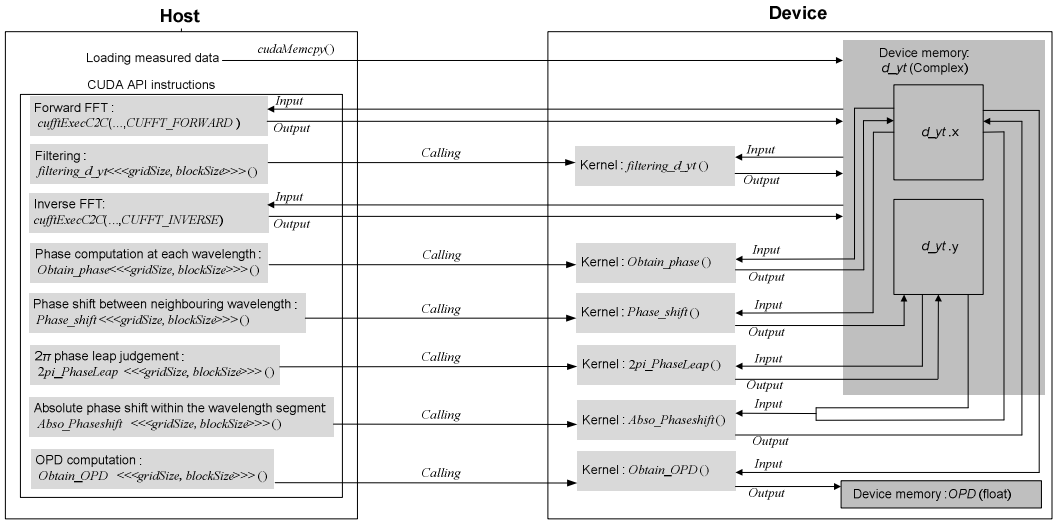


Figure 7.15 Flow of CUDA-based data processing in OSSI

7.4.4 Visualization of Processed Results

There remains a big challenge in massive data visualization, which is caused by the problem of real-time rendering as explained in the case study on Gaussian filtering in Chapter 5. The processing speed or “frame-rate” is commonly determined by both the processing speed of graphics card, which is evaluated by the maximum frames that the graphics card can render in one second, and the latency characteristic of communication between CPU and GPU.

Many solutions to alleviate this problem have been attempted in the past including graphics immediate mode, display list, vertex array, and the latest version -vertex buffer object (VBO) (Pharr et al., 2005). For the solutions of immediate mode, display list, and vertex array, data sets about vertices attributes

such as vertex 3D coordinates and vertex colours need to be transfered between CPU's memory and GPU's memory, which results in the direct performance correlation with the size of data set and the bandwidth of PCI bus. As an alternative, the VBO processes the vertex attributes (3D coordinates and colours) in physical space in "grouped" style, which is stored in various buffers in GPU's memory. It means the procedure of data transferring between CPU and GPU can be erased to promote the rendering efficiency.

Simply speaking, there are two types of buffer, array buffers and element buffers, defined in VBO. Array buffer, defined as the semantic **ARRAY_BUFFER**, contains vertex attributes, such as vertex coordinates, texture coordinate data, per vertex-color data, and normals. Element buffers, defined as the semantic **ELEMENT_ARRAY_BUFFER**, contains only indices of elements that are used to generate the correct shape of a geometric object.

In this experiment, VBOs are employed to visualize the processed data generated by the CUDA-accelerated OSSI system.

Two VBOs were created for geometric rendering in this experiment; one is used for specifying vertex coordinates and the other is for holding vertex colour. The creation and initialization of these two VBOs are illustrated by the following OpenGL code:

```
void createVBO(GLuint* vbo1,GLuint* vbo2)
{
  // create VBO to store vertex coordinates
  glGenBuffers(1, vbo1);
  glBindBuffer(GL_ARRAY_BUFFER, *vbo1);

  // initializing the size of the buffer data
  unsigned int size = ImageWidth * ImageHeight * 4 * sizeof(float);
  glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);

  glBindBuffer(GL_ARRAY_BUFFER, 0);

  // register VBO with CUDA
  cutilSafeCall(cudaGLRegisterBufferObject(*vbo1));

  // create VBO for vertex colour rendering
```

```
    glGenBuffers(1, vbo2);
    glBindBuffer(GL_ARRAY_BUFFER, *vbo2);

    // initializing the size of the buffer data
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);

   glBindBuffer(GL_ARRAY_BUFFER, 0);

    // register VBO with CUDA
    cutilSafeCall(cudaGLRegisterBufferObject(*vbo2));

}
```

Because both the vertex coordinates and the per-vertex colour are 4-component vectors, for example, (x,y,z,h) for vertex coordinates and (r,g,b,a) for the colour, the actual data size of these two VBOs need to be set as *ImageWidth* * *ImageHeight* * 4. Following the creation of VBOs, the index of each vertex must be specified to form the correct geometric shape. An element array buffer is created to store the vertex index when drawing the geometry. The main OpenGL instructions to initialize the element array buffer is listed as follow:

```
    glGenBuffersARB(1, indexbuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, * indexbuffer);
   glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER, size, 0, GL_STATIC_DRAW);
```

The parameter *indexbuffer* is a pointer of the *GLuint* data type, the parameter *size* is the buffer size that is determined by the drawing mode that is used by OpenGL to define primitives. The symbolic drawing mode includes GL_LINES, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, **and** GL_POLYGON etc. Different primitive definitions require different number of vertices.

The profile of the structured surface can now be formed by making use of the calculated optical path difference (OPD) that are stored in the device memory, i.e., in the OPD array as depicted in Figure 7.15. This procedure involves the drawing of the heightmap and the rendering of vertex colours according to the value of the OPD, which also requires resetting of the data value in the aforementioned VBOs (**vbo1** and **vbo2).** For accessing the two VBOs, two

pointers were set as the entry to the beginning of the buffer *vbo*1 and *vbo*2 by the following instructions.

```
float4 *dptr1;
cudaGLMapBufferObject((void**)&dptr1, vbo1);
float4 *dptr2;
cudaGLMapBufferObject((void**)&dptr2, vbo2);
```

The geometry will be then rendered by the kernel *rendering*() in the developed CUDA program. The code of this kernel is shown as below.

```
Static _global_ void rendering (float4* height, float4* colour, unsigned int ImageWidth, unsigned
                                int ImageHeight, float *OPD)
{
   const int x = blockIdx.x*blockDim.x + threadIdx.x;
   const int y = blockIdx.y*blockDim.y + threadIdx.y;

   float u = y / (float) (ImageWidth -1);
   float v = x / (float) (ImageHeight -1);

   u = u*2.0f - 1.0f;
   v = v*2.0f - 1.0f;

   // The height value of the vertex
   float w = OPD [x*width+y]/3000.0;

   // Output the vertex coordinate
   height [x* ImageWidth + y] = make_float4(u, w, v, 1.0f);

  // Output the vertex colour according to the height value
   colour [x* ImageWidth + y] = SetVertexColor(w);

}
```

Since *SetVertexColor*() is called in the device code, it has the following expression in CUDA applications.

```
static __device__ __host__ inline float4 SetVertexColor(float w)
{
        ... ;
}
```

Where *__device__* and *__host__* indicate this function can be invoked by both the host and the device code.

The API function *rendering*() is used to set the vertex coordinate and vertex colour in *vbo*1 and *vbo*2, both the VBOs and the aforementioned element array buffer – *indexbuffer*, are integrated to visualize the measured surface in the call-back function *display*() as listed below.

```
void display()

  // Visualizing the measured surface  from the corresponding VBOs and element array buffer
  glBindBuffer(GL_ARRAY_BUFFER, vbo1);
  glVertexPointer(4, GL_FLOAT, 0, 0);
  glEnableClientState(GL_VERTEX_ARRAY);

  glBindBuffer(GL_ARRAY_BUFFER, vbo2);
  glColorPointer(4, GL_FLOAT, 0, 0);
  glEnableClientState(GL_COLOR_ARRAY);

  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  glDrawElements(GL_TRIANGLE_STRIP,                ((mesh_width*2)+2)*(mesh_height-1),
                 GL_UNSIGNED_INT, 0);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

  glutSwapBuffers();
  glutPostRedisplay();
```

