國立交通大學

資訊科學與工程研究所

碩 士 論 文

一個用於以圖形處理器為基底之線上遊戲
平 台 的 程 式 碼 產 生 器

A Code Generator for GPU-Based MMOG Platform

研 究 生：林宜豐

指導教授：袁賢銘 教授

中 華 民 國 九 十 七 年 六 月

一個用於以圖形處理器爲基底之線上遊戲平台的程式碼產生器

研究生：林宜豐　　指導教授：袁賢銘

國立交通大學資訊科學與工程研究所

摘要

本論文將介紹一個用於以圖形處理器爲基底之線上遊戲平台的程式碼產生器。利用此程式碼產生器，遊戲設計者可建立和管理遊戲邏輯，而不需要了解圖形處理器程式設計的細節。爲了讓遊戲設計者更容易定做他們的遊戲邏輯，我們定義了一系列的 XML (eXtensible Markup Language)的文法，利用程式碼產生器將該 XML 轉換爲可在圖形處理器上執行的程式碼，並以插件的形式與平台共同處理整個遊戲的運作。此外，我們針對平台在不同情況下的效能表現做了各種測試。其結果顯示，所產生的程式碼可由該平台以平行化的方式有效率地處理從客戶端送來的指令。

A Code Generator for GPU-Based MMOG Platform

Student : Yi-Lin Lin Advisor : Shyan-Ming Yuan

Department of Computer Science

National Chiao Tung University

Abstract

This research presents a code generator for a Graphics Processor Unit (GPU) based Massive Multiplayer Online Game (MMOG) platform. Through the code generator, game designers can construct and maintain game logics without comprehending details of GPU programming. We defined a set of eXtensible Markup Language (XML) schema for game designers to customize their game logics in a manageable way. Furthermore, we provide a code generator to translate the XML to GPU code pieces as the server plug-ins to process the entire game. In addition, several tests are performed to validate the performance of the MMOG platform in different situations. As a result, commands issued by MMOG clients can be processed efficiently by GPU-based server through the generated GPU code in parallel.

Acknowledgement

首先我要感謝袁賢銘教授所給予的指導，自大四專題開始即以自由明朗的風氣帶領著我們，對於我們所做的研究，給予了最大的創意空間。另外也要感謝各位口試委員，在百忙中抽空前來，並對論文提出了許多建議，使論文更加地完整和嚴謹。

此外，我要感謝分散式實驗室的各位學長姐、同學和學弟妹，不管是生活上的點點滴滴，或是學術上的各種討論，都讓這兩年間更加地充實。其中，我要特別感謝學長葉秉哲、邱繼宏、林家鋒、高永威以及宋牧奇等，在研究過程中給了我不少建議。還有感謝實驗室同學簡士強、周鴻仁、林辰璞和謝明志，在這兩年中的互相激勵和幫助。

最後，我要感謝父母從小的教誨，給予我良好的學習環境，使我能夠專心於做學術上的研究，謹以這篇論文來感謝您們的養育之情。

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1. Preface

As the network technology evolved, MMOG (Massive Multiplayer Online Game) has become the one of the largest computer game industry. Each MMOG forms a virtual world, within which people act and live together. No matter where they are, the virtual world connects them in the same space, at the same time.

1.2. Motivation

MMOG is sweet and sour for most game company. The sweetness comes from the fact that no private copy is possible because everyone needs to connect to the virtual world, thus increase the revenue. On the other hand, the sourness is high cost and long development cycle. Because there are several critical aspects needed to address even for a simplest MMOG, such as networking which requires enterprise-level reliability, stability and scalability to supply tens of thousands player concurrently.

To ease the overall MMOG development, several middleware solutions exist both in commercial market and open source community. They all have the same goal to support the online game development and to shorten the development cycle by reducing the building complexity. Although there is no in-depth comparison between these solutions, but generally speaking, in terms of scalability and robustness, they are not good enough as what we expect. According to some statistics, all real-world online games fail to scale up to more than 200k players, so that the game operator

usually gives more shards (i.e. a separate virtual world) as the number of subscriber grows. Each shard is running on a server cluster consists of about 5 to 20 servers to form a virtual world with thousands of concurrent users. The number of concurrent players on each server is about hundreds to thousands, which is pretty small. Meanwhile, players in different shards cannot communicate or interact with each other because they are physically separated world and this makes shard design less-attractive.

As shard is less-attractive, there are some research works and middleware trying to break the shard boundary to make it a shard-less design. For example, Sun Microsystems initiate an ambitious project called Darkstar [1] to build a large and shard-less virtual world, but the project doesn't seems to actually solve the problem from their latest benchmark numbers, and also the cost for entire server cluster still remains high and unaffordable for small game companies.

In this thesis, we look into the scalability problem existed in most commercial and open source middleware solutions, and propose a solution to scale up to a very large number of concurrent players in single server while maintaining the ease of use in overall game development.

1.3. Problem Description

Designing a flexible, scalable, customizable, and also easy-to-use MMOG platform is challenging. Among all, the most critical factor of a MMOG platform is still the performance which refers to scalability. Scalability usually a key of cost for operations because the better scalability a platform has, the fewer clusters operator needs to deploy. In additional to scalability, flexibility and ease of use of a MMOG platform are also significant. Flexible and customizable designs would give more

possibility for game designer to construct more complicated game play, which can really make differences in competitive online game market.

To give better understanding of the problems, we first elaborate some design pattern and common constraints in MMOGs. Almost all MMOGs are based on command-update communication model in which every players uses a client application to send commands to the server and receive updates in return. For each command, server needs to process the request through a specific game logic and sends updates back in a limited timeframe. The timeframe depends on game genre, and, for example, most MMORPG requires 200ms response time to deliver a smooth game play, and MMOFPS usually requires less 70ms to make it playable. Many research works have been conducted to reduce the latency, but most of them attack the problem from the network perspective. They tried to reduce the network latency by proposing different network topologies. Proposed communication architecture includes peer-to-peer and scalable server/proxy architecture. They have succeeded to alleviate the network latency between server and client. However the number of concurrent player per server is far from 10,000 even if the network latency has been improved by various works.

In our point of view, as the network technology evolved, the network latency is getting lower and lower and has dropped to a certain level. The bottleneck becomes the client command processing on server-side. As the number of players grows, the number of client commands needed to be processed in the limited timeframe increases. However, the current design of CPU is not capable of processing such a huge amount of commands in the limited timeframe. Although recently multi-core CPUs are introduced, the cache-misses occurred for different execution context of game logics and network handlers, and the synchronization among threads or processes will even

lower the CPU processing capability of client commands when lock is needed to avoid update conflict to guarantee consistent results.

Because of the limit, we started to investigate the source the problem and found it the result of sequential nature of CPU processing. In our previous work, we give the answer for this problem: make it parallel by using GPU. For the last decade, GPU had evolved into a powerful array of general SIMD processer instead of just being a simple 3D accelerating silicon. To date, the computation power of GPU is hundreds of times more than that of CPU, and it's believed that the GPU computation power will continue to supersede Moore's Law, while CPU has hit a barrier.

Although light has been spotted on GPU, converting a sequential program to a parallel one is not straightforward. Algorithm needs to be re-designed and data structure needs to be re-organized to reflect the different access pattern in parallel programming. Several attempts have been made to exploit GPU computation power to general problem such as collision detection [2] and online database processing [3]. For MMOG, the world first GPU-assisted MMOG platform is proposed in [14]. However it is still not a "full-featured" platform in terms of flexibility and ease-of-use. Due to the fact that most programmers or game developers are not familiar with parallel programming or GPU programming, it's still not possible to build an online game server that is accelerated by GPU computation.

1.4. Research Objectives

In this thesis, we try to give a more modular GPU-assisted MMOG platform, and provide way for game designer to construct game logics processed on GPU without the knowledge of parallel programming or GPU programming. As we don't require game designer to write real GPU code, an interface that bridges game logic design

concept to real code is needed, which is basically a code generator. In addition, various optimization of code generator needs to consider further enhance the server performance , thus increase the scalability of the platform.

