UNIVERSITY OF OSLO
Department of Informatics

# Cheat Detection in On-Line Multi-Player Games using Graphics Processing Units

Master thesis

Martin Øinæs Myrseth

# Cheat Detection in On-Line Multi-Player Games using Graphics Processing Units

Martin Øinæs Myrseth

# Abstract

On-line multi-player games have experienced an impressive growth over the last decade. Despite this success, game providers struggle to keep up with the many different types of cheating occurring in these games. Due to the computational demand of a large scale multi-player on-line game, server resources are becoming scarce. This makes the task of implementing cheat detection mechanisms difficult, because of the lack of computational resources. Advances within the field of General Purpose computing on Graphic Processing Units (GPGPU), have given developers easier access to the computational power of the GPU.

In this thesis, we investigate what possible benefits there are of implementing a GPGPU cheat detection mechanism. We have developed a framework for a game simulator that includes a simple customizable physical engine and a cheat detection mechanism. We have created both a CPU and a GPGPU version of the cheat detection mechanism we have constructed. The GPGPU implementation runs on NVIDIA GPUs using the Compute Unified Device Architecture (CUDA) framework. We have also constructed a simple user interface to provide a graphical representation of the game simulator.

The results we have obtained from our research indicate that offloading cheat detection mechanisms to the GPU, increases the speed of the mechanism. We also discover that in addition to being faster, the GPU mechanism allows the Central Processing Unit (CPU) to perform other game relevant tasks while the mechanism is executing. Overall, our research shows that game providers can benefit from offloading certain parts of their server side processing to the GPU.

# Acknowledgements

First of all, I want to thank my advisors Håkon Kvale Stensland, Carsten Griwodz and Pål Halvorsen for their valuable feedback and help. I would not have been able to complete this thesis without their invaluable support.

I would also like to thank everybody at the Simula lab for many constructive discussions and for making the lab a pleasant workplace. Also many thanks to Simula in general, for providing a productive and comfortable work environment.

Finally, I would like to thank my family, friends and especially my girlfriend Ingeborg, for endless support and their unique ability to keep me motivated.

Oslo, August 3, 2009

Martin Øinæs Myrseth

# Contents

# List of Figures

# Chapter 1

# Introduction

On-line multi-player gaming has experienced an amazing growth over the last decade. Game providers of popular games must deliver reliable service to thousands of concurrent users, indicating a need for scalable solutions. Even when scalability is taken into account and thoroughly considered in the design of an on-line game, its service providers find that server side processing power is inevitably becoming scarce [8]. Along with the success of on-line multi-player gaming, cheating is prominent as one of the leading cases of malicious behavior performed by game players. It is in the best interest of game service providers to keep cheating to a minimum, but the demand for a stable service for resource intensive games, restricts the amount of resources that can be dedicated to cheat detection mechanisms.

Our goal with this thesis, is to determine if programming the Graphics Processing Unit (GPU), can help offloading cheat detection mechanisms in game systems. We also want to investigate how such a solution might scale, compared to a mechanism executing on the Central Processing Unit (CPU). If cheat detection mechanisms running on graphics hardware leave a smaller performance footprint compared to that of standard mechanisms, our research can give game providers an efficient alternative when implementing cheat detection mechanisms in on-line multi-player games.

## 1.1 Background and Motivation

On-line computer games are becoming increasingly popular, but with this growth, cheating is also advancing and could turn into a major problem for on-line game operators and service providers [9]. As an effect of the large variety of current on-line computer games, cheating has evolved into a complicated phenomenon. Adapting current client-server game architectures to support even basic cheat detection would certainly imply the need for architectural changes, as the scalability of game systems is already a major issue [8].

While adding more hardware to a system can increase its performance, it only serves as a temporary solution. Hardware used in commercial game server clusters is expensive, and the performance gained might only be sufficient for a short period of time. Because of the physical limitations halting the performance increase in the single-core CPUs, expansion needs to be horizontal rather than vertical. Vertical expansion is a common term used to explain the performance increase of single processing components withing a system. Horizontal expansion is where the overall system performance is increased by adding several identical processing units which all work in parallel. This process is called parallelization and has proved to be a solution to many slow and aging systems running serial algorithms.

Vertical expansion has for a long time been the solution for system administrators to increase the performance of their system. However, the process of adding new, faster hardware is slowly being substituted by migrating systems to run on parallel hardware. For this change to be beneficial, serial algorithms must be parallelized. The modern GPU is a relatively inexpensive example of such a parallel device. We aim to utilize general programming of GPUs to achieve a low-budget and scalable solution for detection of cheating in on-line multi-player games.

## 1.2 Problem Statement

Cheating in on-line multi-player games is an issue that has not been given the attention it deserves, neither from the game industry nor the scientific field. Many on-line multi-player games still suffer from excessive cheating in one form or another. However, in many cases the existence of cheating is hard to prove [10]. The only part of a distributed system that a game service provider can trust, is the part of the system running on hardware under their control, which is commonly the central servers. Any other part of the system can and most likely will be attempted to be exploited by a cheater.

As of yet, no existing framework manage to eliminate all kinds of cheating, so game developers are forced to either create their own mechanisms or use a selection of existing solutions to cover all the aspects of a game that a cheater might exploit. In-game physics, aimed to increase game realism, is experiencing increased popularity in many kinds of games. However, this is just another added feature where bugs can be exploited or modifications can yield advantages to dishonest players. Most games that have implemented in-game physics use it as major part of the game-play experience, some even base the entire game-play around physics alone [11]. In-game physics is therefore a very likely part of a game to be exploited.

Figure 1.1: *The anti-gravity gun of Half-Life 2 [1] from Valve Software allowed the player to move and throw game objects around. Courtesy of [2].*

To solve such a problem, central servers, or other trusted entities, must ensure consistency in the movements of all of the clients of the game. This is potentially a computationally heavy task, which most game servers would struggle to complete within real-time demand limits. Game servers need to run many concurrent tasks in addition to the cheat detection mechanism. The servers must maintain stable and reliable communication with all connected clients, accept new connections and terminate connections to clients that wish to disconnect. Security related tasks, like user authentication and account management, must also be performed concurrently with the cheat detection mechanism. We investigate alternative hardware where the cheat detection mechanism can execute to offload the main CPU. The goal is to allow game servers to run physics consistency checks whilst performing standard game administration tasks with considerable less trouble.

In this thesis, we will focus on game models which are based in *virtual worlds*. Virtual worlds are simulated virtual environments where partakers control one or several avatars[1]. If the virtual environment has a physical model, the movement of these avatars are often affected by a set of simulated physical laws, or forces. These forces are similar to the ones we experience in the real world; like gravity for instance. It is also common for participants to be able to move through different mediums; i.e., the air of an atmosphere or a liquid like water. All game participants will be affected by these mediums based on the structure and shape of their avatar. Cheating participants might find ways to modify the parameters of the physics engine calculating the way forces and mediums act upon the objects of the world. This way, the cheaters can gain unfair advantages over adversaries to achieve undeserved results and rewards.

The algorithms used to detect cheating participants have to be adapted to the physical model of the game. The entire cheat detection system of a game can consist of many different cheat detection mechanisms, if there is a need for it, so it is important to create a scalable framework for running a widely varied set of cheat detection mechanisms on a server. In this thesis, we have not researched how to best implement cheat detection mechanisms, mainly because mechanisms would rarely be usable unmodified over different kinds of games. Mechanisms must be tailored to the game that will use them. We have, however, designed two simple mechanisms to fit a game model we have defined ourselves. We are interested in how our mechanism executes and if it manages to offload the main CPU. Further work could be to research how different kinds of mechanisms running on GPUs can be optimized and which techniques work with which kinds of games.

## 1.3 Main Contributions

We investigate existing research and solutions to limit cheating in on-line multi-player games. Cheating and cheat detection are problematic issues in the game industry, because of the many types of cheating that exist. The scope of the different proposed and implemented solutions vary from detecting cheats where the client has modified his or her own system, to expel or disable the accounts of people selling in-game items and currency on on-line auction sites. We have researched what types of cheating is relevant to our thesis. Because modern AAA[2] game titles focus not only on top notch graphics, but also in-game real-time physics, several physics engine simulators have emerged in the last few years. We have researched the general theory behind these physics engines to gain

[1]Avatar is the common name for the game object (or objects) that the client controls.
[2]AAA is a common classification for high-budget game titles.

knowledge about how physics in games is implemented.