7.5 Performance Evaluation

The OSSI system uses various wavelength segments to evaluate different structured surfaces. It means the number of wavelengths used for producing interference signals is different in each application. To evaluate the performance of the developed CUDA-based parallel solution, in this experiment the measured data sets obtained from a 616×458-pixel CCD camera (with the respective wavelength number of 64, 128, 300 and 400) were tested. The parallel processing programs have been tested on a Quad-Core Pentium 2.66GHz PC equipped with a Nvidia GeForce GTX 260 GPU. Evaluations have been carried out to compare the results with a MATLAB-based multithreaded implementation on the same PC to assess the acceleration factor. The data accuracy were also assessed through comparing the maximum differences between the CUDA

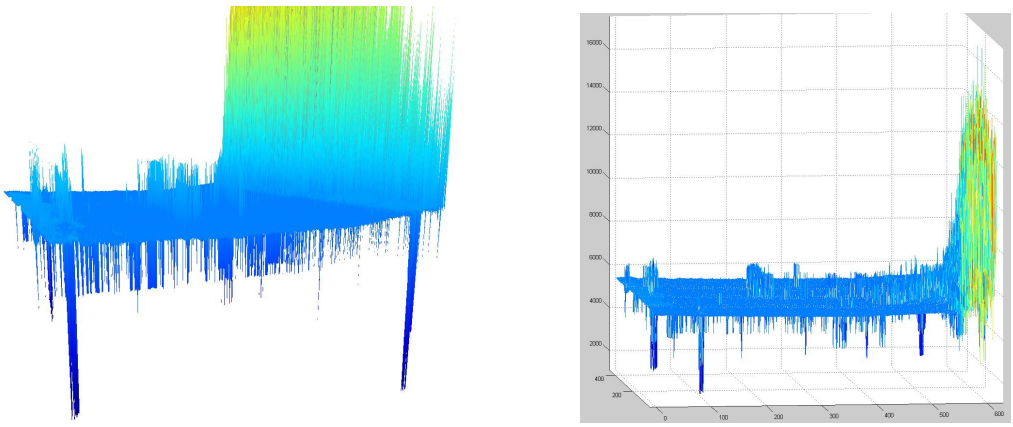
programs and the MATLAB simulations. Table 7.3 lists the average processing time and the approximate accelerating factors of the two approaches.

Table 7.3 Multi-thread and Multi-stream Performance Comparison

Spectral wavelength number	64	128	300	400
MATLAB processing time	15254.8ms	26842.4ms	58110.6ms	73297.3ms
CUDA-based processing time	611.4ms	1188.1ms	3136.2ms	4001.7ms
Accelerating factor	24.9	22.6	18.5	18.3

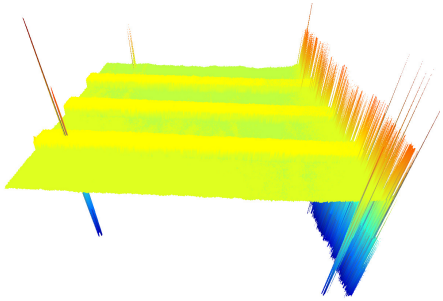
Based on the results shown in Table 7.3, it is evident that the GPU-based hardware accelerated approach has surpassed the performance from a serial computing solution by the factor of approximately 20.

Figure 7.16-19 show the surface profile obtained by the CUDA program and MATLAB simulations when wavelength number is 64, 128, 300 and 400.

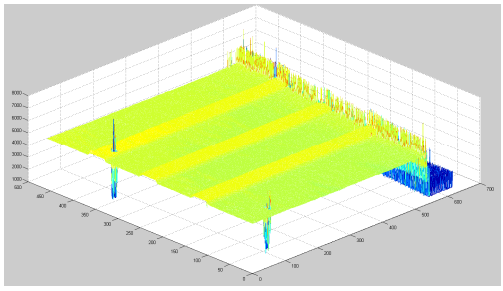


(a) Result of CUDA programme (b) Result of MATLAB programme

Figure 7.16 The surface profile (wavelength number=64)

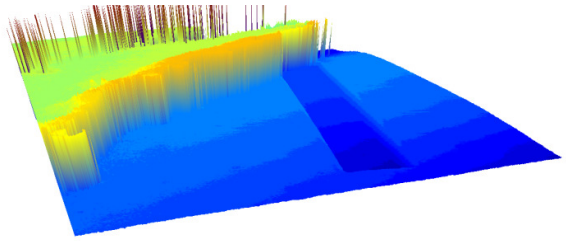


(a) Result of CUDA programme

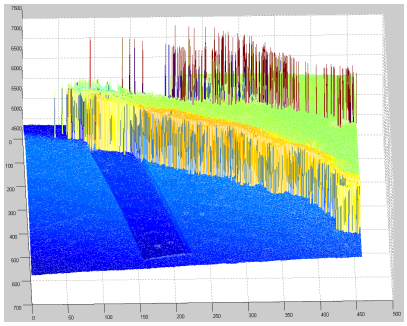


(b) Result of MATLAB programme

Figure 7.17 The surface profile (wavelength number=128)

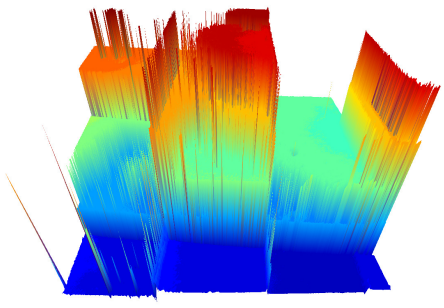


(a) Result of CUDA programme

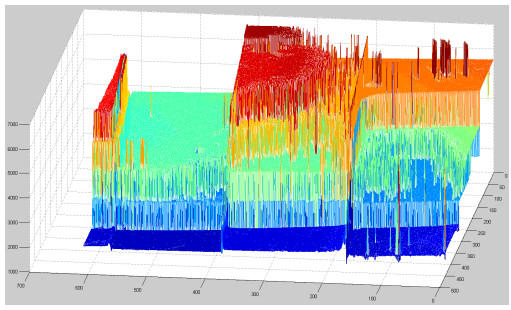


(b) Result of MATLAB programme

Figure 7.18 The surface profile (wavelength number=300)



(a) Result of CUDA programme



(b) Result of MATLAB programme

Figure 7.19 The surface profile (wavelength number=400)

Table 7.4 lists the maximum difference in absolute values between the CUDA solution and the MATLAB test when the wavelength numbers are 64,128,300, and 400 respectively. It is noted that the unit of difference is of nana-metre, thus the accuracy of CUDA programme satisfies the precision requirement of OSSI system for nano-scale surface measurement and analysis.

Table 7.4 The maximum difference in absolute value

Wavelength number	64	128	300	400
The maximum difference (nm)	0.004837	0.001074	0.007135	0.003385

It is also recorded that the total time consumed by the visualization of the processed data is around 3-quaterth of a second for all test cases. Therefore, it is concluded that the time for visualization is mainly determined by the size of processed data and not related to the length of wavelength segments. In the CUDA implementation, the data stored in device memory can be directly bound with a vertex buffer object to allow accesses to various array buffer types relating to vertex attributes such as coordinates, colours and normal values. In contrast to the time-consuming process of transferring data from texture memory or framebuffers back to the host CPU memory, the bottleneck of the GPU-to-CPU cross border operation has been eliminated with sunstantial improvements on the data visualization.

7.6 Summary

In this chapter, the latest GPU infrastructure and the CUDA have been explored and employed for parallel processing in an OSSI system. The aim of this case study is to release the potential of the unified programming architecture for real-world applications. In contrast to the legacy graphics-based GPGPU

programming framework, developers can focus on the real task in hand by using the C-like CUDA to avoid the tedious work of remapping their algorithms to graphics concepts. Except the original design of texture memory in all graphics-based GPGPU, the CUDA also equipped with the global memory, constant memory, and shared memory. This design has enabled CUDA programs to achieve the classical "gather" and "scatter" operations. Also attributing to the evolution of GPU's hardware architecture, especially since the release of NVIDIA's 8-Series, the distinctive vertex and fragment shaders have been substituted by functions dedicated to "*_device_*" or "*_global_*" for the unified shaders.

Through testing the CUDA-base GPGPU solution on four group of measured data, it is found the proposed solution can achieve an approximate speed up factor of 20 times as depicted by Table 7.3, while the data error of processed results obtained by the GPGPU and MATLAB programs is limited within the numeric level of 0.001 with the unit of nanometre, which completely satisfies the requirement on data accuracy of nanometre-level surface metrology. In addition, it is evident in this test series that through combining CUDA's device code with the VBOs, the geometry drawing operations have become much more efficient and resolved the bottleneck problem of transferring data from GPU to CPU.