1.5. Research Contribution

We discusses several issues and problems to build a MMOG platform with flexibility, scalability and customizability while handling client commands and create updates on both CPU and GPU. Because of the architecture difference between CPU and GPU, platform itself has been re-designed to exploit the CPU/GPU computation power by generating compatible code on both CPU and GPU. We define a set Extensible Markup Language (XML) schema for game designer to customize their game logic and provide a code generator to translate the XML to GPU code pieces as the server plug-ins to process the entire game play. A number of regression tests are performed to validate the correctness and robustness of the code generator and demonstrate the efficiency of GPU-assisted MMOG platform. As a result, commands issued by MMOG clients can be processed efficiently by GPU-assisted server through the generated GPU code in parallel.

2. Background and Related Work

2.1. Graphics Processor Unit (GPU)

The GPU technology has changed a lot in a last few years, both in hardware and software.

The hardware structure has transformed from fixed function rendering pipeline to programmable pipeline consists of multiple SIMD processors. The computation power gap between CPU and GPU is getting larger and larger, as described in Fig. 2-1.

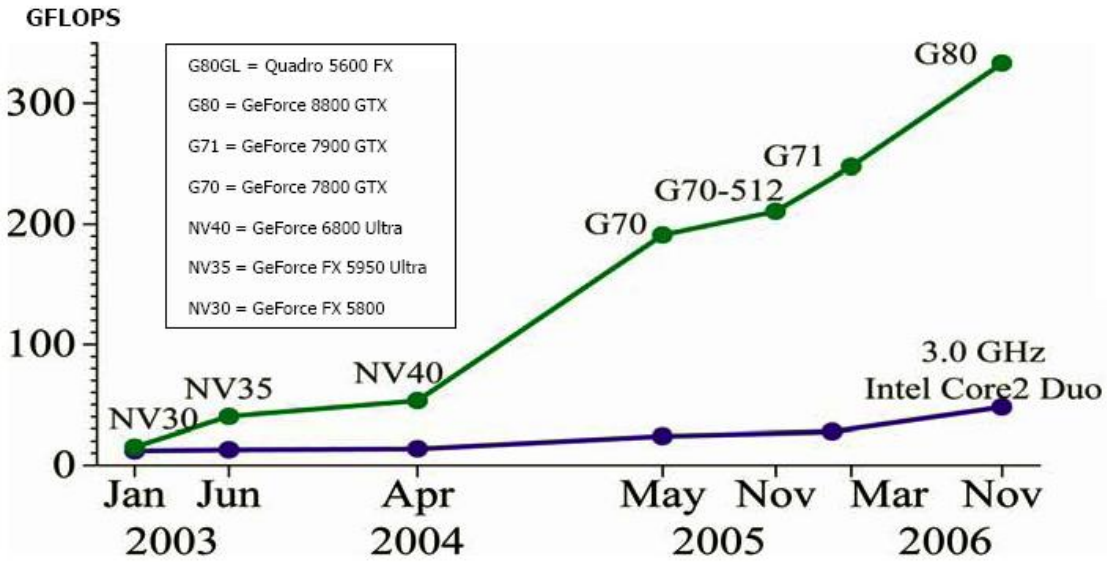

Fig. 2-1 Floating-Point Operations per Second for the CPU and GPU

For software, OpenGL persists but several extensions added by OpenGL ARB [4] for usage of the programmable pipeline. Base on programmable pipeline and the SIMD architecture of current GPU, there is a new kind of computing method called general computation on graphics hardware, also called GPGPU [5][6].

The concept of the computing method is first to map a compute-intensive problem into multiple small pieces, then solve the small pieces within pixel-rendering context (this is now programmable) and finally store the result into frame buffer object.

This method has extremely good performance for compute-intensive problem, like matrix operations [7][8]. However the implementation is hard for programmers not familiar to graphics processing because of the texture processing. For the general processing purpose on GPU, the two largest graphics manufacturers, NVIDIA and AMD/ATI, give different answer to developers toward GPGPU. NAVDIA proposed Compute Unified Device Architecture (CUDA) while AMD/ATI threw Close-To-Metal (CTM), which becomes AMD Stream technology later. Here we give a briefly introduction for these two technologies.

2.1.1. Compute Unified Device Architecture (CUDA)

For general purpose computing usage on GPU, NVIDIA released CUDA for developers to speed up there program on their G80/G92 based GPU. It's the architecture that unifies the general computation model on graphics devices. CUDA use C language with extensions from C++, such as template and variable declaration in C++ style. This makes a great convenience for programmers. Here we give more details for CUDA because the platform is entirely implemented in CUDA.

CUDA is very different from OpenGL and other programming languages. Compared to traditional GPGPU approach, the most important advantage of CUDA is the "scatter write" capability. Scatter write, by definition, is that in CUDA code you can write any data to arbitrary address in GPU memory, which is not possible in traditional GPU shader programming. With this capability, many of parallel

algorithms are possible implemented on GPU, such as parallel prefix sum and bitonic sort.

In addition, for communication and synchronization, there is *shared memory* on each multiprocessor, which can be shared among sub-processors. Shared memory has extremely high bandwidth compared to onboard GPU memory and can be used as user-managed cache.

Furthermore, as stated in CUDA official document, GPU is capable of executing a large number of threads in parallel with very low context switch overhead as shown in Fig. 2-2. From top to bottom, each CUDA kernel be can executed by a grid of blocks, and each block is composed of a number of threads. Each block is conceptually mapped to a single SIMD processor; however it's still possible for different threads to take different branch and set synchronization barriers. Threads inside the same block can communicate with others using per-block shared memory and synchronize when needed.

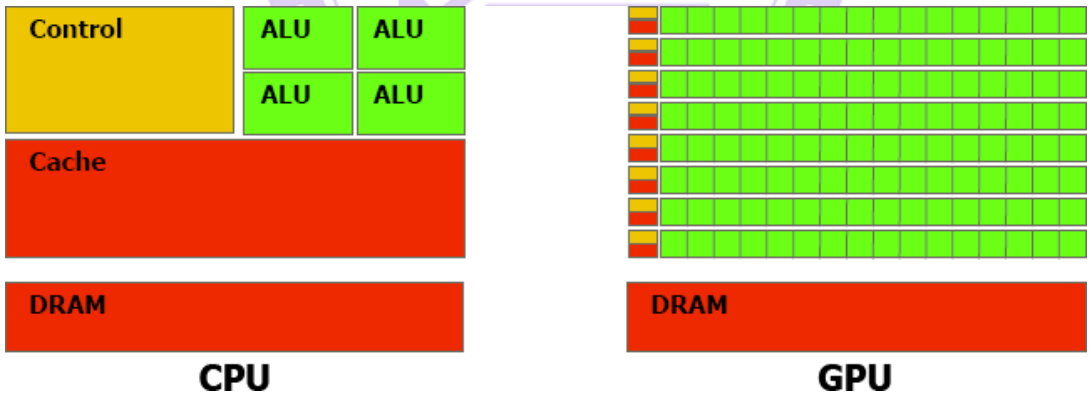

Fig. 2-2 GPU Devotes More Transistors to Data Processing

Name	Accessibility	Scope	Speed	Cache
Register	read/write	per-thread	zero delay (on chip)	X
Local Memory	read/write	per-thread	DRAM	N
Shared Memory	read/write	per-block	zero delay (on chip)	N
Global Memory	read/write	per-grid	DRAM	N
Constant Memory	read only	per-grid	DRAM	Y
Texture Memory	read only	per-grid	DRAM	Y

Table. 2-1 Memory Addressing Spaces Available in CUDA

Except for shared memory, actually there are six different types of memory for different purpose and access pattern, listed in Table. 2-1. The GPU memory layout is shown in Fig. 2-3. In fact, physically there are only two types of memory physically available on GPU: on-chip memory and off-chip memory. But for difference usage, they are divided into difference memory space and optimized for different purpose.