Our contribution is to investigate how a cheat detection mechanism can be implemented as transparently as possible in a game system by utilizing parallel hardware. We present our research with a concrete example of a game simulation framework we have developed. In the simulation, cheating clients modify physical properties of their game avatar to gain an advantage. The framework uses a Client-Server communication model and includes a physics engine, basic client management and the cheat detection mechanism. We investigate how parallel hardware, more specifically the GPU, performs when used for cheat detection. Originally aimed at graphics tasks, the GPU has proved to be an ideal platform for parallel development and execution with the introduction and advances of General Purpose computing on Graphic Processing Units (GPGPU). We use the CUDA framework and API from NVIDIA to develop and run cheat detection mechanisms within our game simulation.

We show that by using the GPU to offload a cheat detection mechanism, we can save CPU cycles that can be used for other processing. We also show that the execution time of the GPU mechanism is considerably lower than the CPU implementation. We hope our contribution to the matter of cheating in on-line multi-player games can help in the battle against malicious behavior in networked systems. We also hope this thesis will server as a motivation to inspire developers to reap the benefits of parallel computation and to contribute to the kind of development likely to become more prominent in the near future.

## 1.4 Outline

In chapter 2, we will look into what kinds of cheating exists in games in general and which types of cheating applies to this thesis. We investigate the basic theory behind common physics engines and what parts of this applies to our implementation in chapter 3. In chapter 4, we introduce the technology and architecture behind our cheat detection enabled multi-player game simulator, where we examine modern multi-core architectures and frameworks used to program and develop on these architectures. We present our implementation, including our physics engine and cheat detection mechanisms, in chapter 5. In chapter 6, we try to explain the results we have found and what might be the cause of these. Finally, we will give a brief conclusion and summary of the thesis in chapter 7.

# Chapter 2

# Cheat Detection in On-Line Multi-Player Games

To implement mechanisms against cheating in on-line multi-player games, we need to gain a better understanding of what kinds of cheating actually exists, and what has already been attempted and achieved to hinder cheating. In this chapter, we will therefore introduce existing work prior to this thesis, and explain why cheat detection and prevention is important to the game industry. We investigate how cheating exists in modern on-line multi-player games and review existing solutions to cheating.

## 2.1 Introduction

One of the leading factors for the attractiveness of all games, is their sense of fairness. Cheating compromises the fairness of a game and thereby also its appeal to users. Discontent users affect the reputation of a game and thus its revenue. Good revenue results in continuous game development which again leads to better games. Hence, it is in the best interest of both a game service provider and the clients to keep cheating to a minimum, to maintain the best possible user experience. However, cheat detection and avoidance systems are not trivial to implement and maintain because of the diverse means of which cheating is employed [9, 10, 12]. To gain an understanding of the different types of cheats that exist in current games, and to be able to distinguish which varieties of cheats are relevant to which types of games, a classification of cheating is needed.

## 2.2 Classification of Cheating

Through the evolution of on-line multi-player games, cheating has emerged to become a serious problem for game providers. Cheating can ruin in-game economics, turn honest players into cheating players and in the worst case, lead to players abandoning the game [13]. This way, cheating can be a major factor affecting the success of a game. The diversity of the current games being played on the internet allows for equally diverse means of cheating, as each genre of games have their own unique exploitable vulnerabilities. The first important step towards a cheat-free game, is to examine and determine which forms of cheating are most likely to be attempted in games of the type being designed and developed. For this reason, cheating must be categorized and linked to the types of games where the certain cheat is feasible. This depends on many different attributes of a game, such as its architectural structure, its style or genre, player-to-player interaction and how security and privacy should be handled in the game.

An early review of the existence of cheating and its prevention was done by Matt Pritchard [10]. As one of the developers of *Age of Empires* and an avid gamer, he has experienced cheating and its impact on both the developers and game players. His paper, aimed at the game development industry, mentions concrete examples of games which have experienced problems with cheating, different game communication models and how cheating applies to these models. He also presents several ideas on solving different cheating cases. Not much scientific effort had yet been put into the field of cheating behavior in multi-player games before his review. Mostly because at the time, the multi-player on-line gaming industry was still in its infancy.

As an early research field, the study of cheating in multi-player games was unstructured, without classifications of the different existing methods of cheating. Cheating problems were largely investigated and dealt with on a case-by-case basis until Yan and Randell [9] presented an extensive list of different categories existing cheating might fall into, and with it, a taxonomy of on-line cheating. It is a three-dimensional taxonomy based on what is *the underlying vulnerability*, *the cheating consequence* and *the cheating principal*, which translates to: what within the game is exploited, what type of failure can this exploit cause and who is performing the cheating, respectively.

The taxonomy presented in [9] is thorough, but unstructured, so GauthierDickey et al [14] present a more structured taxonomy by categorizing cheats in the layer in which they occur: *game*, *application*, *protocol* and *network*. Continuing from this, Webb and Soh [12] present an updated review and classification of cheating in networked computer games based on the same categories defined by [14]. They have renamed the network category

| | |
|---|---|
| **Game Level** | Bug<br>Real Money Transactions |
| **Application Level** | Information Exposure<br>Invalid Commands<br>Bost/reflex enhancers |
| **Protocol Level** | Suppressed update<br>Timestamp<br>Fixed Delay<br>Inconsistency<br>Collusion<br>Spoofing, Replay<br>Undo<br>Blind Opponent |
| **Infrastructure Level** | Information Exposure<br>Proxy/Reflex Enhancers |

Table 2.1: Classification of certain game cheats into categories. Adapted from [12].

to *infrastructure*. Table 2.1 illustrates the four categories and which types of cheats that falls under these. The four categories are defined as follows:

**Game level cheats** Achieved by breaking the rules or misusing features of the game. Game level cheats do not require any modifications to the game client or the general infrastructure. An example is Real Money Transactions, which are common in Multi-Player Online Games (MMOGs). People are able to retrieve valuable items in a game by buying these with real money and exchanging the items within the game. Many MMOGs use this as the revenue plan for the game, while several other MMOGs, mainly western games, forbid Real Money Transactions [12].

**Application level cheats** Cheating by modifications to the code of the game or the operating system. A rather common form of application level cheats are reflex enhancers and farming bots which give the cheater an unfair advantage by boosting such as the accuracy of the aim or allow for automation of certain tasks to let the cheater gain resources while not even playing the game. An example of a farming bot is MMO Glider [15] for World of Warcraft (WoW) [16].

**Protocol level cheats** Changes to the protocol of a game (i.e., changing packet contents or delaying packets). Fixed delay cheats are based on introducing a delay before sending packets from the cheater. This delay appears only as latency for the other players and the central server. The delay can allow the cheater to examine all updates received from the other players before choosing an appropriate action based on the knowledge acquired.

**Infrastructure level cheats** Cheats involving modifications and manipulations of game dependent pieces of infrastructure. i.e., modifications to driver, libraries, hardware, network, etc. Information exposure cheats can examine broadcasted network traffic to give additional information to a cheater. Reflex enhancers can run on proxies that are placed between the cheater and the server and can modify packets to give the cheater an advantage.

Many of the categories of cheating Yan and Randell mentioned in [9] can be classified as general security issues for internet based applications and do not apply just to games. Security in on-line applications is a research field much more explored compared to that of cheating. Webb and Soh [12] discard these categories as cheating to separate general security issues from cheating, which is unique to gaming applications. Some categories from [9] that fall outside of the classification scheme of Webb and Soh are: *compromised passwords*, exploits due to *lack of authentication* and *unreliable communication* between participants or servers. Cheating that falls under several of the other categories listed in [9] can be caused by player interaction on other communication channels than the ones intended by the game. Players might exchange information between each other by phone to beat a common opponent in a situation where cooperation might not be intended (cheating by collusion). In addition, untrustworthy personnel might compromise all kinds of private game data.

Since cheat classification is not part of the research in this thesis, we will settle with the classification scheme proposed by Webb and Soh [12].