Chapter 8 Experiment Analyses and Discussions

This chapter starts with the further analyses on the cases examined in Chapter 5, 6, and 7 in terms of GPGPU-based LTI systems. It proves onto the comparable effort in hardware acceleration for High Performance Computing (HPC) on consumer-level devices. Following the analyses, the specific conclusions in this research are then summarized in this chapter.

8.1 GPGPU-based LTI Systems Analysis

Based on the devised general GPGPU programming frameworks detailed in Chapter 4, three comprehensive case studies, focusing on filtering techniques, wavelet transforms, and Fast Fourier Transform (FFT), have been presented in Chapter 5, 6, and 7 to evaluate the performance of the corresponding GPGPU solutions. Although the case studies belong to the applications in surface metrology and image processing, the implementation of the solutions corresponds to a wide spectrum of specializations in LTI systems, which can be extended to other engineering applications. The following subsections summarize the technical criteria for GPGPU when applying to the LTI systems based on the devised programming models.

8.1.1 Time Domain Analysis on GPGPU-based LTI Systems

The characteristics of data parallelism determine that GPUs can be ideally employed by linear time-invariant (LTI) systems for process acceleration subject to a few adjustments.

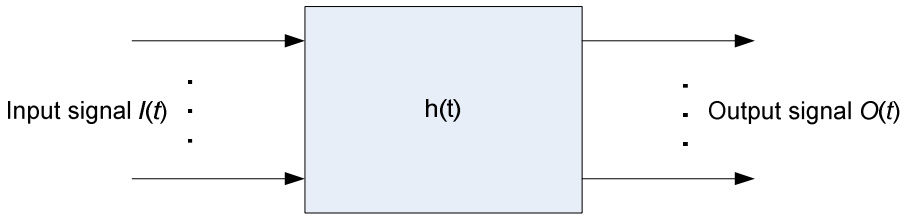


Figure 8.1 Diagram of linear time-invariant system

As shown in Figure 8.1, a LTI system is normally characterized by its impulse response function (indicated by $h(t)$ in Fig. 8.1) which takes in signal in the spatial domain and generates output $O(t)$ that is equivalent to the convolution between $I(t)$ and $h(t)$ depicted in Figure 8.1. Equally, a LTI system can also be analysed in frequency domain through transfer function which is the Fourier transform of its impulse response. The output in frequency domain is the multiplication of the transfer function and the input signal. The flowchart of a LTI system in the time and frequency domain is illustrated in Figure 8.2 (Willems, 1986).

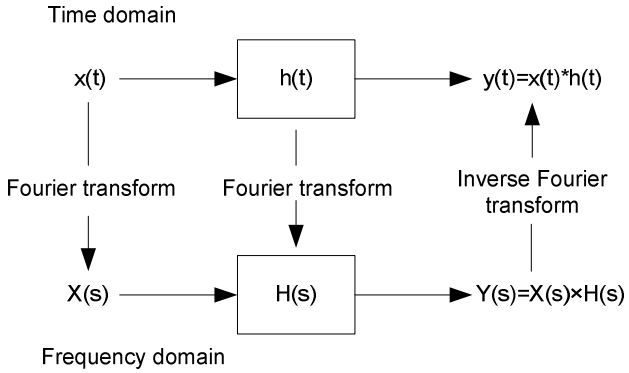


Figure 8.2 LTI system's flowchart in the time and frequency domain

The case study in Chapter 5 has examined in detail techniques and know-how in realizing GPGPU to implement filtering algorithms in the time domain. It is designed in the experiment that the continuous system in time domain will first be transformed into a discrete system, in which signals and impulse response must be discretized through sampling. As a result, the convolution is transformed from the integral operation in continuous domain to multiplication and addition operations in discrete domain. In the actual shader program development, this

corresponds to the multiplication and addition operations of matrix/vector that represent the input signal and the impulse response respectively. The multiple stream processors will carry out this basic linear algebraic computation in a parallel style. In conclusion of the case studied in Chapter 5, it can be summarized that the GPGPU-based time domain LTI can be issued by the steps shown in Figure 8.3.

As explained in Chapter 4, the fragment shader is commonly chosen as the "worker" in contrast to its predecessor, the vertex shader, for carrying out linear algebraic operations on a GPU. Figure 8.3 also highlights the GPGPU framework pattern as illustrated in Figure 4.3.

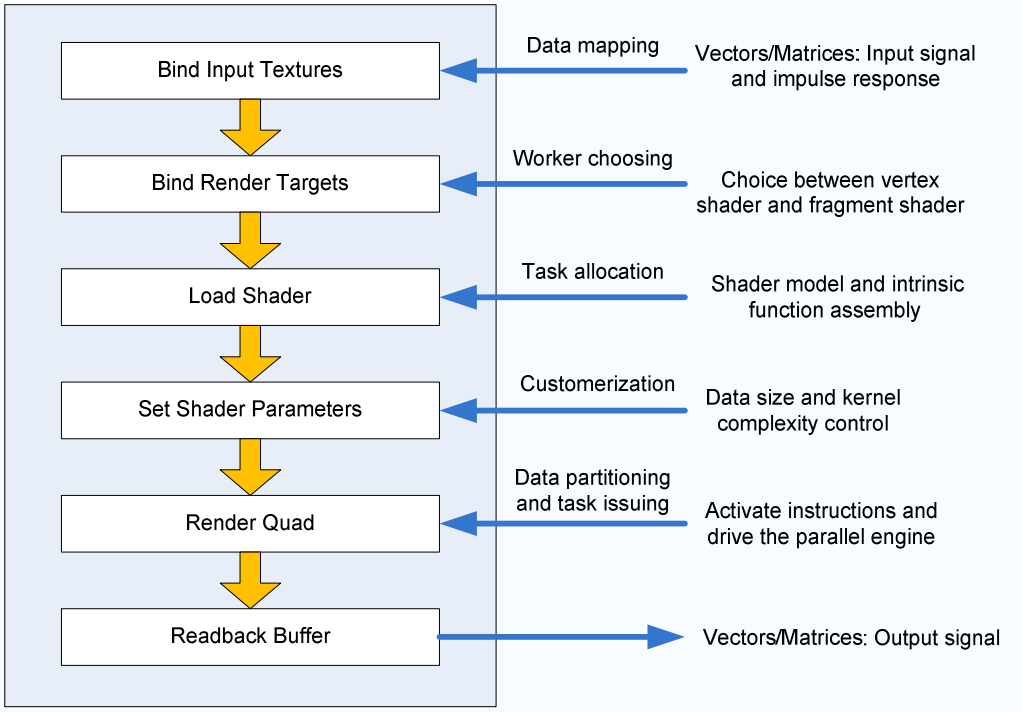


Figure 8.3 Flowchart of the GPGPU-based LTI systems

In addition to the filtering techniques used in LTI systems, other applications such as image processing have also extensively applied this type of processes. The case study reported in Chapter 6 has explored the GPGPU-based parallel implementation on wavelet-based image denoising. In comparison to Figure 8.1 which only shows the simplest LTI system, most applications involve multiple

impulse responses. For the practical applications of that nature, a thorough analysis on the wavelet transform and its realization on GPU is beneficial to developers since there is a cascading connection of vertical and horizontal filtering on the same decomposition level. As shown in Figure 8.4, for the whole wavelet transform process, there exists a uniform-style cascading connection of various levels of decomposition and reconstruction.

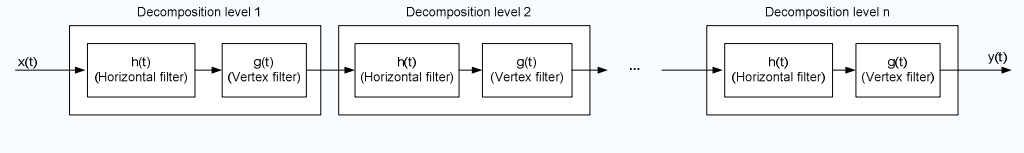


Figure 8.4 Cascading connection of LTI systems in wavelet transform

The structure of the cascading connection indicates a series of parallel cores have to be performed in sequence (serial processing) on a GPU. Figure 6.3 in Chapter 6 has illustrated the operational flowchart of the wavelet-based denoising on a GPU and Figure 7.12 depicts the responsibility of the host and device in a CUDA paradigm. The Framebuffer Object (FBO) has been chosen in the solution design to store the intermediate results of the cascading operation due to its flexibility and robustness with satisfactory result.