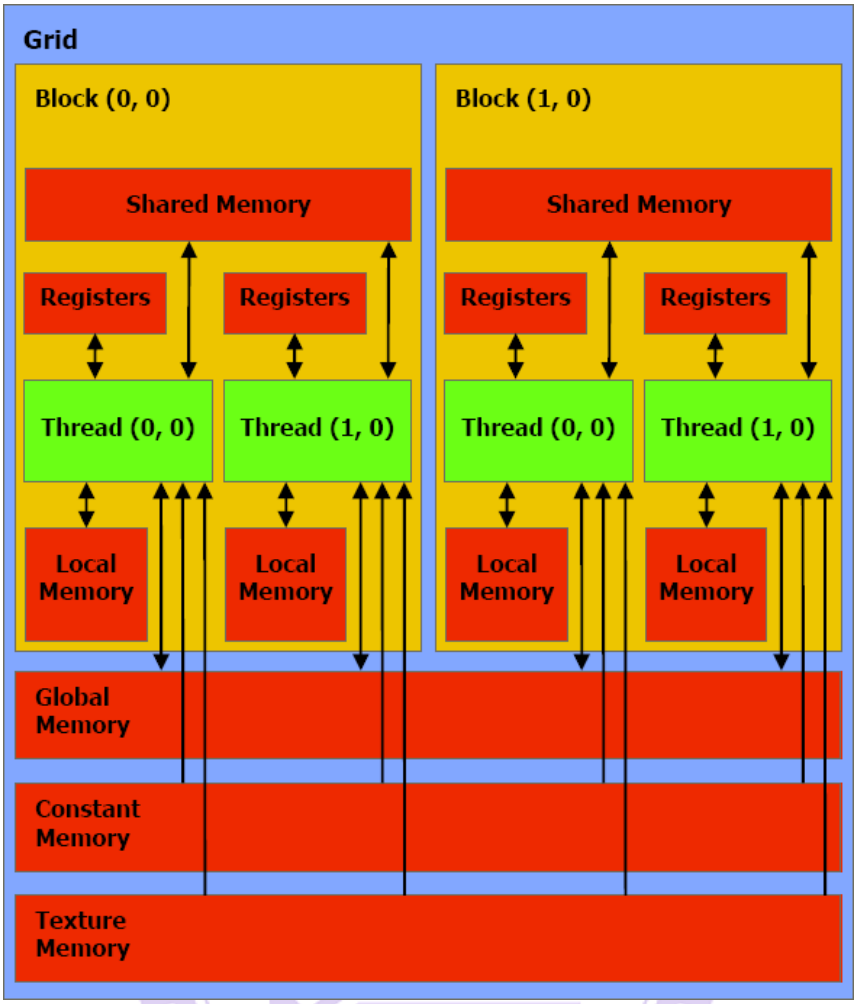

Fig. 2-3 CUDA Programming Model

The on-chip memory is embedded into the SIMD processor. This make the access extremely fast, which takes only 2 clocks to read or write when no bank conflict is occurred. Compared to on-chip memory, access to off-chip memory is very expensive which takes 200~300 clocks each operation. On-chip memory, which is usually refer to shared memory, is typically used as a user-manageable cache for SIMD processors to avoid duplicated access to off-chip memory. Also, shared memory can be used as for inter-communication among threads in the same block.

CUDA seems to be the best choice for parallel programming, but there are some limitations due to hardware design. First of all, the precision of floating point computation is limited. Only single precision floating point is supported right now. .

Second, there's no recursive function allowed on GPU, simply because stack is not presented on GPU. Third, the branching of threads within same block has some performance issue because that each block is a single SIMD processor which can only execute a single instruction at a time. If different threads take different branch paths, they will be serialized by the thread scheduler on GPU, thus penalty is introduced. Fourth, the data transfer between CPU and GPU memory is relatively slow due to the limited bandwidth of PCI Express bus. For more details, please refer to official CUDA programming guide [9].

2.1.2. Close to Metal (CTM) (AMD Stream™)

In comparison with CUDA, AMD/ATI announced the Close To Metal (CTM) [10] technology before CUDA. For similar goal of CUDA, CTM try to open the computation power of GPU and apply to general purpose computation. But instead of CUDA, CTM comes with no comprehensive toolkits. There are no compiler, linker and high-level language interface but only low-level, assembly like raw commands executable on AMD/ATI's GPU. The differences set a barrier for developers and make it less-attractive, even though the CTM covers almost all aspects that CUDA can do. Recently AMD made a swift to its strategy to add a higher level abstraction over CTM called Compute Abstraction Layer (CAL), and combine with Brook+ as AMD Stream Computing SDK. However, the development progress is still slow compared to CUDA and Brook+ is not as flexible as CUDA. That's why we choose to use CUDA finally.

2.2. Related Work

To solve the scalability problem while making it easy to use for most game

designer, we reviewed some popular game development tools and middleware here. Basically online game development involves many aspects, and one of the most important jobs is to design the game logic both in server-side and in client-side. Many tools exist nowadays to facilitate client-side game design such as popular Virtools [11].

Virtools provides an integrated development environment with many build-in functionalities such as 3D animation, logic design, world design and several add-on components such as physics, AI, and networking. The tool design is very flexible and allows creating games on many devices like PCs, web browsers, and even popular game consoles. The main design idea in Virtools is called ‘building block’ which represents an action, or said game logic. It abstracts the programming details and visualizes the logic as a flow chart, which is extremely easy for game designers. Virtools successfully reduced the complexity of game logic development by such modular design, so we adopt a similar concept in our code generator.

As for server-side middleware, many commercial products are available such as HeroEngine [15], Multiverse [16], and Bigworld [17]. Meanwhile, project DarkStar [1] and DOIT [12][13] are developed by open source community and our research lab. They adopt similar network architecture to improve the network scalability while maintaining low latency by adding an additional gateway layer between server and client. The gateway-server architecture has been proven to reduce the network latency without any compromise of scalability, flexibility, high-performance and ease-to-use. For game development customization and enhancement, both of them provide java-based plug-in framework and flexible network protocol.

3. Customizable GPU-Assisted MMOG Platform

In this chapter, we will describe how the MMOG platform is GPU-assisted and customizable. We proposed a modular platform. Each module serves as different role and provides different functionalities. Some of the modules make use of GPU to improve the processing performance, and since the capability of GPU is still limited to general computation, some modules are still running on CPU. We will give detailed explanations for each module in the following section.

3.1. GPU-Assisted MMOG Platform Architecture

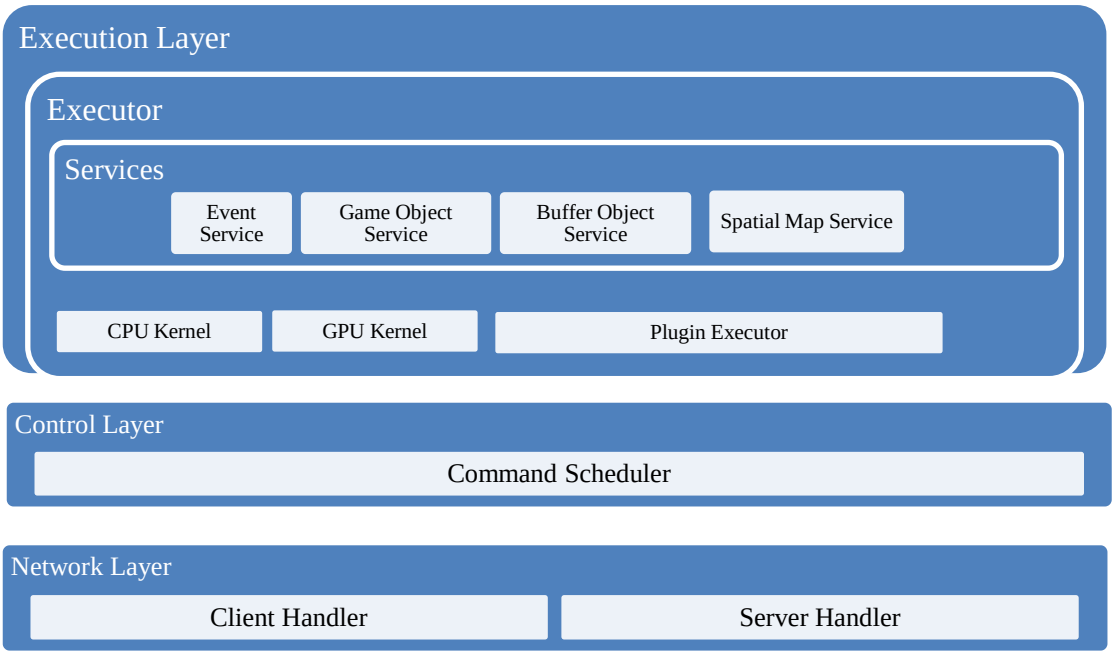

Fig. 3-1 GPU-Assisted MMOG Platform Architecture

The entire system architecture is depicted in Fig 3-1. Basically, system is divided

into three layers: network layer, control layer, and execution layer.