## 2.3 Protecting Online Games Against Cheating

One of the main problems with implementing cheat detection and prevention mechanisms in games, is the loss of valuable computational resources to the execution of the cheat detection mechanism. Game developers strive to create revolutionary games, with the latest features in the fields of graphical effects, in-game physics, low-latency communication and artificial intelligence. These features already require most of the resources of a computer, both in the server and the systems of the clients. A mechanism for the detection of cheating is only usable if its impact on the application is very transparent, both with regard to performance demands and modifications to the existing infrastructure[1].

[1]Changes to existing infrastructure would mostly affect games that have already been released to the public and can be considered in active use.

### 2.3.1 Communication Models of Networked Games

The architectural model of the game affects the approach to take when implementing cheat detection mechanisms. The two major architectures used in game design are Client-Server (C/S) and Peer-to-Peer (P2P) models, although P2P games can be said to be rather rare. The general communication models of the two architectures are illustrated by figure 2.1. C/S models consist of centralized servers controlling game state and communication flow. Every client is connected to the server alone, and all communication between clients passes through the server. P2P architectures are decentralized, meaning there is no single, controlling server. Clients must communicate with each other and exchange game state themselves. There are also proposals for hybrid architectural models, mixing C/S and P2P models to gain advantages from both of them [17].

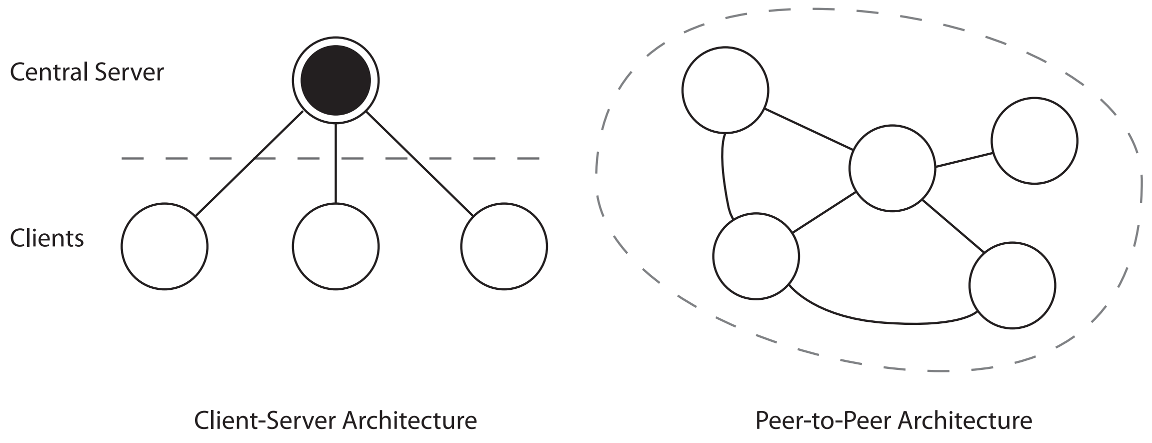

Figure 2.1: *Client-Server architecture vs. Peer-to-Peer architecture.*

The game industry still turns to the C/S architecture, not only for MMOGs, but for most game types. With MMOGs like WoW [16], a server is the host of just one game instance. The server is also under the control of the game service provider, Blizzard. This gives the game service provider total control of the server and how it behaves. With total control of the server, implementing cheat detection can be considered relatively easy. In First Person Shooters (FPSs), however, servers are often distributed and hosted by clients themselves, like Counter-Strike: Source [18]. A client hosts a server for a small group of peers who can then join the server. A disadvantage with a solution like that is that the connecting clients have no way of knowing if the server has been tainted or not.

Scalability is a main goal for game service providers during the entire lifetime of a game, as a growing user base would increase revenue. If game producers want to create and maintain a successful game, an architecture supporting thousands of active clients must be at the base of the system. If the infrastructure risks performance shortcomings as the game expands, the actions needed to meet the new performance requirements should be minimal. That being said, with a scalable infrastructure, there should be no need for overcompensations to maintain a stable service, because expansions will be quick and transparent, and if possible, automatic. In other words, a system that manages to maintain a stable service during a rapid increase of clients, connections or in-game events, is not scalable if the way it handles such an increase is solely by running redundant idle servers that wait to offload the main servers.

C/S communication models suffer from the fact that scalability is an issue. Since processing power in a single machine is not infinite, a single server is not capable of serving a steadily increasing number of participants forever, as the demand for processing power will reach an upper bound. A simple solution, and probably the most common, is to add new hardware to the infrastructure which offloads critical parts of the system. A problem which is not solved by adding more hardware is increased bandwidth demand. Not only is the internal processing power of the server an issue, the network bandwidth must also be able to serve the rising number of clients. Another problem is that the server is also the single control unit in the model, which means it is also a single point of failure. If a central server goes down, the entire game might come to a halt, whereas in a P2P architecture, the game may continue to run without problems. Communication between clients internally in the game also suffers from the fact that messages must be relayed through the central server, increasing the time a message would take to arrive, compared to the time it would take through a direct client-to-client connection.

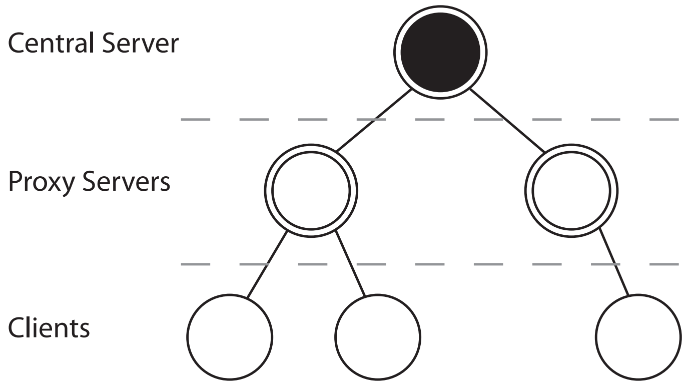

Figure 2.2: *Client-Server architecture with proxy servers.*

Proxies, illustrated by figure 2.2, have been a common solution to internal resource problems, bandwidth issues and high latencies. Proxy servers are helper servers that are placed between a central server and the clients. Proxies allow for servers to be placed closer to the clients, reducing latencies. Because all communication passes through the proxies, the bandwidth requirements of the central server can be reduced. The central server is still the controlling part of the system, but much of the information passed from the clients to the proxies does not have to be forwarded to the central server, reducing the bandwidth demand of the central server. Finally, if a central server would go down for some reason, the proxies might be able to keep the game running until the server is back up again or replaced.

P2P systems have been shown to deal with scalability issues better than C/S systems in the sense that expansions give a collective increase in resource demand, while in C/S models, the resource demand growth must be handled by the central server alone [19]. C/S systems may therefore have a lower overall system cost increase with the increase of clients, but the central server can not maintain this increase forever. However, a pure P2P network model, where clients maintain a connection and communicate with all the other clients, has been proved to suffer from scalability issues as well. MiMaze [20] is a distributed game created for research of distributed game development. MiMaze suffers from these scalability issues, supporting fewer than 100 players [19]. The main question is: where in the system is it best to place a cheat detection mechanism? P2P architectures have the disadvantage that implementing a collective cheat detection mechanism is challenging [14, 17, 21], but in such systems the processing power is available. The opposite is true for C/S architectures, where implementing a cheat detection mechanism is relatively easy, but processing power is scarce.

Although not very common, there are games that have chosen P2P as the communication model. The Genie Engine of Age of Empires I and Age of Empires II support P2P communication [22]. Age of Empires was an early game to feature internet-based multiplayer possibilities and despite its commercial success, it was very vulnerable to many forms of cheating [14]. Age of Empires I bypassed some of the scalability issues by letting every client calculate game progress, reducing network traffic. This did, however, introduce the issue that the game could not progress faster than the slowest connected client. FreeMMG [23] is a scalable, hybrid game distribution model that uses a lightweight server that delegates the bulk of the game simulation to the clients.

### 2.3.2 Existing Cheat Detection Mechanisms

Because of the many existing forms of cheating, there are many attempted solutions to battle cheating, both within academics and the game industry. Different types of cheats apply to different types of games, so the solutions we investigate in this section, approach different problems with different communication models and game types. Some are designed for C/S systems, while others for P2P. There are also systems that are usable with whichever communication model the game has. Other cheat detection and avoidance systems only apply to certain game genres and so on.