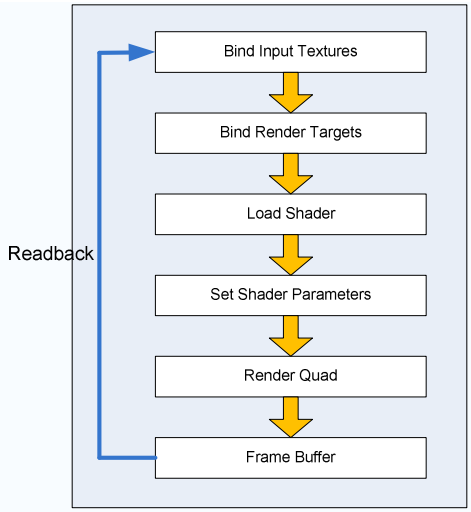


Figure 8.5 Flowchart of GPGPU-based signal processing on cascaded LTI systems

The loop structure shown in the Figure 8.5 indicates the multi-pass cascading relationship between various level of decomposition and reconstruction in the GPGPU application.

8.1.2 Frequency Domain Analysis on GPGPU-based LTI Systems

The case studies reported in Chapter 5 and 6 were focusing on the LTI system performance in the time domain. As highlighted in Figure 8.2, LTI systems can also be analysed in the frequency domain by applying Fourier transforms on the system impulse response. The experiment examined in Chapter 7 has demonstrated the realization of GPGPU-based LTI systems by the means of Fourier transform and the inverse Fourier transform. In this case, continuous-time system has to be transformed into discrete-time linear shift-invariant system. To further enhance the system performance, the discrete Fourier transform (DFT) has been adopted together with its inverse in implementing the practical system. It is well understood from practice that direct compute the DFT and IDFT can be extremely slow, therefore, the technique of Fast Fourier transform (FFT) is often employed as a practical solution. As such, the FFT is widely regarded as the foundation for analyzing LTI systems in the frequency domain. A wide spectrum of algorithms for implementing the FFT have been developed in the past, among them, the Cooley-Tukey algorithm, Prime-factor FFT algorithm, Bruun's FFT algorithm, Rader's FFT algorithm, and Bluestein's FFT algorithm are the most prominent and successful (Auslander et al., 1996; Temperton, 1983; Kekre et al., 1988; Swarztrauber et al., 1991). Echoing this development, the implementation of the FFT and IFFT on a GPU for speed gain has been proposed since the appearance of programmable GPUs, most of the pilot projects were carried out on the basis of the so-called butterfly operations.

The importance of the hardware-assisted FFT is also evident by GPU vendors' innovation on their software products. A classical example of this is the CUFFT, a

FFT library for CUDA covering the 1D power of 2 FFTs and the 3D non-power of 2 FFTs. “The CUFFT liberates the GPGPU programmer from tedious work of remapping their algorithms to graphics concepts”, as claimed by its developer. The case study in Chapter 7 has demonstrated the adoption of CUFFT functions for issuing FFT transforms on various data sizes.

8.2 Final Discussions

Based on the test results from the case studies in previous chapters and the further analysis of GPGPU’s processing flowchart on LTI systems as illustrated in Section 8.1, it can be concluded that GPGPU idealism and practices are effective and efficient for applications in which the system can be modelled as a LTI system. In addition, supported by the GPU’s operations on the floating-point level, GPGPU programs can achieve sufficient precision level as CPU-based programs in terms of data accuracy, which has been demonstrated by the case studies in the thesis. This has been proven valuable especially for the applications in surface metrology where data and processes are often having high demand on accuracy.

For nonlinear systems, GPGPU has also been applied in the area such as partial differential equation (PDE) solving and computational fluid dynamics (CFD) simulations, in which the nonlinear Navier-Stokes equation is commonly employed for modelling. It is observed that GPGPU can also achieve satisfactory acceleration for nonlinear systems if the operations can be linearized or be transformed into the algebraic operations expressed by matrices or vectors.

Therefore, as a cost-effective consumer-level stream processor, GPUs will play a much more important role in the future for real-world applications. As a guideline for the future effort on harnessing the power of GPUs, it is worth remembering that the acceleration performance of GPUs is largely determined by the following two pre-conditions:

- Independencies between data elements at each step in the computation for the reason that the parallel processing of today's GPUs is still pre-dominantly localized on the level of data parallelism;
- Uniformity of computations on data for the reason that GPU's operation is still only based on the SIMD mode.

For the algebraic operations on small-scale data sets, GPGPU solution does not bring obvious advantage in terms of computational efficiency due to the overhead of the operations between CPU and GPU. As a result, this thesis has only chosen the visualization of massive processed data, which is viewed as the classical setting, to demonstrate the effective ways in dealing with the issue of communications between CPU and GPU. To overcome this problem, one straightforward solution is to increase the bandwidth of the interface, for example, the bandwidth of the new generation of PCI-Express bus is up to 6Gb/s. However, under the condition subject to this research where the bandwidth of the data bus is fixed, the thesis has proven two effective methods to alleviate the negative impact of this overhead:

- To split a massive data into multiple smaller parts for "cross-border" transferring, which is originated from the strategy of the Divide-and-Conquer pattern;
- Utilizing GPU's memory and other hardware features for "pre-" and "post-" style processes to shifting the computational weight from CPU to GPU.

The effectiveness of the first method was validated by the case study in Chapter 5, while the latter one was validated by the using of vertex buffer object in the case study in Chapter 7. It is noted that the latter method is based on the functional enhancement of the new generation GPUs which allows the access to GPU's memory, that is, the support of direct memory address indexing.

Finally, through the domain survey in Chapter 2 and 3, it can be seen that the GPU's hardware structure and software development platform has evolved greatly since its first appearance over 10 years ago, and this trend is still

continuing. Considering the fact that different users are likely to be exposed to different specifications or even generations of GPU products, a general GPGPU programming framework is devised and presented in the thesis with the aim in shielding the detail distinctions of GPUs and focusing on the essential implementations in parallel processing. Based on the programming framework, a set of programming models for various applications have also been developed. The models focus on the kernel clarification through task partitioning which is viewed as the core of GPGPU practices and directly influences the computational efficiency of the devised solutions. The validity and feasibility of the research results were evaluated and proven through the cases studies in the surface metrology and image processing domains.

Chapter 9 Contributions and Future Works

The perceived contribution to domain knowledge from this research is summarized in this chapter, along with the anticipated future works in the related fields.

9.1 Contributions

The research reported in this dissertation has been focused on exploring and adopting commodity GPUs as parallel processors for accelerating scientific computation. Several discoveries and contributions have been made in the areas of GPGPU's parallel architectural patterns, overarching GPGPU programming framework, and GPGPU programming model design.

1. **Graphics Hardware Characterization**

   One of the contributions of this dissertation is the systematic and programmatic characterization of graphics hardware features, such as the vector processor architecture and various pipeline elements for mapping to the parallel processing paradigms, which has laid down a solid foundation for this research and future GPGPU applications.

2. **General GPGPU Programming Framework Definition**

   Following the detailed study and clarification of GPGPU's parallel architectural patterns, an overarching GPGPU programming framework was proposed aiming to formularize a common guideline to GPGPU programming model designs. The proposed framework supports the conventional GPUs equipped with traditional rendering pipelines, and the latest GPUs with the uniformed pipelines.

3. **Effective GPGPU Programming Model Design**

The dissertation has adopted a scope of popular engineering algorithms to devise the GPGPU programming model for implementing the algorithm in LTI systems. The adopted algorithms include filtering algorithms, Fast Fourier Transform (FFT) and wavelet transform, which are both generic and representative in practical engineering domains. In general, these algorithms represent the single-level and cascaded LTI systems in both the spatial and frequency domains. Based on the programming model, this research has revealed the detailed GPGPU solutions for the adopted algorithms in terms of kernel definition and development phases. The evaluation carried out in Chapter 5, 6 and 7 have proven that GPUs are capable of satisfactory performances on acceleration and precision for the adopted algorithms.

4. Measurable Criteria for GPGPU Performance Analysis

Another important contribution from this dissertation is the forming of criteria for quantifying and evaluating the performance of GPGPU solutions in term of speed up factor, and, operational and data accuracy. For testing the performance criteria, a set of CPU-based programs have also been developed in the research to compare with the performance of the proposed GPGPU solutions. While exercised, the evaluation results on the GPGPU solutions have clearly demonstrated advantages over their counterparts in a quantifiable fashion. Since the implementation of GPGPU is based on the “farmer-and-worker” architectural pattern with both GPU and CPU playing important roles, the effect of workload weighting on CPU in an entire GPGPU application cycle has also been thoroughly evaluated to validate the feasibility of the GPGPU programming framework in practical applications.