The network layer is the front-end interface between server and client. Commands from clients are first parsed by client handler, and then pass to the control layer. Some commands are not processed locally such as chatting to redirect to other server node through server handlers. The server handler will handle these kinds of network messages and make other separate services work with game server together.

In the control layer, server will translate commands into objects and send to command scheduler to wait for execution. The command scheduler will then dispatch each command to different executor in a batched manner, and finally collect the result from executor to reflect updates back to clients.

Execution layer is the logical processing layer in game server. Each command will be dispatched to either CPU or GPU specified by control layer, and then execution layer performs the actual processing on CPU and GPU. Execution layer hides the underlying details and abstracts the processor to provide a uniform game logic execution environment. In following section, we further elaborate each important module execution layer, and explain the relationship among them.

3.1.1. GPU and CPU Kernel

GPU/CPU kernel is the most important module in execution layer. It provides a uniform environment and abstracts the underlying processor unit by providing several primitives running on both CPU and GPU. Resource initialization such as memory allocation and thread pool allocation are all implemented in this module. The kernel module is responsible for initializing other execution-related modules such as plug-in executor module, event service module, game object service module, and so on.

3.1.2. Plug-in Executor

Game logics can be seen as a set of plug-in in our platform. Plug-in can be executed on either CPU or GPU by plug-in executor, which provides some basic programming construct to hide the complexity of programming on CPU and GPU, so plug-in code can be reused for CPU execution as well as GPU execution. Plug-in Executor is also responsible for controlling each service module to generate correct result, and all services are controlled by plug-in executor. For example, if a game object creation request is made during the plug-in execution, the plug-in executor needs to ask game object service to process the request the get a new game object id back. Such post-processing or pre-processing works are carried out at each service module, but they are all invoked by plug-in executor.

3.1.3. Event Service

Plug-in executor collaborates with event service to invoke corresponding plug-in in different execution context. Each plug-in can be triggered by one or many events, which can be dispatched through event service. Furthermore, we provide java style “synchronized method” design to solve the synchronization issue. We allow a plug-in to be executed in a “synchronized” way, that is, all invocations to that plug-in will be serialized to avoid inconsistency. Although such design would reduce the parallelism and introduce penalty to overall execution, it’s still crucial to synchronize concurrent modification to a shared variable, which is unavoidable in most game design.

As mentioned earlier, plug-in can be executed on either GPU or CPU in either synchronized or non-synchronized way, so we can say an event must belong to one of the 4 different scopes: CPU non-synchronized, CPU synchronized, GPU

non-synchronized, GPU synchronized. So, if a plug-in is triggered by a CPU synchronized event, it will be executed on CPU in a synchronized way. By analogy, if a plug-in is triggered by a GPU non-synchronized event, it will be executed on GPU in a non-synchronized way. Also, dispatching an event that belongs to different scope (from current one) is allowed. For example, you can dispatch a CPU synchronized event within a GPU non-synchronized plug-in and vice versa.

Event service is also the key to decouple the dependency of different plug-ins, and to chain plug-in execution, even they are in different scope.

3.1.4. Game Object Service

To talk about game object service, we need to define what a game object is and how it can be used. Game object represents a persistent entity in the virtual world. Each game object has several attributes, which allows game logic to obtain some information about a specific game object. For example, a player in the virtual is represented as a game object of one type; a building in the virtual world is represented as another game object of different type. Regardless their types, they all have same number of attributes, even though their meaning could be different.

Now step back to game object service. Basically, game object service is the main persistent storage of game object. It provides a set of APIs to get/set game object attributes, create/destroy game object, get/set type of a game object, and so on. Game object service manages all game objects and synchronizes the GPU storage with CPU storage.

3.1.5. Buffer Object Service

Compared to game object service, which stores game object attributes with

persistency, buffer object service is used to manage transient data set. Buffer object is used to hold temporary result, such as a list of neighborhood id, or a buffer array to store update results. Each buffer object is only valid until the end of next iteration of plug-in execution, so a typical use of buffer object is parameter passing between different plug-ins in different execution scopes. For example, by using buffer object, we can save some intermediate result in a buffer object, and pass the buffer object id as the parameter for the event triggering next plug-in in the execution chain, and load the intermediate result back to the plug-in and continue processing.

3.1.6. Spatial Map Service

The spatial map service provides the range search primitive which is widely used in most MMORPG to find out nearby players when an update needs to be broadcasted. Basically we implement this service on GPU by using the method in [14], but in addition, we extend the existing 2D range search to support both 2D and 3D spatial queries. Range query can be requested as a square range query in 2D or a cubic range query in 3D to support various game designs including MMORPG, MMOFPS, and MMORTS.

3.2. Game Logic Customization

Given that modular design of the GPU-assisted MMOG architecture, the game development work is still not hard without effective tools to help game designer design logics and write compatible GPU code. In addition, as GPU lacks of debugging capability, we need to a translation of game logic in some high level representation instead of real GPU code to be able to simulate the logic. The high level representation is imperative not only for debugging but also for developing some

GUI tool to make the platform easy to use. Therefore, we defined a configuration file format to describe any game logic flow and context. Code generator is implemented to translate the configuration file to real GPU routines provided by the platform.

3.2.1. Code Generator Architecture

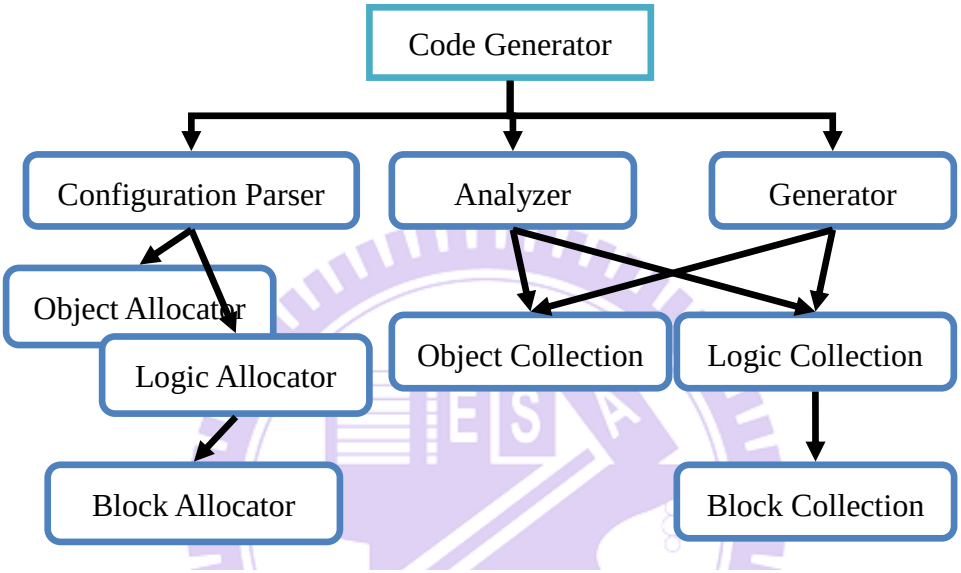

Fig. 3-2 Architecture of Code Generator

Fig. 3-2 is the architecture of the code generator. There are three important components in the generator, which are configuration parser, analyzer and generator. The generator will take a configuration file and output the corresponding code for GPU and CPU.

In order to transform the configuration file to code, we need to parse the configurations first, which is the job of configuration parser. But the parser doesn't know the details of each object, so we provide some allocators which are closer to the implementation. By using the allocators, parser will pass part of the configuration to allocators to generator corresponding objects. After parsing, there will several collections allocated by allocators to be shared by other components. With these collections, other components could know the description of the configuration file.

After that, we need to check the relationship between objects, logics and blocks, which is completed by analyzer. Finally, the generator generates the corresponding code for the configuration file.

4. Implementation Details

In this chapter, we describe the current implementation of this work. Since the platform implementation detail can be found in [14], we focus on code generation part only.