GauthierDickey et al [14] present a low delay event ordering called NEO, for P2P games. They claim it to be secure for five different types of cheats defined as protocol level cheats. *Fixed-delay* cheats are handled by ignoring late updates. *Timestamp* cheats are avoided by encryption. After a round, updates are decrypted and players can no longer submit actions. *Suppressed update* cheats are circumvented as the other clients will stop sending updates to the cheater, because the missing updates from the cheater will be perceived as congestion by peers. *Inconsistency* cheats can be dealt with by using digital signatures and state comparisons. Players perform state comparisons by regularly auditing the game state. Cheating by *collusion* is avoided by decreasing the probability of colluding players forming a majority.

Mönch et al [24] present a solution that prevents modification to the code of the game-client, and prevents uncontrolled access to sensitive game data stored in the memory of the client. It does not detect cheats concerning modifications to network data between different nodes in the game. The system relies on *mobile guards* which ensure the integrity of the protection mechanism and dynamically change and extend the protection mechanism used in a game. In essence, they are small code segments changed with small time intervals, making them impossible to reverse engineer before they expire. The solution greatly increases the difficulty of cheating in a game. However, the solution is also dependent on the creation of trustworthy mobile guards which is non-trivial in P2P games. The solution also increases client-side computation demand which affects the performance of the clients.

Baughman et al [25] have created a system where lookahead based cheats are avoided. A lookahead cheat is where a cheating participant knows which actions the other participants are going to perform in the next time frame, before choosing his or her own action. Normally, the cheating participant would run an automated agent of some sort, that initiates a defensive or offensive action based on the actions received from the other players. Their solution is a variation of the lockstep protocol, called Asynchronous Synchronization (AS). The lockstep protocol prevents lookahead based cheats, with the disadvantage that packet delay is introduced. AS reduces this delay by assigning a Sphere of Influence (SoI) to each client. Whenever a client falls out of synchronization with another client, the two clients are no longer within each others SoI. Clients continue to synchronize with other clients within their SoI until another sphere intersects with the same sphere. Then the two are joined to form a new SoI, which is then synchronized.

Webb et al [17] have come up with a system for cheat protection in games they call Referee

Anti Cheat Scheme (RACS). The system is primarily designed for P2P based games. The anti-cheat scheme depends upon a referee being in charge of the game state. The referee acts similar to a central server in a C/S architecture. Communication between clients can follow two ways: (1) relayed through the referee, Peer-Referee-Peer (PRP), which is the way communication is handled in C/S games and (2) between peers directly (PP). Their anti-cheat scheme would most likely also be possible to have in a C/S based architecture. RACS is in fact a hybrid between P2P and C/S inter-node communication wise. They present the basic concept of RACS and that it limits referees to be processes running on trusted hosts. In [26], Webb et al present a way to select referees in a strictly P2P based game, where referees can be normal game clients. This way, cheat detection can be distributed amongst the clients in a game, ensuring a low probability of corrupt referees, while maintaining the benefits of a P2P communication model.

There are several anti-cheating systems that have been put to mainstream use. Two of the most notable are Valve Anti-Cheat System (VAC) [27] and PunkBuster [28]. Warden [29], developed by Blizzard, is also worth mentioning as it is used to protect most of the newer games developed by Blizzard, most notably World of Warcraft. The similarity between the three mentioned anti-cheat systems is that they are separate programs that examine programs running alongside the game being played. They inspect the main memory of the computer for programs altering or reading the memory of the game. Valve reported over 10.000 cheating players of Counter-Strike: Source were caught with running cheating software within a single week in late 2006 [30].

We can only mention a fraction of the work attempting to reduce cheating in on-line multi-player games, but even from what is listed above, it is apparent that cheating can be dealt with in many ways. While Mönch et al prevent modifications to game-clients, which are application level cheats, GauthierDickey et al prevent several protocol level cheats. RACS aims to prevent cheating by selecting trusted referees that examine and hold the game state. RACS relies on systems like VAC or PunkBuster to avoid some game and application level cheats, showing us that cheat detection mechanisms can be used together to increase the protection against cheats.

Rather than to attempt to solve many forms of cheating, we investigate ways to effectively implement cheat detection mechanisms into games. We focus on parallel hardware and how it can be utilized to make the impact of a cheat detection mechanism on the game system as transparent as possible. The cheats that can be exposed by our cheat detection mechanism would be application, protocol and infrastructure level cheats. Mainly such cheats where there are modifications to the client application and network packets aiming to improve physical properties. Cheats have also been discovered where clients increase the internal clock speed of their computer, increasing simulation speeds so that objects may accelerate faster. These kinds of cheats can also be discovered.

## 2.4 Summary

In this chapter, we have looked at reasons why cheat detection should be a priority for both game service providers and gamers. We have given a short classification of the different ways cheating exists in on-line multi-player games. Because of the large processing demands of modern on-line multi-player games, we have had a focus on the importance of scalable solutions and performance. Before we move on to the underlying hardware and frameworks that form the base of our implementation, we will investigate some of the theory behind the physical models of modern games in chapter 3.

# Chapter 3

# Physics in Modern Games

To determine if a client has modified his or her system to exploit the physical model of a game, we need to check the reported movements of all clients and see if these are possible with regard to the physical model. A physical model is a model that defines how objects move through a virtual world, and how they might interact with each other. Physical models are enforced by physics engine simulators, or just physics engines for short. To perform such a check, we need to know how typical physics engine simulations work, by investigating the theory behind them. In this chapter, we focus on simple physics theory that would be essential to any physical model, and that is used in all physics engines. We present some existing physics engines, before describing what we have decided to include in our own implementation of a simplified physics engine (see section 5.3).

## 3.1 Introduction

In modern games, there is not only a large focus on graphical realism, but also giving the user a realistic interaction with the gaming environment. Games have evolved from being constrained within static boundaries, to fully animated and interactive worlds, where objects can be moved around and collide into each other. In the same way, the movement and actions of a controlled game avatar is constrained by physical laws implemented in the game. Mass inertia, viscous drag and gravitational pull are just a few of these physical properties game objects adhere to. A consequence of adding physical properties to a game, is the substantial increase in its computational demand. Every animated object must be affected by the environment and the other objects according to the specified rules, which means every physical property must be calculated for every object.

Performing all physics calculations on the server in a MMOG, would drastically increase game latency and degrade the realistic gaming experience. For this reason, most of the physics of a game is calculated on the machines of the clients. As we stated in section 1.2, computations performed by the clients can potentially be exploited by malicious users. The solution would be to perform consistency checks of the movement data coming from the clients, but even performing occasional checks for consistency by using conventional methods might overstrain a server. This could mean a considerable part of a game is exploitable, and with no checks to uncover the exploits. We investigate if it is possible to add consistency checks that have a smaller performance footprint to a server, by offloading those calculations to a GPU. We know that PhysX [31] benefit from performing such physical calculations on the GPU.

To implement both a physics engine and a mechanism to check the consistency of movement generated by such an engine, we need to know the theory behind general physical models and physics engines. Most of the formulas and the theory we discuss in this chapter is based on [32]. While a considerable part of the material in this chapter is general physics, much aid came from this book when we implemented the theory into a game simulation.

## 3.2 Basic game physics - Linear Motion

Which physical properties are implemented and used in a game, depends on the type or genre of the game. However, for most games, the central concept of their physical implementation revolves around Newton's second law of motion:

$$\sum \mathbf{F} = m\mathbf{a} \tag{3.1}$$

In short, the law states that the sum of all forces $\sum \mathbf{F}$ acting on an object, is the product of the mass $m$ of the object and its acceleration $\mathbf{a}$. Also, because most games are situated in a multidimensional environment[1], the forces acting on objects must be considered vectors with a given direction and magnitude. This is a consequence of $\mathbf{a}$ being a vector value, and thus $\sum \mathbf{F} = m\mathbf{a}$ must be a vector. We will typeset vector values with a bold font to distinguish them from scalar values.

[1]Multidimensional environments are environments where coordinates are made up of more than one component, each along a dimensional axis. An environment with two dimensions contain coordinates on a plane, while a three dimensional environment contain coordinates in a virtual space, as depth is the third dimension.