5. Efficient Visualization Techniques for Massive Data Sets

Another relative trivial but more “obvious” contribution is the near real-time visualization of the processed data. This dissertation presented a visualization solution for efficiently displaying massive data sets by splitting data through using the GPU resource of Vertex Buffer Object, which greatly promotes the

acceleration performance of the graphical operations such as the transformation computation and the visualization processes.

9.2 Future Works

Commodity-level parallel computing has a wide diversity of applications from embedded and mobile software through consumer applications such as games and multimedia to HPC solutions. This demand of more computational power and capacity has been driving a steadily increasing market for parallel computing products. Apart from GPUs, other intrinsically parallel processors such as FPGAs and Cell CPUs have also appeared on the consumer doorsteps, providing a spectrum of parallel computing options. To better harnessing the raw power of the less regulated consumer “gadgets”, it is essential to devise a unified programming model for devices such as GPUs, Cells, DSPs and other standalone or embedded processors in a system. CUDA has attempted to provide such a unified development platform, in the form of conceptions for host and computer devices that correspond to different kinds of processors. However, only limited success has been observed on Nvidia’s own GPUs. It is still not even compatible with other vendor’s GPUs.

A natural extension of this research would be the investigation of a heterogeneous framework consists of multi-core Cell CPUs, multiple GPUs and FPGAs that are interconnected by networks and databases to form clusters and grids. To support such a framework, the parallel task and distribution model will need to be developed and evaluated. The future research on the heterogeneous parallel programming framework can be centred on the following aspects.

- **Platform Models Definition**

The focus will be on the abstraction of an integrated parallel model (or models) for heterogeneous and asychronized hardware and networks. The investigation approach could follow the one adopted by the CUDA initiative, that is, a

hierarchical structure consists of one or more hosts plus one or more computing clients. Each computing client is composed of one or more computing units, while each unit can be further divided into processing elements corresponding to software elements (functions), and arithmetic as well as register units. The clients can communicate with each other by using device-portable or middleware functions with all communications monitored by the hosts.

- **Optimization and Standardization on Memory Access**

For efficient cooperation among the processing elements, shared memory can be used as a low-latency solution, much like the L1 cache in a processor core similar to the shared memory model adopted by CUDA. However, due to the likely limited shared memory resources from the heterogeneous framework, optimization on memory access needs to be further investigated. This part of the work should mainly be concentrating on the memory coalescing strategy in which memory transactions might be issued in an irregular mode.

- **Task Distribution Model**

As a branch of high performance computing, distributed computing is commonly implemented in the form of clusters and grids in which a Message Passing Interface (MPI) model is employed for task parallelism (Foster, 1995). GPU clusters were initially developed for graphics and rendering demanding tasks, such as the visualization of time-dependent Computational Fluid Dynamics (CFD) simulation, which can comprise several gigabytes of intermediate processed data in a single clock cycle and lasting through several hundred or thousand frames. It is anticipated that further research on the GPU or other device clusters need to be carried out to achieve load balance and optimized task parallelism. A classic application of such a model can be explained in the following example: the practical implementation of a parallel LTI system for video event detection in which a series of continuous video frames need to be processed in a timely fashion. The demanding process might even require frames from different time segments been processed by different algorithms. For automating and analysing

the videos online, the practical solution has to employ distributed processors with well balanced workload. There are currently limited researches on the GPU clusters based on the author's survey, possibly because the challenging demand in developing the complex parallel multigrid solvers with decoupled local smoothing mechanisms.

- **Syntax and Semantics for the Adaptable Program**

At the end of 2008, Apple, AMD and Nvidia have jointly released the Open Computing Language (OpenCL) as a future programming model and platform for developing programs that can execute across integrated parallel processing systems (IPPS). Similar to CUDA, OpenCL also employed the concepts of "host programs" and "kernels". However, OpenCL has added the flavour of task-parallelism to its kernel settings, for example, it envisages that a heterogeneous parallel system might be deployed through the task-parallel-based kernels for program execution. Although OpenCL is closing its Beta release, much researches are still needed in the application level.

References

[1] Adams, J. (2003) “Advanced animation with DirctX”, Boston, Massachusetts, USA.

[2] Almasi, G. S. and Gottlieb, A. (1990) ‘Review of "Highly parallel computing"’, IBM Systems Journal, 29(1) : 165 – 176.

[3] Arjona, J. L. O. (2006) “Architectural Patterns for Parallel Programming”, PhD Thesis, Department of Computer Science, University College London, 2006.

[4] ASME B46.1 (1995) “Surface Texture: Surface Roughness, Waviness, and Lay”, New York: American Society of Mechanical Engineers,1995.

[5] ATI Corporation (2009), “Product Archive”, Available online: <http://ati.amd.com/products>.

[6] Auslander, L., et al.(1996) “Multidimensional Cooley–Tukey Algorithms Revisited”, Advances in Applied Mathematics, 17 (4): 477-519.

[7] Azzalini, A., et al. (2005) “Nonlinear wavelet thresholding: A recursive method to determine the optimal denoising threshold”, Applied and Computational Harmonic Analysis, 18(2): 177-185.

[8] Barron, A., et al. (1999) “Risk bounds for model selection via penalization”, Probability Theory and Related Fields, 113(3): 301-413.

[9] Baskaran, M. M., et al. (2008) “A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs”, Proceedings of the 22nd annual international conference on Supercomputing, June 2008, Island of Kos, Aegean Sea, Greece. pp. 225-234.

[10] Berkovich, E. and Berkovich, S.(1998) “A combinatorial architecture for instruction-level parallelism, Microprocessors and Microsystems”, 22 (1) : 23-31.

[11] Berrington, N., et al. (1993) “Guaranteeing unpredictability”, The Computer Journal, 36 (8) :723-733.

[12] Birgé, L. and Massart, P. (1997) “From model selection to adaptive estimation, Festschrift for Lucien Le Cam”, Research Papers in Probability and Statistics, Springer(1997), pp. 55-88.

[13] Blasquez, I. and Poiraudeau, J. F.(2004) “Undo facilities for the extended z-buffer in NC machining simulation”, Computers in Industry, 53 (2) :193-204.

[14] Blunt L. and Jiang X. Q.(2003) “Advanced techniques for assessment surface topography: Development of a basis for 3D surface texture standards ‘SURFSTAND’”, Kogan Page Science, London.

[15] Bovik, A. (2005) “Handbook of image and video processing (Second Edition)”, Elsevier Academic Press, London, UK.

[16] Brinkmann, S., et al. (2000) “Development of a robust Gaussian regression filter for three-dimensional surface analysis”, In: Proceedings of the Xth International Colloquium on Surfaces.Chemnitz University of Technology, Chemnitz, 2000, pp. 122–132.

[17] Buck, I., et al.(2004) “Brook for GPUs: stream computing on graphics hardware”, ACM Transactions on Graphics, 23 (3) : 777-786.

[18] Buschmann, F., et al. (1996) “Pattern-Oriented Software Architecture”, John Wiley & Sons, Ltd., Oxford , UK.

[19] Cai, H., et al. (2004) “A performance anomaly in clustered on-line transaction processing systems”, Computer Communications, 27 (12) :1166-1173.

[20] Carriero, N. and Gelernter, D. (1988) “How to Write Parallel Programs. A Guide to the Perplexed”, Department of Computer Science, Yale University, New Heaven, Connecticut, USA.

[21] Chandy, K. M., and Taylor, S. (1992) “An Introduction to Parallel Programming”. Jones and Bartlett Publishers, Boston, Massachusetts, USA.

[22] Che, S., et al. (2008) “A performance study of general-purpose applications on graphics processors using CUDA”, Journal of Parallel and Distributed Computing, 68, (10): 1370-1380

[23] Chicken, E. and Cai, T. T. (2005) “Block thresholding for density estimation: local and global adaptivity”, Journal of Multivariate Analysis, 95 (1): 76-106.

[24] Christopher H., et al. (1994) “Laboratories for Parallel Computing”, Jones and Bartlett Publishers, London.

[25] Clark, J. H. (1982) “The geometry engine: A VLSI geometry system for graphics”, Proceeding of the SIGGRAPH’82. pp:127~133.

[26] Collange, S., et al. (2007) “Graphic processors to speed-up simulations for the design of high performance solar receptors”, Proceedings of the IEEE 18th International Conference Application-specific Systems, Architectures and Processors, May 2007, Volume 2, pp.377-382.

[27] Cormen, T. H., et al. (2001) “Introduction to Algorithms (2nd ed.)”, MIT Press and McGraw-Hill, USA, 2001.

[28] Culler, D., et al.(1997) “Parallel Computer Architecture”, Morgan Kaufmann Publishers, San Francisco, USA.