4.1. Configuration File Format

In current implementation, the configuration file provides descriptions of game objects and game logics. As shown in Fig. 4-1, for game object, designers can describe the attributes along with storage types as the specification of a type of game object by using <*game_object*> tag. For game logic, execution sequence can be by grouping <*block*> tags within a <*logic*> tag.

```
<?xml version="1.0"?>
<root>
    <game_object id="0">
        <attribute index="0" type="float" />
        <attribute index="1" type="float" />
        <attribute index="2" type="int" />
        <attribute index="3" type="uint" />
    </game_object>
    <logic id="123" boot="0">
        <block id="0" type="constant" next="1" constant-value="1" constant-type="uint" />
        <block id="1" type="constant" next="2" constant-value="12" constant-type="int" />
        <block id="2" type="constant" next="3" constant-value="4321" constant-type="uint" />
        <block id="3" type="constant" next="10" constant-value="3" constant-type="uint" />
        <block id="10" type="plus" next="20">
            <input index="0" source="0" source-index="0" />
            <input index="1" source="1" source-index="0" />
        </block>
        <block id="20" type="attribute_get" next="30" reference-id="0">
            <input index="0" source="3" source-index="0" />
        </block>
        <block id="30" type="parameter" next="40" parameter-type="uint" parameter-index="0" />
        <block id="40" type="condition">
            <input index="0" source="0" source-index="0" />
            <branch index="0" block-id="50" />
        </block>
        <block id="50" type="attribute_set" reference-id="0">
            <input index="0" source="2" source-index="0" />
            <input index="1" source="3" source-index="0" />
            <input index="2" source="1" source-index="0" />
        </block>
    </logic>
</root>
```

Fig. 4-1 Sample Configuration File

Enclosed by <*game_object*> tag, <attribute> tags are used to describe the attributes which is owned by the game object with logical index and storage type. In this way, we can define different types of game objects with different numbers of attributes.

Compared to <game_object> tag, which describes static data structure, the <logic> tag defines dynamic behaviors of game play. We use <block> tag to define a single action and chain all actions by using the “next” attribute in the <block> tag. The first block id is specified in “boot” attribute of the <logic> tag. The “type” attribute in each <block> tag indicates the real action taken by this block, as summarized in Table. 4-2. Furthermore, <input> and <branch> tags are used under <block> tag to define the input of this block and spawn new execution sequence if conditions applied.

By using <input> tag, every <block> tag can have none or one or many outputs. For example, if the type of the block is “attribute_get”, the number of outputs depends the number of attributes in the game object specification. The “source” and “source-index” attribute in the <input> tag is used to specify the “source-index”-th output of the “source” block. All available tags are summarized in Table. 4-2.

The <branch> tag in a <block> tag spawn a new execution sequence if the given condition applied. The execution will continue on the new branched block, and return to the current block when the new execution reaches its end.

Tag Name	Parent Tag	Attributes	Optional	Attribute Description
Root			N	
game_object	Root	Id	N	Unique Identifier
attribute	game_object	Index	N	Logical index in game object
		Type	N	Storage type of attribute
Logic	Root	Id	N	Unique Identifier
		Boot	N	First executed block
Block	Logic	id	N	Unique Identifier
		type	N	Pre-defined function block
		next	Y	Next executed block
Input	block	index	N	Index of input
		source	N	Source of input
		source-index	N	Source index of input
branch	Block	index	N	Index of branch
		block-id	N	Executed block

Table. 4-1 Available Tags

Type Classes	Types	Input/Branch/Output	Additional Required Attributes
Arithmetic	Plus subtract multiply divide	2 / 0 / 1	
Attribute_get	attribute_get	1 / 0 / Variable	reference-id (Type of game object)
Attribute_set	attribute_set	3 / 0 / 0	reference-id (Type of game object)
Condition	condition	1 / 1 / 0	
Constant	constant	0 / 0 / 1	constant-value constant-type (Storage type)
Loop	loop	1 / 1 / 0	
Parameter	parameter	0 / 0 / 1	parameter-type (Storage type) parameter-index (Logical index)

Table. 4-2 Available Types of Block tag

4.2. Code Generation

We implemented a code generator to translate the configuration file to desired GPU routines on our platform. In terms of architecture, there are three top-level components in the code generator as shown in Fig. 3-2: Configuration Parser, Analyzer and Generator.

Configuration parser parses the input and makes use of object allocator and logic allocator to make instances of game object and game logic according to the given configuration. In addition, game logic will use block allocator to get the instances of blocks described in configuration file. Theses object instances are stored in

corresponding collection. (i.e. game object instances are stored in the object collection, and game logic instances are stored in the logic collection respectively.)

Next, Analyzer analyzes the relationship among game logics and blocks, and checks the validity of the configuration by tracing the link between game logic instances and game object instances from top to bottom. Because there could be many invalid descriptions in a configuration such as invalid of inputs, non-unique identifier, and input count mismatch, corrupted routine may be generated with erroneous configuration and leads to incorrect runtime behavior of our platform. So it's crucial to remove any inconsistency in the configuration file. More code generation constraints can be considered and checked in this stage.

After validation of configuration, the Generator generates the corresponding GPU platform routines according to the given specification on a block-by-block basis. The generated routines conform to several specifications: CUDA language specification for GPU routine, C++ language specification for CPU routine, and the standard application interface of the platform.

Fig. 4-2 demonstrate the different pieces of code that would be generated by our code generator to plug customized game logic into our GPU-assisted MMOG platform. Generator needs to produce game logic code body as well as the logic dispatcher for both CPU and GPU execution. The dispatcher is used to invoke different game logic by different events. During the platform execution, the plug-in executor module will call the dispatcher and tell the event type along with an event parameter. The dispatcher then uses the information from the platform kernel and invokes the corresponding game logic. After finishing the execution of game logic, the dispatcher returns the control of execution back to the platform.

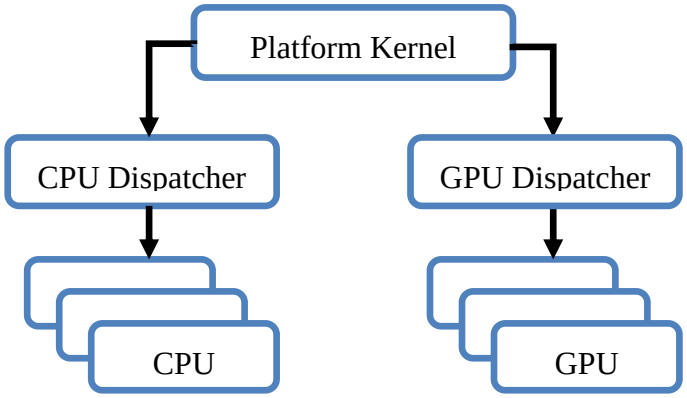

Fig. 4-2 Work Flow of Customized Platform

Take a step further, in the code analysis and generation stage, each functional block instance supplies following interfaces for Analyzer and Generator:

$$void\ pre\_analysis\_impl(logic\ \&parent)\ ;$$

$$void\ analysis\_impl(game\_object\_collection\ \&, logic\ \&);$$

$$void\ generate\_declaration\_impl(config\ \&, const\ logic\ \&, output\_stream\ \&);$$

$$void\ generate\_content\_impl(const\ config\ \&, const\ logic\ \&, output\_stream\ \&)\ ;$$

Analyzer uses the *pre_analysis_impl()* and *analysis_impl()* to perform functional analysis. The *pre_analysis_impl()* checks if all input references are valid, but no actual code analysis is performed here. The *analysis_impl()*, on the other hand, performs code analysis as well as code optimization. Take the condition block as an example, we check if the id of the branch block is valid in the *pre_analysis_impl()*, and perform constant value computation, if any, in *analysis_impl()*.