Equation 3.1 illustrates an important property with real-life objects; inertia. When we rewrite equation 3.1 to:

$$\mathbf{a} = \frac{\sum \mathbf{F}}{m}$$

we can see that when $\sum \mathbf{F}$ is constant, changing the value of $m$ affects the acceleration of an object. If the sums of the forces acting on two objects of unequal mass are $\mathbf{F}_1$ and $\mathbf{F}_2$, and where $\mathbf{F}_1 = \mathbf{F}_2$, then the object with the lowest mass will experience the largest acceleration. In other words, objects with higher mass will be more resistant to change its current motion. If the magnitude of $\sum \mathbf{F}$ acting on an object is zero, the object will maintain its current motion in a straight line, or stay still if it was not moving. Otherwise, the object will accelerate in the direction of $\mathbf{a}$. We can define inertia as the resistance an object has to accelerate.

Acceleration is defined as the value of which the velocity $\mathbf{v}$ of an object is changing at a given point in time. This means that $\mathbf{a} = f_v'(t)$, where $\mathbf{v} = f_v(t)$. This is the same as stating that $\mathbf{a} = \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}$. When the acceleration is constant, the velocity can be found with equation 3.2:

$$\mathbf{v}_2 = \mathbf{v}_1 + \mathbf{a}t \tag{3.2}$$

The linear displacement[2] of an object $\mathbf{s}$, is defined as the sum of the changes to its speed over time. When the speed of an object is constant, the displacement can be found with equation 3.3:

$$\mathbf{s}_2 = \mathbf{s}_1 + \mathbf{v}_1 t + \frac{1}{2}\mathbf{a}t^2 \tag{3.3}$$

By implementing these simple concepts into a game, static game objects begin to turn into animated life-like objects, creating realistic environments in the virtual world. If these laws somehow get broken, or the parameters to the functions altered, the movement of certain objects would seem unnatural. Since an object is only able to change its speed as quickly as the maximum acceleration in a direction allows, sudden stopping or turning is perceived as defying the physics of a game.

The physical model begins to get more complicated once the need for non-constant ac-

[2] We define displacement as a measure of uniform movement, where linear displacement is the distance an object has moved over a given time period, while angular displacement is the rotation of an object about an axis over a given time period.

celeration arises. One of the main forces acting on real-life objects that is a type of non-constant acceleration, is drag. Any object moving through any kind of medium with a density (i.e. not vacuum), experiences drag. The drag force is proportional to the velocity of the object squared, but in the opposite direction of the direction the object is moving. This means that drag $\mathbf{F}_d$ is a function of velocity $\mathbf{v}$. In simplified terms, $\mathbf{F}_d$ is defined as:

$$\mathbf{F}_d = f(\mathbf{v}) = -C\mathbf{v}^2 \tag{3.4}$$

where $C$ is a drag coefficient[3] of an object, which is reliant on empirical data determining the drag properties of the object. Also notice the negation of the statement above, which indicates that $\mathbf{F}_d$ is a force acting in the opposite direction of $\mathbf{v}$.

The sum of forces $\sum \mathbf{F} = -C\mathbf{v}^2$, means that $m\mathbf{a} = -C\mathbf{v}^2$. We stated earlier that $a = \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}$ and rewrite the equation to the following:

$$\begin{aligned} m\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} &= -C\mathbf{v}^2 \\ \mathrm{d}t &= m\frac{1}{-C\mathbf{v}^2}\mathrm{d}\mathbf{v} \end{aligned}$$

We can then integrate the left side from 0 to $t_1$ and the right side from $v_1$ to $v_2$:

$$\begin{aligned} \int_0^{t_1} \mathrm{d}t &= \frac{m}{C}\int_{v_1}^{v_2} -\frac{1}{\mathbf{v}^2}\mathrm{d}\mathbf{v} \\ t_1 - 0 &= \frac{m}{C}(\frac{1}{\mathbf{v}_2} - \frac{1}{\mathbf{v}_1}) \end{aligned}$$

We rewrite the equation to form an expression for $v_2$:

[3] $C$ can be calculated from a function dependent on the surface area and friction properties of the object and the medium in which the object is moving. In many cases, it is sufficient to keep $C$ constant after its initial calculation.

$$\begin{aligned}
t_1 &= \frac{m}{C}(\frac{1}{\mathbf{v}_2} - \frac{1}{\mathbf{v}_1}) \\
\frac{Ct_1}{m} &= \frac{1}{\mathbf{v}_2} - \frac{1}{\mathbf{v}_1} \\
\frac{1}{\mathbf{v}_2} &= \frac{Ct_1}{m} + \frac{1}{\mathbf{v}_1} \\
\mathbf{v}_2 &= \frac{1}{\frac{Ct_1}{m} + \frac{1}{\mathbf{v}_1}} \qquad (3.5)
\end{aligned}$$

Equation 3.5 is an expression for finding the velocity $\mathbf{v}$ of an object as a function of time, which is undefined if $\mathbf{v}_1 = 0$. This function in itself, is useless to represent the velocity as a function of time, as drag alone will not make an object move. To describe a moving object, we need to add a thrusting force of some sort, for example gravity $\mathbf{F}_g$:

$$\sum \mathbf{F} = m\mathbf{a} = \mathbf{F}_g - C\mathbf{v}^2 \qquad (3.6)$$

Equation 3.6 describes the vertical forces acting on a non-propelled, falling object. As the integration of equation 3.6 will be rather messy, we will not include it here.

## 3.3 Angular Motion

To allow object rotations, we need to extend the properties of the objects in the game simulation. Similar to the linear motion properties of $s$, $v$ and $a$, we have angular motion properties: $\mathbf{\Omega}$, $\omega$ and $\alpha$, each corresponding to the former respectively. $\mathbf{\Omega}$ is the angular displacement of an object in radians, $\omega$ is the angular velocity in radians per second, and $\alpha$ the angular acceleration in radians per second squared. The following equations show how these relate to each other:

$$\begin{aligned}
\Omega &= \frac{\mathrm{d}\omega}{\mathrm{d}t} \\
\omega &= \frac{\mathrm{d}\alpha}{\mathrm{d}t}
\end{aligned}$$

The corresponding equations for expressing these properties as a function of time, closely resemble the ones for linear motion:

$$\begin{aligned}\omega_2 &= \omega_1 + \alpha t & (3.7)\\ \Omega_2 &= \Omega_1 + \omega_1 t + \frac{1}{2}\alpha t^2 & (3.8)\end{aligned}$$

Where equation 3.7 is similar to 3.2 and equation 3.8 is similar to 3.3.

As with linear movement, angular acceleration of objects is affected by inertia. Where mass inertia is the term used for describing linear inertia, mass moment of inertia, or just moment of inertia, describes the angular inertia of an object. The moment of inertia of an object can be explained as the resistance the object has for changes in angular velocity in a given axis of rotation.

## 3.4 Collisions

The motion of objects moving around in a virtual world would not seem very realistic if the objects pass right through each other when they come into contact. Experience from real-life situations would suggest that some kind of a collision would occur when the exterior of two objects intersect. Collisions between game objects are a major concept in game physics. Either if the player interacts with the world through pushing or pulling objects or if other obstacles like a rockslide appear in front of the player, the physics engine would have to calculate whether a collision has happened or not between the objects involved. If so, the physics engine must calculate new trajectories and velocities of the colliding objects.

Collision detection is necessary to determine if objects actually have collided. The collision detection mechanism can become a complex routine in advanced physics engines. Objects of irregular size and shape must have their vertexes and edges calculated to find intersections with the vertexes and edges of other objects. Even for more advanced objects, collisions between parts of the object itself might be possible.

Once a collision has been detected, the engine applies an impulse to both of the objects. An impulse, by definition, is the integral of a force acting upon an object over time, changing the momentum of that object.

## 3.5 Existing Physics Engines

Because of the fact that physics incorporated into games is becoming more and more popular, several physics engines and frameworks have emerged over the last years. Some are open source and available for all to use and change, while others are released on more limited licenses. Physics engines vary in complexity, ranging from simple two-dimensional engines, to advanced off-line three-dimensional physics simulators used in scientific simulations. Most of the mainstream available physics engines are designed for games. A requirement for a physics engine designed for games, is that it must be able to compute all physics operations in real-time. This means that physics engines used in games are not as accurate as physics engines used for scientific simulations, but their accuracy is sufficient to provide a realistic gaming environment.