[29] Dai, X. and Katuo, S. (1998) “High-accuracy absolute distance measurement by means of wavelength scanning heterodyne interferometry”, Measure Science & Technology, 9 (5): 1031-1035.

[30] Dally, W. J., et al. (2003) “Merrimac: Supercomputing with streams”, In Proceedings of Super Computing (SC'03 Proceedings).

[31] Dally, W. J., et al. (2004) “Stream Processors: Programmability with Efficiency”, ACM Queue, 2 (1): 52-62.

[32] Darlington, J., et al. (1993) “Parallel programming using skeleton functions”, In Parallel Architecture and Languages Europe (PARLE’93), 1993.

[33] David J. (1994) “A multiprocessor architecture combining fine-grained and coarse-grained parallelism strategies”, Parallel Computing, 20 (5) : 729-751.

[34] DIN 4776 (1990) “Measurement of surface roughness for describing the material portion in the roughness profile”.

[35] Donald, K. (1998). “The Art of Computer Programming”, Addison-Wesley. pp. 158–168.

[36] Donoho, D. L. and Johnstone, I. M. (1995) “Adapting to unknown smoothness via wavelet shrinkage”, Journal of the American Statistical Association, Vol.90, pp.1200-1224.

[37] Donoho, D. L. and Johnstone, I. M. (1998) “Minimax estimation via wavelet shrinkage”, Journal of Applied Probability, 26 (3): 879-921.

[38] Donoho, D. L., et al. (1995) “Wavelet shrinkage: Asymptopia”, Journal of the Royal Statistical Society, Series B(Methodological), 57 (2) : 301-369.

[39] Douglas, C. (2009) “Computer Networks and Internets(5th Edition)”, Pearson Education, Inc..

[40] El-Rewini, H. and Abd-El-Barr, M.(2005) “Advanced Computer Architecture and Parallel Processing”, John Wiley & Sons, Inc., Oxford, UK.

[41] Enderle, G., et al. (1984) “Computer Graphics Programming: GKS --The Graphics Standard”, Berlin: Springer-Verlag.

[42] Flynn, M.(1966) “Very high-speed computing systems”, Proceedings of the IEEE,1966.

[43] Foster, I. (1995) “Designing and building parallel programs: concepts and tools for parallel software engineering”, Addison-Wesley, Reading, Massachusetts.

[44] Fuchs, H. and Poulton, J. (1981) “Pixel-Planes: A VLSI-oriented design for a raster graphics engine”, VLSI Design, 2 (3) : 20~28.

[45] Fuchs, H., et al. (1989) “Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories”, Proceeding of the SIGGRAPH’89, 1989, pp. 79~88.

[46] Geys, I., and Gool, L. V.(2007) “View synthesis by the parallel use of GPU and CPU”, Image and Vision Computing, 25 (7): 1154-1164.

[47] Goodeve, D. M. (1994) “Performance of Multiprocessor Communications Networks”, PhD Thesis, Department of Electronics, University of York, 1994.

[48] Goossens, B.(2001) “Handling 16 instructions per cycle in a superscalar processor”, Future Generation Computer Systems, 17 (6) : 699-709.

[49] Goswami, D., et al. (2002) “From design patterns to parallel architectural skeletons”, Journal of Parallel and Distributed Computing, 62 (4): 669 – 695.

[50] Gschwind, M. (2007) “The Cell Broadband Engine: Exploiting multiple levels of parallelism in a chip multiprocessor”, International Journal of Parallel Programming, 35 (3): 233-262.

[51] Hagen, T. R., et al. (2005) “Visual simulation of shallow-water waves”, Simulation Modelling Practice and Theory, 13 (8): 716-726.

[52] Han, C. Y., et al. (2005) “Geometry engine architecture with early backface culling hardware”, Computers & Graphics, 29 (3): 415-425.

[53] Hayes, J. (2002) “Dynamic interferometry handles vibration”, Laser Focus World, 38 (3): 109-118.

[54] Hector, F. C.(2002) “Signal de-noising in magnetic resonance spectroscopy using wavelet transforms”, Concepts in Magnetic Resonance, 14 (6): 388-401.

[55] Hill, F. S. (2001) “Computer graphics : using OpenGL”, London : Prentice Hall.

[56] Hirai, A., et al. (1999) “White-light interferometry using pseudo random-modulation for high-sensitivity and high-selectivity measurements”, Optics Communications, 162 (1-3): 11-15.

[57] Hlubina, P. (2002) “Dispersive white-light spectral interferometry to measure distances and displacements”, Optics Communications, 212 (1-3): 65-70.

[58] Hopf, M. and Ertl, T. (2000) “Hardware-Accelerated Wavelet Transformations”, Proc. EG/IEEE TVCG Symp. Visualization (SisSym ’00), May 2000, pp. 93-103.

[59] Hopgood, F. R. A., et al. (1983) “Introduction to the Graphics Kernel System (GKS)”, Academic Press.

[60] Howard, T. L. J., et al. (1991) “A Practical Introduction to PHIGS and PHIGS Plus”, Addition-Wesley, NJ, USA.

[61] Huang, J., et al. (1988) “Fringe scanning scatter plate interferometer using a polarized light”, Optics Communications, 68 (4): 235–238.

[62] Huang, T. C., et al. (2004) “The local memory access sequence of multiple induction variables on distributed memory machines”, Computers & Electrical Engineering, 30 (3) : 231-244.

[63] Intel Corporation (2007), “Intel® Hyper-Threading Technology”, Available online: < http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>

[64] Intel Corporation (2008), “Intel® Core™2 Quad Processors”, Available online: <http://www.intel.com/products/processor/core2quad/index.htm>

[65] ISO 3274 (1975) “Instruments for the measurement of surface roughness by the profile method-contact (stylus) instruments of consecutive profile

transformation – contact profile metres, system M", International Organization for Standardization, Geneva, Swiss,1975.

[66] ISO 11562 (1996) "Geometrical Product Specifications (GPS) -- Surface texture: Profile method -- Nominal characteristics of contact (stylus) instruments", International Organization for Standardization, Geneva, Swiss,1996.

[67] James, E., et al. (2004) "Instantaneous phase-shift, point-diffraction interferometer", Interferometry XII: Techniques and Analysis, edited by Katherine Creath, Joanna Schmit, Proceedings of SPIE Vol. 5531 (SPIE, Bellingham, WA, 2004), pp.264-272.

[68] Jiang, X., et al. (2006) "Near common-path optical fibre interferometer for potentially fast real-time micro/nano scale surface measurement", Optics Letters, 31(24) :3603-3605.

[69] Joo, K. and Kim, S. (2006) "Absolute distance measurement by dispersive interferometry using a femtosecond pulse laser", Optics Express, 14 (13): 5954-5960.

[70] Kaya, D. (2005) "The symmetric tridiagonal eigenproblem on a shared memory multiprocessor: Part II", Applied Mathematics and Computation, 163 (1) : 213-244.

[71] Kekre, H. B., et al. (1988) "Application of Rader transforms to the analysis of nuclear spectral data", Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 269 (1):279-281.

[72] Kessenich, J., et al. (2006) "The OpenGL Shading Language". Available online: <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf> [Accessed on 8th February 2007]

[73] Khronos (2009) "OpenCL Overview", Available online: <http://www.khronos.org/opencl/>

[74] Kincaid, D. and Cheney, E. W. (2002) "Numerical analysis : mathematics of scientific computing(3rd Edition)", Pacific Grove, Calif., United Kingdom.

[75] Kolks, J., et al.(2009) "Effects of video game console and snack type on snack consumption during play", Appetite, 52 (3): 841-843.

[76] Krüger, J. and Westermann, R. (2003) "Linear algebra operators for GPU implementation of numerical algorithms", ACM Transactions on Graphics (TOG) (Proceedings of ACM SIGGRAPH 2003), 22(3):908–916.

[77] Lefohn, A. E., et al. (2004) "A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces", IEEE Transactions on Visualization and Computer Graphics, 10 (4):422–433.

[78] Lefohn, A. E., et al. (2006) "Glift: An abstraction for generic, efficient GPU data structures", ACM Transactions on Graphics, 26 (1): 60–99.

[79] Li, W. (2004) "Accelerating Simulation and Visualization on Graphics Hardware." Ph.D. dissertation, Computer Science Department, Stony Brook University.

[80] Liu, S. G., et al. (2008) "Simulation of atmospheric binary mixtures based on two-fluid model", Graphical Models, 70 (6): 117-124.

[81] Losasso, F. and Hugues, H. (2004) "Geometry Clipmaps : Terrain Rendering Using Nested Regular Grids", ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004), 23(3):769–776.