As for Generator, it makes use of *generate_declaration_impl()* and *generate_content_impl()* after analysis stage. The *generate_declaration_impl()* generates the declaration of variables at the beginning of the code because variables might be shared among following functions. Also, this is required if we want to

generate other GPU language which requires C-style variable declaration. (i.e. all variables must be declared in the beginning to allocate register space.) Fig. 4-3 is the pseudo code without optimization generated by the sample configuration.

```
uint     block_0_0;
int      block_1_0;
uint     block_2_0;
uint     block_3_0;
uint     block_10_0;
float    block_20_0;
float    block_20_1;
int      block_20_2;
uint     block_20_3;
uint     block_30_0;

block_0_0 = ( 1u) ;
block_1_0 = ( 12) ;
block_2_0 = ( 4321u) ;
block_3_0 = ( 3u) ;
block_10_0 = ( block_0_0) + ( block_1_0);
block_20_0 = attribute-get-0( block_3_0);
block_20_1 = attribute-get-1( block_3_0);
block_20_2 = attribute-get-2( block_3_0);
block_20_3 = attribute-get-3( block_3_0);
block_30_0 = parameter-get-0() ;

if ( block_0_0) {
  attribute-set( block_2_0, block_3_0, block_1_0);
}
```

Fig. 4-3 Pseudo Code of Sample

Currently, there are two kinds of optimizations employed in our code generator: constant value computation and unused block stripping. We analyze the code to find out all implicit constant values and output the so-called “constant blocks”, which has a single constant property. Fig. 4-4 is an example of the pseudo code after the first optimization.

```
uint     block_0_0;
int      block_1_0;
uint     block_2_0;
uint     block_3_0;
uint     block_10_0;
float    block_20_0;
float    block_20_1;
int      block_20_2;
uint     block_20_3;
uint     block_30_0;

block_0_0 = ( 1u) ;
block_1_0 = ( 12) ;
block_2_0 = ( 4321u) ;
block_3_0 = ( 3u) ;
block_10_0 = ( 13u);
block_20_0 = attribute-get-0( 3u);
block_20_1 = attribute-get-1( 3u);
block_20_2 = attribute-get-2( 3u);
block_20_3 = attribute-get-3( 3u);
block_30_0 = parameter-get-0() ;

if ( 1u) {
  attribute-set( 4321u, 3u, 12);
}
```

Fig. 4-4 Pseudo Code after Constant Value Computation

The unused block stripping is implemented by reference counting. The reference count indicates how many blocks are referencing to it. Therefore, blocks could be stripped itself out if all the outputs are not referenced. After applying the unused block stripping on the example code in Fig. 4-4, the code will be reduced to a single line:

```
attribute-set(    4321u, 3u, 12);
```

5. Experimental Result and Analysis

In this chapter, we try to evaluate the performance of our MMOG platform and compare with naïve CPU approach along with basic game logic generated by our code generator. As we tried all our efforts to deliver the most MMOG platform performance, different game logic design can result in different overhead. Considering the different architecture of GPU from that of CPU, we give two scenarios to illustrate the possible performance penalty. As we focus on the computation part, the command transmission time between client and server is ignored in all test cases. Each scenario runs for 32 times and we make an average for each of them.

5.1. Experiment Configuration

To work with CUDA, we have the following hardware configuration:

CPU	Dual-Core AMD Opteron 2216 x 2
Motherboard	TYAN Thunder n6550W (S1925)
RAM	2G DDR2 667 x 4
GPU	NVIDIA GeForce 8800 GTS 512M

Table. 5-1 Hardware Configuration

Since the performance comparison of CPU and GPU is performed, we list the specification of the GPU in details as follows:

Processor Name	GeForce 8800 GTS 512M (G92)
Number of SIMD Processor	16 (each contains 8 cores)
Number of Registers	8192 (per SIMD processor)
Constant Cache	64K (per SIMD processor)
Processor Clock Frequency	Shader: 1625 MHz, Core: 650 MHz
Memory Clock Frequency	970 MHz
Shared Memory Size	16K (per SIMD processor)
Device Memory Size	512MB DDR3

Table. 5-2 Graphic Card Configuration

As for software part, we use CUDA 2.0 beta released recently. Previous version 1.1 suffers from several bugs such as the variable length limit and incorrect code generation which make the code generation impossible.

Operating System	Ubuntu Feisty Fawn (7.04) 64-bit Version
GPU Driver Version	174.55
CUDA Version	2.0b
GCC Runtime	4.1.2 (Ubuntu 4.1.2-0ubuntu4)

Table. 5-3 Software Configuration

5.2. Results

5.2.1. Logic Execution Performance

In this scenario, we try to measure the performance when different game logic lengths are given. We made different logic length for 100, 200, 300… 1000 by

inserting a dummy variable-length loop and run each logic with different command count both on GPU and CPU. Table. 5-4 shows the result for CPU and Table. 5-5 is for GPU.

Compared to CPU, execution time grows when the length of logic becomes large but with a less rate. For example, we can found the increasing ratio of execution time on GPU with logic length from 100 to 1000 is about 1.5, while on CPU the ratio is 9.5. This is because the setup time of GPU is relatively large compared to GPU logic execution time, and if subtract the setup time, the execution scales linearly. Overall, the GPU suppress CPU in every test case, and in the most extreme case, GPU is 63 times faster than CPU.

Count\Length	100	200	300	400	500	600	700	800	900	1000
131072	69.1	130	190	250	310	370	431	491	552	612
262144	138	259	379	500	620	741	862	982	1103	1223
524288	276	517	759	1000	1241	1482	1723	1964	2206	2446
1048576	551	1034	1517	1999	2481	2964	3446	3929	4411	4893
2097152	1105	2069	3033	3999	4963	5929	6893	7860	8823	9786
4194304	2210	4136	6068	7998	9926	12k	14k	16k	18k	20k

Table. 5-4 Game Logic Execution Time on CPU

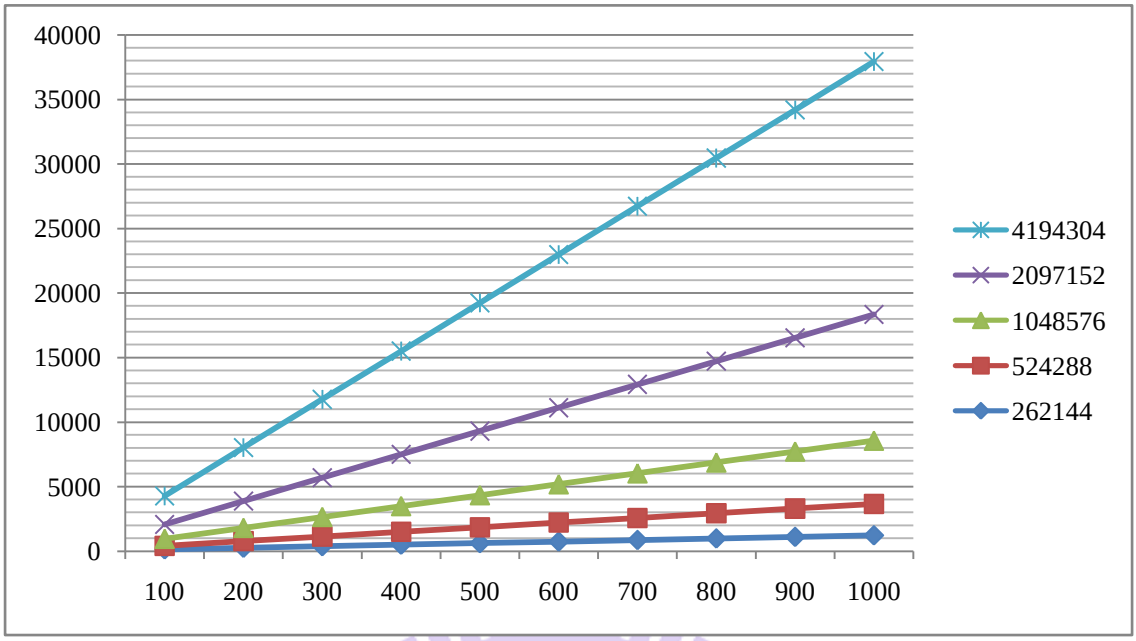

Fig. 5-1 Game Logic Execution Time on CPU

Count\Length	100	200	300	400	500	600	700	800	900	1000
131072	11.8	11.9	12.2	12.4	12.8	13.1	13.2	13.6	13.9	14.4
262144	18.7	19.3	20.4	20.4	21.0	21.5	22.1	23.2	23.2	23.7
524288	31.9	34.3	34.0	35	37.4	37.5	37.9	39.8	40.8	42.0
1048576	55.7	61.0	59.9	65.5	64.3	66.4	69.0	70.8	73.2	75.8
2097152	104	108	113	119	128	126	131	134	140	145
4194304	242	250	257	266	274	283	292	300	308	314