**Box2D [33]** Small two-dimensional, open source physics simulation engine. It is a rigid body simulator written entirely in C++. It has support for object collisions, joints between objects, friction and restitution forces. It also has an integrated testbed for unit-testing and demo environment.

**Bullet [34]** A collision detection, soft body and rigid body dynamics multithreaded physics library released under the MIT license. It is integrated into the free 3D-modeling software Blender 3D [35]. It has optimized back-ends with multi-threaded support for Playstation 3 Cell SPU, CUDA, OpenCL and other platforms.

**Open Dynamics Engine (ODE) [36]** Open source, high performance library for simulating rigid body dynamics. Aiming at platform independence with a C/C++ API. Modified versions of ODE have been used in several high budget games, including S.T.A.L.K.E.R and Call of Juarez [37].

**PhysX [31]** Proprietary physics library owned by NVIDIA. Supports several platforms, including PC and all current gaming consoles. There is also an existing implementation for GPU. Main properties: complex rigid body physics system, character control, multi-threaded and soft body simulation.

**Havok Physics [38]** Extensive proprietary physics library which has been put into mainstream use in many high budget games. It supports integration with major 3D modeling tools along with most other necessary features of a physics engine. It also has a performance tuning and testing suite and is optimized for several modern gaming consoles.

Because of the complexity of all the previously mentioned physics engines, we have decided to implement our own simplified physics engine. Our engine uses the same basic concepts as a normal engine, but we have reduced the complexity by removing features that might complicate the cheat detection mechanism beyond the scope of this thesis. We will explain what we have decided to include in the next section.

## 3.6 Physics used in this thesis

In this thesis, we have only included a few of the many concepts possible of adding to a physics engine. Physics engines are complicated simulators, and creating a cheat detection mechanism for one of the above mentioned physics engines would mean all elements of the engine have to be considered in the mechanism. This would result in a very advanced mechanism, beyond the scope of this thesis. One of the challenges for game developers, is finding a compromise between realism and computational resources, as they are proportional to each other. Increased realism requires more advanced models, which again require more resources. Very realistic models of physical environments, can not be processed in real-time, and are suited for high-precision off-line simulations. Games require real-time physics, so they must have degraded realism.

Our simulation takes place in a space-like environment, near a gravitational field. The objects are not affected by drag, but they experience both linear and angular acceleration. There is a constant gravitational pull, affecting the objects. All the other forces are generated by the objects themselves. These forces are generated from thrusters. Figure 3.1 shows an outline of a game object, with the main rear thruster and the bow thrusters. For simplicity, every object is identical with regard to shape. Size and thruster power can be modified by parameters, to allow objects to behave differently from each other.

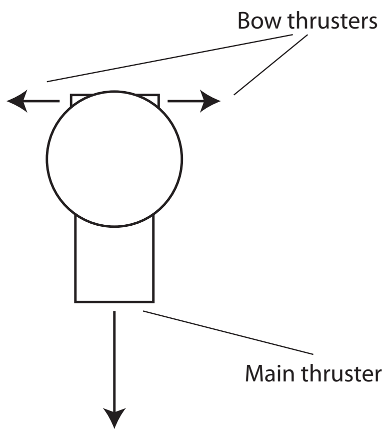

Figure 3.1: *Illustration of a game object with bow thrusters in the front and the main thruster at the back. Arrows indicate the direction of the thrust force from the engines.*

To begin with, the physics engine of the system used just linear motion. Objects within the simulation could not rotate, but they could move in arbitrary directions, based on the notion that objects had an infinite number of thrusters around a spherical shell. This way, they could apply thrust in any direction. Torque from the thrusters was not calculated, another lacking feature affecting realism. After initial testing with linear motion physics, some angular effects were added. Objects now move with a main rear thruster and rotate around by using bow thrusters.

## 3.7 Summary

In this chapter, we have discussed much of the theoretical material behind the physical model of our game simulation. We mentioned some essential functions in Newtonian mechanics that apply to almost any physics engine. We also commented on other topics often seen in physics engines. Finally, we gave information about what of the previously mentioned material we have used in our simplified physical model. Physics has been proved to be be relatively easy to parallelize, and we will investigate hardware alternatives for running parallel physical calculations in the next chapter.

# Chapter 4

# Multi-Core Architectures, Graphics Processing Units and GPGPU Frameworks

We have already seen in the previous chapters that cheat detection may require a considerable amount of processing power, and scaling up this process to support a large number of clients is an important goal. In this chapter, we begin therefore by investigating several multi-core architectures that might serve as the underlying hardware of our implementation. Multi-core architectures provide a foundation for parallel processing, because multiple concurrent threads are made possible by the hardware, not just simulated in software. Many different architectures fall under the category multi-core, because it is such a broad term. Most modern commodity CPUs, like the Core 2 Duo from Intel, are multi-core processors. So is the Cell Broadband Engine from the STI alliance[1].

The GPU is also a multi-core processor. Although the GPU is designed for graphics computation, the advent of GPGPU development has enabled programmers to harvest the computational power of the GPU for general purpose tasks. We want to determine how cheat detection mechanisms can be run on a GPU with GPGPU. In particular, we will research NVIDIA's line of GPUs, with the main focus on the GeForce GT200 processor, the latest generation of NVIDIA GPUs. We continue with a brief description of the GPGPU concept, and techniques used for GPGPU development. We investigate further a particular GPGPU framework; CUDA from NVIDIA. The CUDA framework is under constant development and relatively well documented. It has an active developer community, with an on-line forum where developers help solve each others problems and

[1]STI is an inter-corporation alliance between Sony, Toshiba and IBM.

share experiences [39]. These are amongst the reason we have chosen the GPUs from NVIDIA as the hardware used in this thesis.

## 4.1 Introduction

Since the first transistor based CPUs were introduced, the development of micro- processors have followed a certain trend. The transistor count of a single processor has increased in an exponential fashion, roughly doubling every two years [40, 41]. This trend is often referred to as Moore's law. Along with the increased number of transistors came augmented computational power, and thus a higher power consumption. This again, increased the power density on the chip, and the expansion of the processing units began to reach a limit, because of too much heat being generated by the chip. Probably the most well known product to reach this "heat-wall", was the Pentium 4 from Intel. To work around the heat problem, hardware manufacturers have begun to incorporate several individual processor cores in architectures, to improve the computational power of modern processors, but with less power consumption than a single-core processor.

Another limitation constraining the performance of single-core processors is the "memory-wall". Processors have increased in speed faster than memory latencies can be reduced. The distance between main memory and the processor introduces memory latencies that slow down single-threaded applications, and is a physical limitation that cannot be bypassed. Increasing cache sizes can only hide some of the latency. Inevitably, single-threaded execution will be bound by the latencies of the memory. Multi-threaded execution allows us to hide these latencies, as several operations are running in parallel. These operations can then overlap, so that operations ready for execution run while others wait for memory accesses. Throughput is increased, but the execution time of a single thread is not improved. Problems that can not be solved with parallel execution will for this reason be a bottleneck in a parallel system.

## 4.2 Multi-Core Architectures

Multi-core architectures have multiple processing units packaged into a single integrated circuit, often referred to as die[2]. A multi-core processor can also consist of several dies bundled together onto a single circuit board. Due to the vast nature of multi-core architectures, they are classified in two main categories, homogeneous and heterogeneous

[2]Intel's *Core 2 Duo* [42] is an example of such an processor.

multi-core architectures. Multi-core architectures have been around for quite some time, and early systems mostly used Symmetric Multiprocessing (SMP). SMP is a homogeneous multi-core technology where several identical processing units exist and cooperate in the same system architecture. The processing units share the same main memory, but they may be independent of each other. Modern multi-core CPUs can also be viewed as an SMP system, where the identical processing units are the separate cores of the chip. SMP systems suffer from the fact that scalability is limited. Systems can rarely have more than 16 cores before the benefit of adding more cores stagnates, due to main memory accesses being serialized. Cache coherency also becomes troublesome with added cores, reducing efficiency as integrity is harder to maintain.