[82] Luna, F. D. (2003) "Introduction to 3D game programming with DirectX 9.0", Wordware Pub.

[83] Manuel, V., et al. (1996) "Relating data-parallelism and (and-) parallelism in logic programs", Computer Languages, 22 (2-3): 143-163.

[84] Marziale, L., et al. (2007) “Massive threading: Using GPUs to increase the performance of digital forensics tools”, Digital Investigation, 4 (1): 73-81.

[85] Merlin, J., et al. (1999) “Multiple data parallelism with HPF and KeLP”, Future Generation Computer Systems, 15 (3): 393-405.

[86] Meunier, R. (1995) “The Pipes-and-Filters architecture”, in <Pattern languages of program design>, ACM Press/Addison-Wesley Publishing Co., New York.

[87] Microsoft (2006), “Microsoft Developper Network”, Available online: <http://msdn2.microsoft.com/en-us/library>

[88] Mitchell, J. L. and Sander, P. V. (2004) “Applications of Explicit Early-Z Culling”, In Real-Time Shading Course of SIGGRAPH 2004, 2004.

[89] Molnar, S., et al.(1992) “PixelFlow: High-Speed rendering using image composition”, Proceeding of the SIGGRAPH’92, 1992, pp. 231~240.

[90] Moncrieff, D., et al. (1996) “Heterogeneous computing machines and Amdahl's law”, Parallel Computing, 22 (3) : 407-413.

[91] Natalia, O. and Victor, O. (2006) “Computer networks : principles, technologies, and protocols for network design”, John Wiley & Sons Ltd, Oxford, UK.

[92] North-Morris, M. B., et al. (2002) “Phase-shifting birefringent scatterplate interferometer”, Applied Optics, 41(4): 668-677.

[93] Nvidia Corporation (2009), “Product and Technical Whitepaper Archive”. Available online: <http://www.nvidia.com>

[94] Oat, C. (2005) “Rendering to an off-screen buffer with WGL_ARB_pbuffer”, Technology paper of ATI Inc. pp.1-13. Available online: <http://ati.amd.com/developer/ATIpbuffer.pdf> [Accessed on 5th December 2006]

[95] Ocak, H. (2008) “Optimal classification of epileptic seizures in EEG using wavelet analysis and genetic algorithm”, Signal Processing, 88(7): 1858–1867.

[96] Owens J. D., et al. (2007) “A Survey of General-Purpose Computation on Graphics Hardware”, Computer Graphics Forum, 26 (1) : 80 -113.

[97] Pancake, C. M. (1996) “Is Parallelism for You?”, Computational Science and Engineering, 3 (2) : 18-37.

[98] Parberry, I. (2001) “Introduction to computer game programming with DirectX 8.0”, Wordware Pub.

[99] Persson, E. (2007) “Framebuffer Objects”, Technology paper of ATI Inc. pp.1-12. Available online: <http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/FramebufferObjects.pdf> [Accessed on 3rd September 2007]

[100] Pharr, M., et al. (2005) “GPU Gems 2:Programming Techniques for High-Performance Graphics and General-Purpose Computation”, Addison-Wesley, NJ, USA.

[101] Rabaey, J. M. (1996). “Digital Integrated Circuits: a design perspective”, Prentice Hall.

[102] Rabhi, F. A. (1995) ‘Exploiting parallelism in functional languages: A “paradigm-oriented” approach’, Abstract Machine Models for Highly Parallel Computers (pp.118-139), Oxford University Press.

[103] Raja J., et al. (2002) “Recent advances in separation of roughness, waviness and form”, Journal of the International Societies for Precision Engineering and Nanotechnology, 26(2):222-235.

[104] Rastello, F., et al.(2003) “Optimal task scheduling at run time to exploit intra-tile parallelism”, Parallel Computing, 29 (2) : 209-239.

[105] Reilly, S. P., et al.(2006) “Overview of MEMS sensors and the metrology

requirements for their manufacture Market", NPL report DEPC-EM 008.

[106] Robert L. G., et al. (1998) "Task-oriented asymmetric multiprocessing for interactive image-guided surgery", Parallel Computing, 24 (9-10) : 1323-1343.

[107] Sandoz, P., et al. (1996) "High-resolution profilometry by using phase calculation algorithms for spectroscopic analysis of white-light interferograms", Journal of Modern Optics, 43 (4): 701-708.

[108] Schneider, B. and Rossignac, J. (1995) "M-Buffer: A flexible MISD architecture for advanced graphics", Computers & Graphics, 19 (2) : 239-246.

[109] Schnell, U., et al.(1996) "Dispersive white-light interferometry for absolute distance measurement with dielectric multilayer systems on the target", Optics Letter, 21 (7): 528-530.

[110] Schwider, J. and Zhou, l.(1994) "Dispersive interferometric profilometer", Optics Letter, 19 (13): 995-997.

[111] Segal, M. and Peercy, M. (2006) "A performance-oriented data parallel virtual machine for GPUs". ACM SIGGRAPH 2006 Sketches.

[112] Seitz, C., "Evolution of GPUs", Available online:

<ftp://download.nvidia.com/developer/presentations/2004/Perfect_Kitchen_Art/English_Evolution_of_GPUs.pdf> [Accessed on 12th October 2006]

[113] Shaw, M.(1995) "Patterns for Software Architectures", In < Pattern Languages of Program Design>, ACM Press/Addison-Wesley Publishing Co., New York.

[114] Shirley, P.(2005) "Fundamentals of computer graphics (2nd Edition)", A K Peters, Wellesley, Massachusetts, USA.

[115] Silicon Graphics Inc. (1996) “The OpenGL Machine”, Available online: <http://www.opengl.org/documentation/specs/version1.1/state.pdf> [Accessed on 19th December 2006]

[116] Silicon Graphics Inc. (2005) “Silicon Graphics Prism Systems Breaking Barriers to Large Data Visualization for Researchers and Film Industry at SIGGRAPH 2005” Available online: <http://www.sgi.com/company_info/newsroom/press_releases/2005/august/siggraph2005.html> [Accessed on 24th March 2008]

[117] Sina, B., et al (2003) “Analog VLSI design automation”, London : CRC Press.

[118] Singleton, L., et al. (2002) “Report on the analysis of the MEMSTAND survey on Standardisation of MicroSystems Technology”, MEMSTAND Project IST-2001-37682.

[119] Sinnen,O. (2007) “Task scheduling for parallel systems”, John Wiley & Sons, Inc., Oxford, UK.

[120] Simon F. et al. (2007) “High-performance direct gravitational N-body simulations on graphics processing units”, New Astronomy, 12 (8): 641-650.

[121] Steve, H. (1995) “Microprocessor architectures : RISC, CISC, and DSP(2nd Edition)”, Oxford Press.

[122] Steven, G., et al. (1997) “A superscalar architecture to exploit instruction level parallelism, Microprocessors and Microsystems”, 20 (7): 391-400.

[123] Stout K. J. and Blunt L.(2000) “Three-dimensional surface topography (2nd Edition)”, Penton Press, London.

[124] Strang, G. and Nguyen, T. (1996) “Wavelets and Filter Banks”, Wellesley-Cambridge Publisher, Cambridge, Massachusetts, USA.

[125] Su, D. C. and Shu, L. H. (1991) “Phase-shifting scatter plate interferometerusing a polarization technique”, Journal of Modern Optics, 38(5): 951–959.

[126] Swarztrauber, P. N., et al. (1991) “Bluestein's FFT for arbitrary N on the hypercube”, Parallel Computing, 17 (6-7): 607-617.

[127] Szirmay-Kalos, L., et al. (2008) “GPU-based techniques for global illumination effects”, Morgan & Claypool Publishers, San Rafael, California, USA.

[128] Takeda, M. and Yamamoto, H. (1994) “Fourier-transform speckle profilometry: three-dimentional shape measurements of diffuse objects with large height steps and/or spatially isolated surfaces”, Applied Optics, 33(34): 7829-7837.

[129] Temperton, C. (1983) “Note a note on prime factor FFT algorithms”, Journal of Computational Physics, 52 (1): 198-204.

[130] Tenllado, C., et al. (2008) “Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting”, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. 19(3): 299-310.

[131] Thomas, F., and Zsolt, N. (2007) “Distributed and parallel systems : from cluster to grid computing”, New York : Springer.

[132] Thomaszewski, B., et al.(2008) “Parallel techniques for physically based simulation on multi-core processor architectures”, Computers & Graphics, 32 (1) : 25-40.

[133] Tomov, S., et al. (2005) “Benchmarking and implementation of probability-based simulations on programmable graphics cards”. Computers & Graphics, 29 (1):71-80.