Table. 5-5 Game Logic Execution Time on GPU

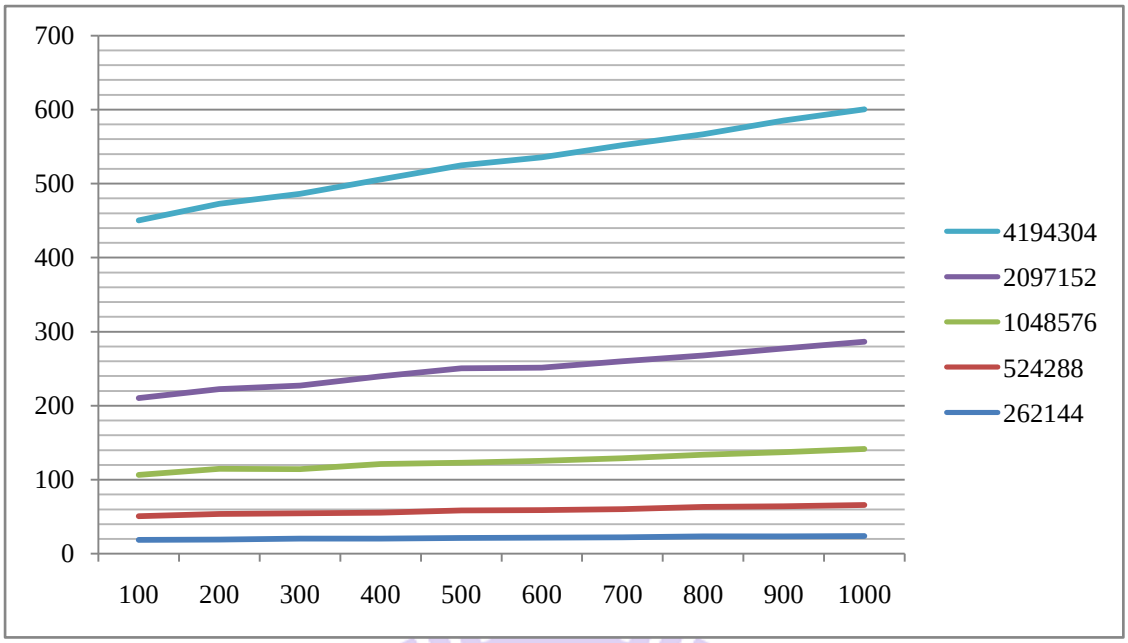

Fig. 5-2 Game Logic Execution Time on GPU

Since the setup time on GPU, we can find setup times for each command count, which is Fig. 5-3.

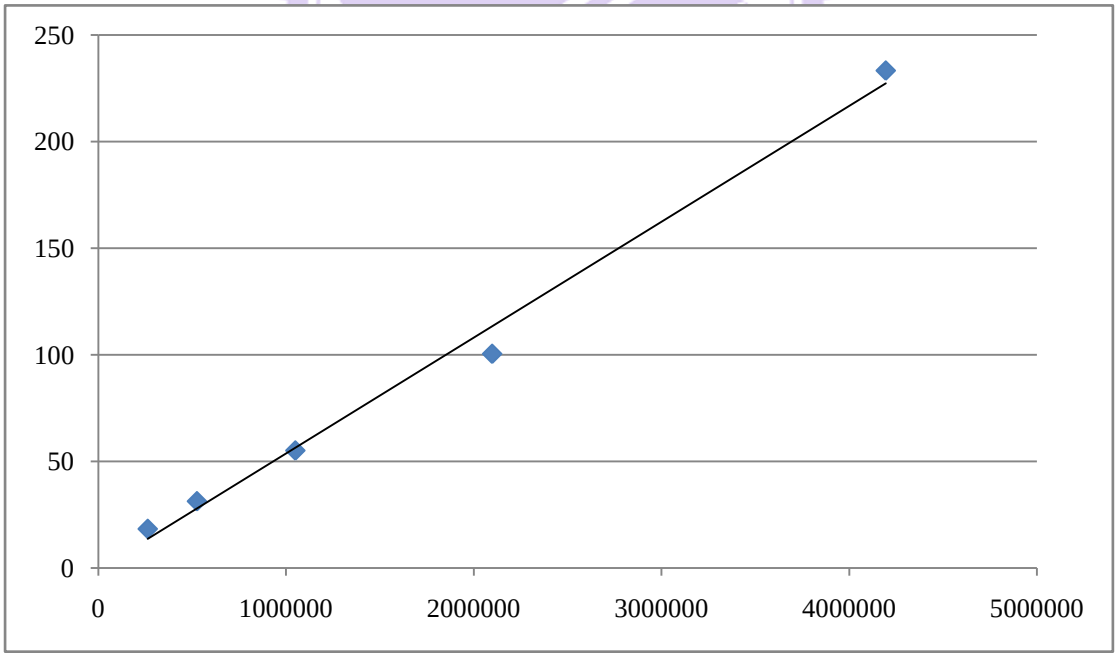

Fig. 5-3 Setup Times of Different Command Count on GPU

According to the results, we can derive an equation below by using methods for linear homogeneous recurrence relations:

$$z = 2\times10^{-8}\times xy + 5\times10^{-5}\times x + 6\times10^{-4}\times y - 0.6531$$

In the equation, x is the command count, y is the logic length and z is the execution time. Because the command count is large and the orders of magnitude, $5\times10^{-5}\times x$ will make the most of effect in the equation. This will make the execution time is highly depends on the command count, which shown in this scenario.

5.2.2. Branching Overhead

Modern CPU design employed branch prediction to hide the branching overhead in the CPU pipeline. However, unlike CPU, current GPU does not have this powerful feature. In CUDA, if different threads within the same block take different branching paths, the execution will be serialized by hardware thread scheduler, which incurs performance overhead.

To exam the degree of the overhead, we implement a test game logic in which 4 different branches may be taken to make the execution partially sequential on GPU. Each branch has a length of 1000 and one branch would be selected according to the parameter to each command. We specify the parameters as branching indices in a circular order, which is the worst case:

0, 1, 2, 3, 0, 1, 2, 3, 0, 1, …

The results are demonstrated in Fig. 5-4, and as you can see, GPU becomes 1.4 times slower when changing from no branch to 4 different branches. Meanwhile, as CPU is equipped with branch prediction, the overhead of branching is almost close to zero. However, the performance is still very good by comparing to CPU even if branching penalty is introduced.

Command Count\Branch Count	4	3	2	1
131072	318	324	319	310
262144	636	640	632	619
524288	1271	1280	1269	1299
1048576	2538	2558	2573	2533
2097152	5060	5135	5089	4963
4194304	10179	10186	10184	10214

Table. 5-6 Execution Time of Different Branches on CPU

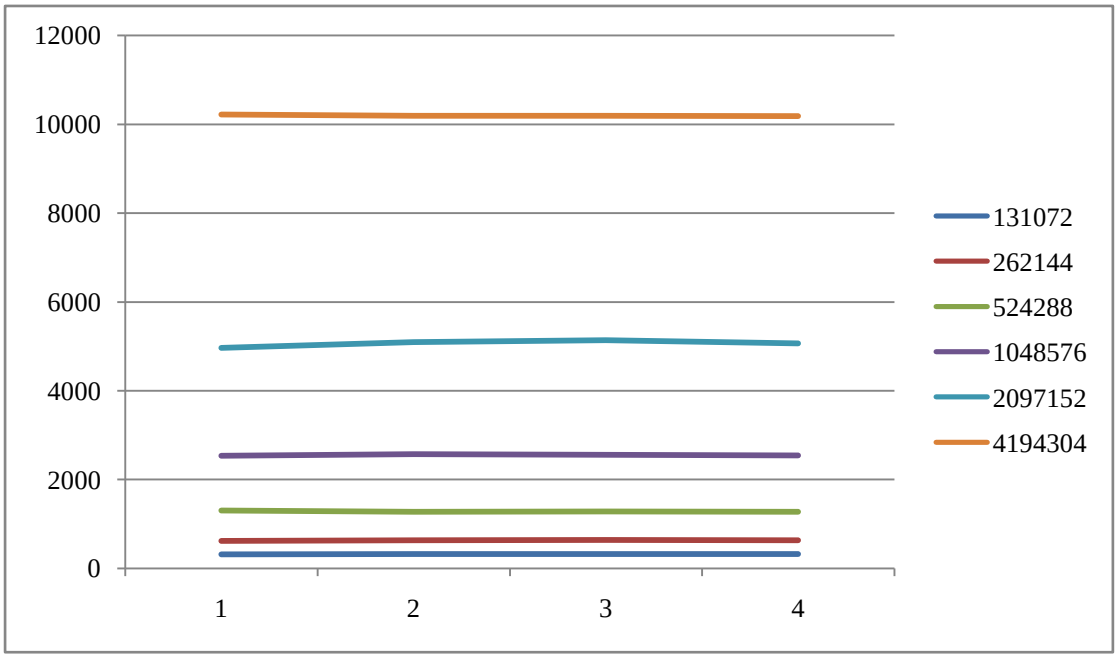

Fig. 5-4 Execution Time of Different Branches on CPU

Command Count\Branch Count	4	3	2	1
131072	16.9	15.4	14.1	12.7
262144	29.1	26.8	24.3	21.1
524288	52.5	47.1	42.9	36.3
1048576	94.8	84.7	74.7	67.8
2097152	209	185	164	144
4194304	373	328	301	275