### 4.2.1 Homogeneous Multi-Core Architectures

Homogeneous multi-core architectures are architectures where all processing units are symmetric. This means that all of the processing units are identical with regards to *instruction set*, *registers*, *local cache* and the *clock frequency* of the unit (see figure 4.1). Most modern multi-core, commodity CPUs have identical cores and are classified as homogeneous. The modern multi-core CPU normally has the processing units on the same die. Intel's dual-core Intel's Core 2 Duo [42] processor is an example of such a homogeneous architecture. A system with several identical single-core CPUs placed in several sockets on the same motherboard would also be classified as a homogeneous system. This was a solution used in personal computers before the release of multi-core CPUs and also common in SMP systems.

### 4.2.2 Heterogeneous Multi-Core Architectures

A heterogeneous multi-core architecture is a system where at least one processing unit differs from the other units in either of the points stated under section 4.2.1: instruction set, registers, local cache or the clock frequency. Figure 4.2 illustrates an example of a heterogeneous processor design. It is quite common that only the main processor has access to main memory, while the co-processors have their own memory. Communication and memory transactions happen over a bus. A legacy example of a heterogeneous architecture was the correlation between the Intel 8088/8086 processors [40, 41] and the Intel 8087 floating point co-processor. The 8086/8088 was able to pair up with the 8087 to offload floating point operations.

A more recent example in commodity systems, is the relationship between the CPU and

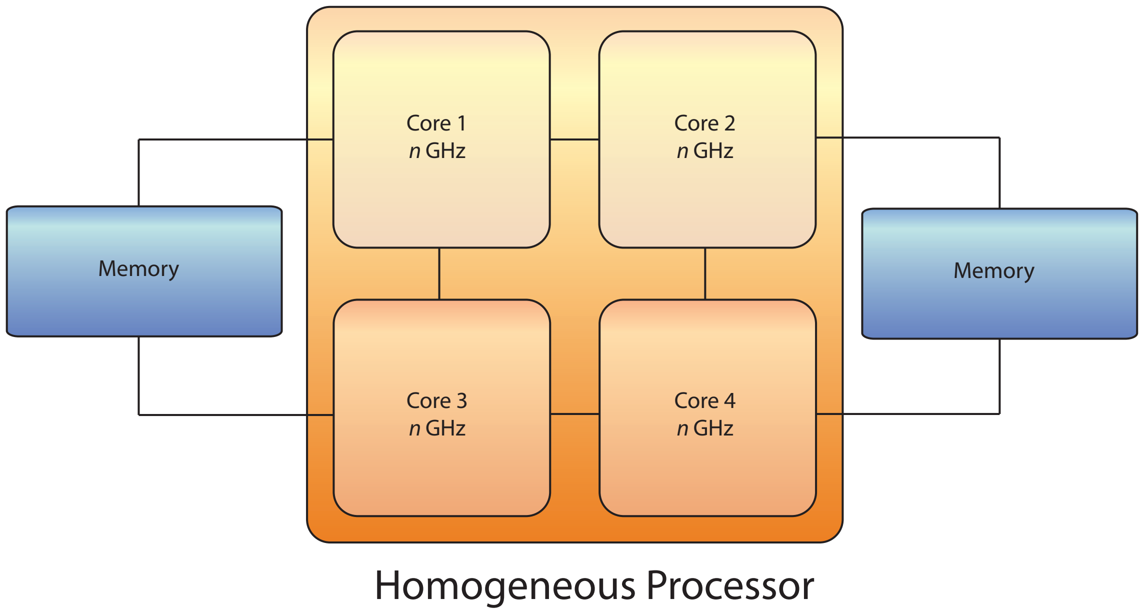

Figure 4.1: *Sketch of a homogeneous processor with four identical cores. All cores have access to the same main memory.*

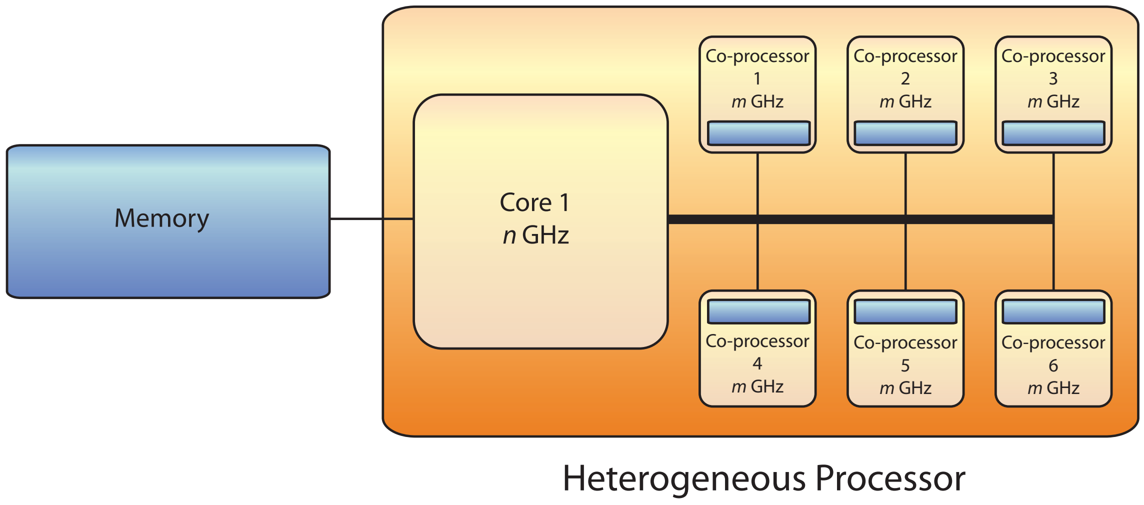

Figure 4.2: *An example illustration of a heterogeneous processor with a main processor and six co-processors connected through a bus.*

the GPU. Many modern computers have a GPU responsible for offloading graphical computations from the CPU. This way, the CPU is able to assign tasks, concerning graphics to the GPU, which then performs calculations and displays the results to the system monitor. The GPU is a highly parallel processor, with a large number of simple processing cores.

Another heterogeneous CPU architecture is the Cell Broadband Engine (CBE) [43], developed as an inter-corporation project by the STI alliance. The CBE is currently being used as the central processor in Sony PlayStation 3 game console. It is also available in IBM blade centers. A cut-down version of the Cell processor is also used as a co-processor in several Toshiba laptops, and as a processor to upscale DVD video. The CBE consists of a main processing unit from the Power architecture, and eight smaller, vector processing cores. The vector processing cores are designed to accelerate multimedia applications, and other types of dedicated computation. Figure 4.2 is based very loosely on the way the CBE is constructed.

We can also witness heterogeneity in newer homogeneous multi-core processors like the Core i7[3] from Intel [44]. The processor allows for independent clocking of the individual cores. Generally, with multi-core processors when all cores are loaded, the chip will have a maximum power consumption and a given heat dissipation. When one or more cores are idle and the remaining are running at a constant clock frequency, they will not have the same power consumption and heat dissipation as the fully occupied chip, leaving power headroom for overclocking. Since the individual cores of the Core i7 can be clocked to higher clock frequencies, the chip can achieve up to the same power consumption and heat generation as when all cores are loaded, allowing the processor to perform closer to its peak performance even when cores are idle [45]. This feature is largely made possible by the fact that the Nehalem architecture allows for idle cores with a power consumption close to zero watts.

## 4.3 Graphics Processing Units

A GPU is a co-processor alongside the CPU. It is found in most commodity personal computers, workstations and game consoles. The GPU has served the purpose of offloading visual data computations from the CPU. The GPU is designed to handle graphic specific calculations, which includes floating point arithmetic and computations on primitive geometric objects, like triangles, rectangles and circles. Given the highly parallel nature of graphics rendering, the GPU has a large amount of transistors devoted to data processing, rather than data caching and flow control. Figure 4.3 shows this logical difference between a CPU and a GPU. The GPU has a significantly larger number of arithmetic units, and a considerably smaller number of transistors are devoted to caching and control. Rough estimates by NVIDIA show that about 20% of transistors are devoted to computational tasks on a CPU against 80% on a GPU [46].

[3]Core i7 is of the Nehalem architecture and the successor to the Core 2 architecture.