[134] Victor, J. D., et al. (2005) “Interaction of luminance and higher-order statistics in texture discrimination”, Vision Research, 45 (3) : 311-328.

[135] Wagner, A. S., et al. (1997) “Performance Models for the Processor Farm Paradigm”, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, 8 (5):475-489.

[136] Walsh, P.(2008) “Advanced 3D game programming with DirectX 10.0”, Wordware Pub.

[137] Watt, A. H. (1999) “3D computer graphics”, Addition-Wesley, NJ, USA.

[138] Willems, J. C. (1986) “From time series to linear system—Part I. Finite dimensional linear time invariant systems”, Automatica, 22 (5): 561-580.

[139] William, G. and Rajeev, T.(2007) “Thread-safety in an MPI implementation: Requirements and analysis”, Parallel Computing, 33 (9) : 595-604.

[140] Wong, T. T., et al. (2007) “Discrete Wavelet Transform on Consumer-Level Graphics Hardware”, IEEE Transaction on Multimedia, 9(3): 668-673.

[141] Xie, K., et al. (2008) “Real-time visualization of large volume datasets on standard PC hardware”, Computer Methods and Programs in Biomedicine, 90 (2): 117-123.

[142] Yamaguchi, I., et al. (2000) “Surface topography by wavelength scanning interferometry”, Optical Engineering, 39(1): 40-46.

[143] Yamamoto, A. and Yamaguchi, I. (2000) “Surface profilometry by wavelength scanning Fizeau interferometer”, Optics & Laser Technology, 32 (4) : 261-266.

[144] Yanagi, K. and Hara, S. (2003) “Technical committee for standardizing the software to characterize surface topographic data—in concert with the geometrical product specifications: surface texture in ISO”, J Jpn Soc Precision Eng, 69(8):1057–1060.

[145] Zhao, Y., et al. (2006) “Melting and flowing in multiphase environment”, Computers & Graphics, 30 (4): 519-528.

Appendix A: Hardware Acceleration Prospects for High Performance Computing

For the traditional meaning, high performance computing (HPC) has the specific requirement on the hardware platform, which brings expensive cost for the users. However, the situation has somewhat changed recently, in the last decade or so, attributing to 'consumer-level' computing devices such as game consoles, mobile devices, and PCs. Except GPU, there are some other types of PC-grade HPC systems which include multicore processors, chip multithreading, Cell processors, field-programmable gate arrays. The technological characteristics of these systems are summarized below:

• Multicore Processors

The earliest multicore processor can be originated to the release of dual-core processors which are now installed in PC, hence it was viewed as an early version of PC-grade HPC system. Furthermore, Quad-core processors can yield the processing capability that is same as eight processors if the mother board supports two physical CPU sockets. In this way, more cores began to be integrated into a processor density along with the advancement of chip manufacturing. It is obvious that this trend will provide new opportunities to consumer-level parallel computing and might even bring great impact on some popular engineering/mathematical algorithms, for example, fast but serial-based algorithms. However, the situation of integrating dense cores in a single processor also requires high-standard communication buses in terms of bandwidth to main memory, synchronization and clocks. It is even clear to relative novice in computing that one shouldn't expect a double, quadruple, or octuple of program execution speed simple because there are dual-core, quad-core, or octa-core CPU's employed. A basic reason is that the process must cope with the problems of communication latency and bandwidth allocation among cores. This challenge becomes much greater when multiple CPU sockets exist on a main board. It also results in the consequence that a processor with more

cores has to run at a decrease clock frequency. It unavoidably decreases the multicore CPU's performance when used in HPC as a hardware accelerator. It has been observed from the case study in Chapter 7 that a quad-core CPU has a limited effect on acceleration in contrast to running the HPC workloads on GPU. Although six, eight, and even twelve-core processors are hanging just above the horizon, based on the author's view, this is not an imminent solution for power-hunger parallel computing applications.

- **On-Chip Multithreading Processor**

Chip multithreading technology (CMT) means a processor maintains separate threads, managed in hardware multi-threading rather than software multi-threading. In Software multi-threading, several tasks are implemented within a process where different tasks are implemented by various software threads. This has been widely used in today's operating systems and applications, and is available as a programming paradigm in mainstream languages like C++ and Java. However, the software threads are mainly executed on a single processor in a serial fashion. In contrast, a CMT-enabled processor has the ability of executing many software threads simultaneously within its own cores, which greatly increases a processor's efficiency. The classical products on the market that have adopted the CMT technology are the Sun's UltraSPARC T1 and T2+ processors. A standard configuration of T2+ processors is 8 cores in which 8 threads running in each core, this configuration has the same processing capacity as 64 separate processors.

Although integrated better than the above multicore-based approach, CMT still has the rigid demand on "suitable" applications and algorithm mapping. Unlike the processors based on the Simultaneous Multithreading technology (SMT), which had focused on promoting process ability by efficiently sharing some key resources which include execution pipeline and fetch bandwidth; while a CMT processor still operated at the style that multiple threads share resources on chip level, further research has to be carried out on this kind of resource share to find out the ideal policies or mechanisms to enhance CMT's processing ability. It is reported that on the Sun UltraSPARC T2+ processor, a linear increases in speed

is observed as more cores were added, but beyond eight, there was little increase in performance.

- **Cell Broadband Engine**

The Cell CPU, formally referred as the Cell Broadband Engine (Cell BE) processor, is originated from the design of Playstation3 gaming console. The architecture of the Cell BE has been introduced in Figure 4.8 in Chapter 4. Although this architecture was initially used for gaming, it now has been extensively viewed as an efficient HPC system. The newly released product of Cell CPU series, PowerXCell 8i, has more powerful processing ability on floating-point number than its predecessor. However, based on the author's observation, the access of the raw power of the Cell CPU on play station is deliberately made difficult for application developers due to commercial considerations by the manufacturer and vendors.

Within the Cell, the general-purpose Power Processing Element (PPE) hosts the operation system, therefore PPE is the controller of the whole system. Multiple Synergistic Processing Elements (SPEs) operate as PPE's coprocessor and has separate processing power that can achieve more than 25 GFLOPS for single precision mathematics. The SPE's separate processing power means SPE has its own shared and local memory, internal buses, and the interface based on direct memory access (DMA) to the PPE and other parts of Cell. This design provides benefit on data locality but exposes some challenges for programmers, for example, how to allocate workload on PPE and SPE to achieve the most optimal computing performance, and how to evaluate the influence of different workload distribution plan on the compile and run-time. Although IBM, the vendor of Cell, and some other software developing corporations such as RapidMind have released the software development kit (SDK) to guarantee SPEs are more transparent to developers, accessing the SPEs is still tedious due to the DMA model and PPE "front end" to the SPEs which unavoidably increases the development cost of using Cell CPU.

In contrast to the chip multithreading and multicore processors, the Cell processor, if properly tuned for a type of computation, will greatly exceed the

performance of a single microprocessor. For harnessing its power, an open source programming platform, Open Computing Language (OpenCL), has been developed by the HPC research community, which is currently under trial.

- **Field Programmable Gate Array**

Field programmable gate array (FPGA) is a special-purpose vector processor which guarantees developers can “route” applications on hardware, but not “code” them in software. For specially aligned applications, FPGA can achieve performance that is close to that of a standalone application-specific integrated circuit (ASIC), the digital signal processor or special-purpose “board” based devices. However, just small number of specially aligned applications can directly be run on FPGA for the reason that FPGA has the limit on the program size. In addition, the bandwidth limitations and the synchronization restrictions also limit the FPGA’s extensive application in practical engineering domain. Therefore, FPGAs are commonly treated as part of the so-called specific-purpose-built HPC systems, which are mainly used for digital signal processing, bioinformatics computation, and image processing with small-scale data sets. In addition, restrictions on applicable programming models for FPGAs, have limited its spread in industry. Therefore, FPGAs are considered not well studied to general-purpose computing as the ones investigated in this research.

Overall, GPUs has proven a qualified candidate in carrying out many HPC tasks and providing much needed hardware acceleration on an affordable cost to many engineering applications. Their outstanding GFLOPS and the large amount of arithmetic cores have power consumption over a general-purpose CPU. Application programming libraries such as CUDA’s Fast Fourier Transform (FFT) library has benefited programmers by avoiding the prerequisite knowledge on the hardware-level differences between different GPUs and graphical operations. The efficiency of a GPU acting as a hardware accelerator has been validated through the case studies in this project. It is envisaged that GPUs will have bigger shares in future consumer-level HPC systems research and development.