Table. 5-7 Execution Time of Different Branches on GPU

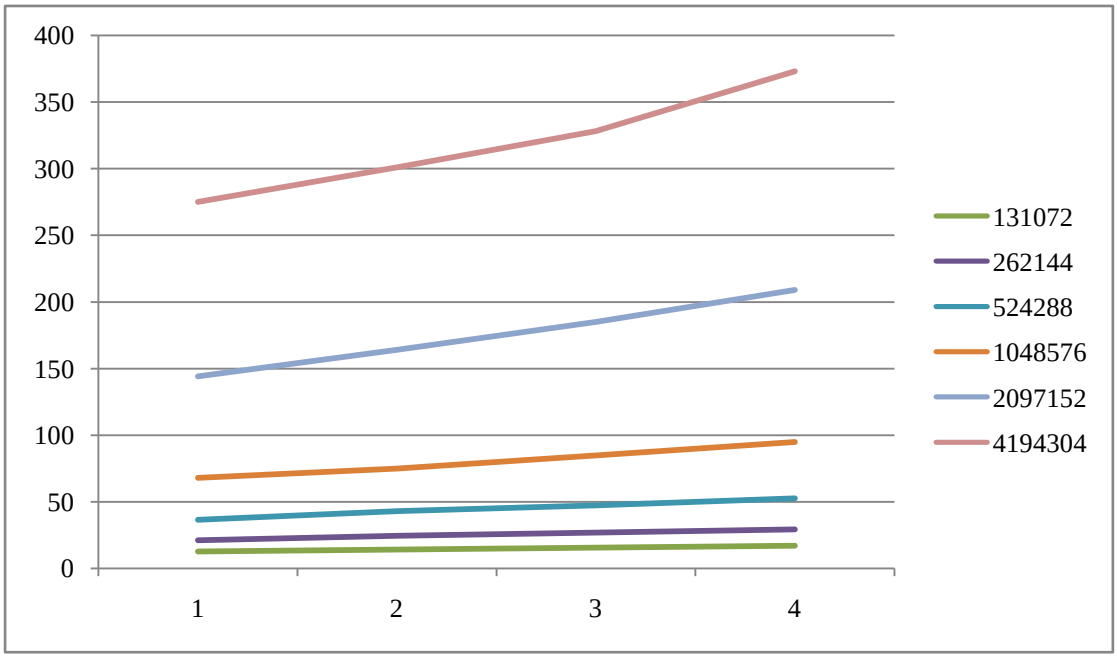

Fig. 5-5 Execution Time of Different Branches on GPU

By using the same method in previous scenario, we can derive an equation similarly:

$$z = 8\times10^{-6}\times xy + 6\times10^{-5}\times x + 1.331\times y + 2.4744$$

In the equation, x is the command count, y is the branch count and z is the execution time. In the equation, we can find the order of magnitude for xy and x is more closely. This means xy will effectively increase the execution time, too. In other words, xy and x both effectively increase the execution in the scenario.

6. Conclusions and Future Work

6.1. Conclusions

For modern MMOG development, a MMOG platform with scalability and customization is needed to shorten the cost of development. It can launch to the market as quickly as possible. In this research, we survey current situation of MMOG development and middleware solutions. We observe that the sequential processing of CPU architecture is not suitable for MMOG processing that needs a large amount of concurrent processing. To solve this problem, we implement a MMOG platform which utilizes the GPU to boost the concurrency. In the experimental result, the execution time of processing in CPU needs about 10 to 100 times of GPU, which indicates the number of clients growing by using GPU. But GPU has several limitations such as the memory space, the existence of both CPU and GPU logics are unavoidable. To solve this problem, one should consider putting only the mostly invoked logics should be processed on GPU and others on CPU.

With the growth of GPU computing power, boost the performance of MMOG server with exploiting the computation power of GPU is promising. We reveal a possible direction for middleware solution of MMOG with a higher computation power and larger virtual world could share with players.

6.2. Future Work

Although the experimental results show promising result in handling a large amount of players, there are still several works needed to be done in the future to improve the platform usability:

1. Dynamic Logic Dispatching to CPU or GPU:

Since the memory space of GPU is smaller than that of CPU, GPU can't handle all the commands when server has heavy load. Therefore, we can provide a dynamic logic dispatcher that forwards only the commands invoked most to GPU and CPU processes the others. With the dynamic logic dispatching mechanism, server can improve the performance without overwhelming the GPU.

2. Load Balancing on Multiple GPUs:

At present, multiple GPUs can be installed on single baseboard. A load balancer is responsible for coordinating GPUs' load. Thus, this feature can bring scalability to further improve the platform scalability.

3. More High-Level Function Blocks:

Currently, the platform provides basic function blocks for attribute manipulation, parameter fetching, condition, and arithmetic operations. However, high-level operations, e.g. AI functions, are often used in game design. We can implement these high-level function blocks to give more power to game designers.

4. Configuration Editor:

A configuration editor can help designers generating corresponding game logics instead of writing in XML syntax. By providing a graphical user interface, designers are able to drag-and-drop function blocks, and even simulate the game logic on the fly. In addition, a configuration editor can immediately visualize the errors and prevent erroneous code generation.

References

[1] Project Darkstar Community - http://www.projectdarkstar.com/

[2] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, Dinesh Manocha, CULLIDE: interactive collision detection between complex models in large environments using graphics hardware, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, July 26-27, 2003, San Diego, California.

[3] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, Dinesh Manocha, Fast computation of database operations using graphics processors, Proceedings of the 2004 ACM SIGMOD international conference on Management of data, June 13-18, 2004, Paris, France.

[4] OpenGL Architecture Review Board - http://www.opengl.org/about/arb/

[5] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, Aaron Lefohn: GPGPU: general purpose computation on graphics hardware, Proceedings of the conference on SIGGRAPH 2004 course notes, p.33-es, August 08-12, 2004, Los Angeles, CA.

[6] General-Purpose Computation Using Graphics Hardware Forum - http://www.gpgpu.org/

[7] HALL J. D., CARR N. A., HART J. C.: Cache and bandwidth aware matrix multiplication on the GPU. UIUC Technical Report UIUCDCSR-2003-2328 (2003).

[8] BOLZ J., FARMER I., GRINSPUN E., SCHRODER P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans. Graph. 22, 3 (2003), 917–924.

[9] NVIDIA CUDA Programming Guide, Version 1,1 - http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf

[10] ATI Close-To-Metal (CTM) Guide - http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[11] Virtools Official Site - http://www.virtools.com/

[12] DOIT - Chen-en Lu, Tsun-Yu Hsiao, Shyan-Ming Yuan. Design issues of a Flexible, Scalable, and Easy-to-use MMOH Middleware. In Proceeding of Symposium on Digital Life and Internet Technologies 2004.

[13] DOIT - Tsun-Yu Hsiao, Shyan-Ming Yuan, "Practical Middleware for Massively Multiple Online Games", IEEE Internet Computing (SCI), Volume 9, Issue 5, Sep/Oct 2005, pp.47-54

[14] Mu-Chi Sung, Shyan-Ming Yuan, Using GPU as Co-processor in MMOG Server-side Computing, 國立交通大學，資訊科學與研究所碩士論文，民國 96 年 6 月

[15] Hero Engine Website - http://www.heroengine.com/

[16] Multiverse Website - http://www.multiverse.net/

[17] BigWorld Technology Website - http://www.bigworldtech.com/index/index.php