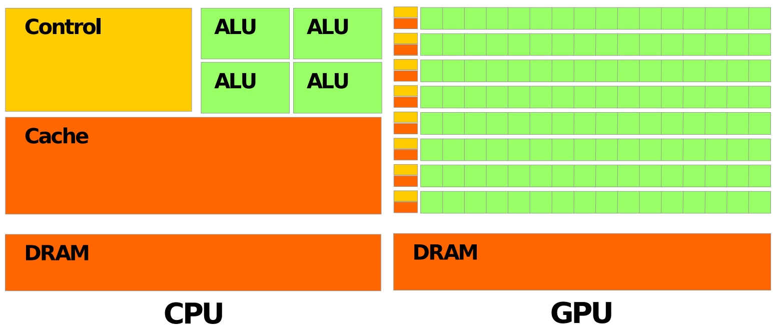

Figure 4.3: *Simple overview of a CPU and a GPU [3]*

### 4.3.1 The Graphics Pipeline

General graphics computations are structured the same way on all GPUs through the use of a graphics pipeline (figure 4.4). The main purpose of the pipeline is for the GPU to maintain high computation rates through parallel execution. Each stage in the pipeline is designed as a dedicated piece of hardware on the chip. In previous generation graphics hardware, the pipeline served as a fixed-function unit. Through the evolution of both APIs and hardware, the pipeline has become more flexible, and in modern hardware, both the vertex and fragment units are programmable using shader languages like OpenGL [47] and DirectX [48]. The graphics pipeline is described in further detail in [49].

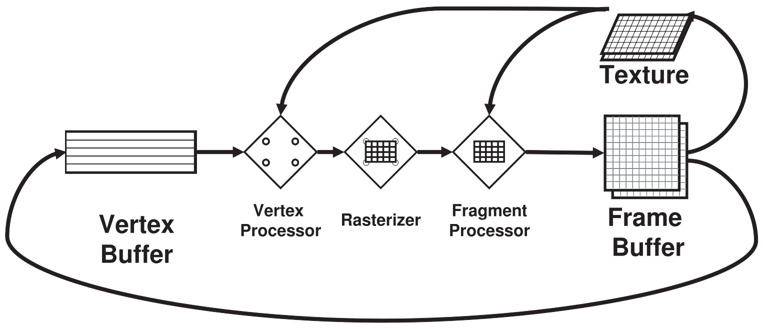

Figure 4.4: *The basic graphics pipeline. Taken from [4].*

The main purpose of the GPU is to generate graphic renderings that are as realistic as possible. Since the need for a more flexible pipeline arose, graphics vendors have added more customizable and programmable features to the pipeline, with each chip generation. This increased flexibility has allowed for general- purpose programming of graphics hardware, exposing the immense processing potential the devices are capable of. Though very complicated with the use of only shader languages in the beginning, the modern GPUs from the leading vendors have turned into fully programmable co-processors with their own Software Development Kit (SDK) [50, 51].

### 4.3.2 A Unified Graphics Processing Unit Design

In late 2006 and through 2007, GPU vendors started introducing unified shader architectures. The releases introduced a total redesign of the graphics pipeline, DirectX 10 [48] support, and powerful GPU physics and high-end floating-point computation ability. The most revolutionary feature was the graphics pipeline redesign, which was given a unified shader design. This resulted in making the original, sequential flow through discrete shader types more looping oriented, with a shader processing core handling all shader stages (illustrated by figure 4.5).

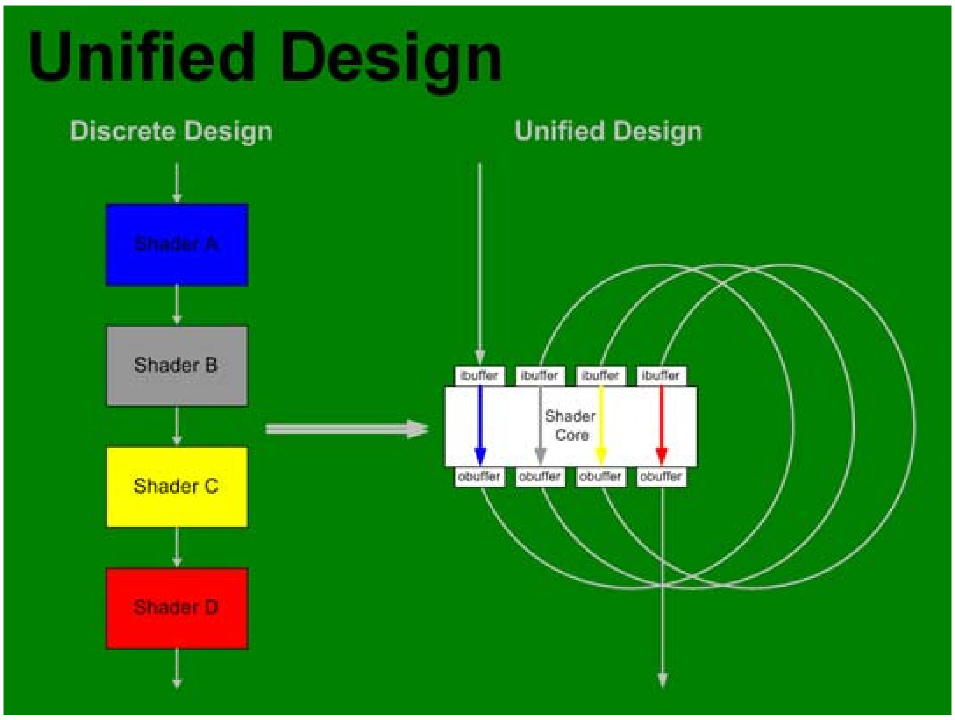

Figure 4.5: *Classic vs. unified shader architecture [5].*

The new unified architecture minimizes idle hardware, because a shader processing core can work both as a vertex, geometry and pixel shader depending on the need [5]. Another feature, made possible by the unified design, was the introduction of APIs simplifying GPGPU. Most notably CUDA from NVIDIA and Close to Metal from AMD. Close to Metal is now obsolete and replaced by FireStream.

### 4.3.3 Overview of the NVIDIA GeForce GT200 architecture

In this section, we will describe the general structure of the GPUs of the unified graphics and computing architecture[4], with basis in the specific architecture of the latest generation of GPUs from NVIDIA, the GT200 processor. Most of the general information given, will also apply to the NVIDIA Tesla [52] line of products. Tesla products are in almost every way regular GPUs, except that they are without the ability to output graphics to a display. They are aimed towards aiding in simulation and other large scale calculations for the professional and scientific fields.

NVIDIA unveiled the GeForce 8800 architecture, codenamed G80, in November 2006 [5]. The release of the G80 graphics processor introduced CUDA, and with it, GPUs able to perform non-graphics specific tasks with considerable less complexity compared to the earlier generations. NVIDIA released the GT200 graphics processor in June 2008, which is referred to by NVIDIA as the second generation of their unified graphics design [46]. Figure 4.6 shows an image of a GT200 processor die with overlays illustrating the different parts of the chip, and its general structure. Several design goals were set for the new GPU, aiming for a doubling of the performance, relative to the previous generation, increased power efficiency, better DirectX 10 support and a significant enhancement of the computation ability for high-performance CUDA applications.

Figure 4.7 gives a logical overview of the GT200 chip, and its features. The GT200 consists of Texture Processing Clusters (TPCs), a global block scheduler, memory controllers, and L2 texture cache in front of the frame buffer (main device memory). The memory controller has also an atomic unit, which is a unit responsible for handling atomic read and write operations to memory. The GPU has a total of ten TPCs, while older generations had up to eight. The TPC is the topmost grouping of the computing units of the architecture; the other units mentioned are basically memory and control units. A TPC has a Texture Unit and three Stream Multiprocessors (SMs)[5]. The TPC also has a L1 cache, several texture filtering units and texture address units.

An SM is divided into several components, most notably its eight Stream Processors (SPs) Cores, which are the main processing units of the GPU. Additionally, it has two Special Function Units for internal functions and complex mathematical operations, and on-chip shared memory. With ten TPCs, each with three SMs, which again has eight SPs each, results in a total of 240 processing cores for the GT200. An SM is responsible for mapping software threads to single hardware threads to each of its SPs. The SM handles thread

[4]This includes all NVIDIA GPUs from the 8th generation and onwards

[5]The older models have two SMs per TPC.

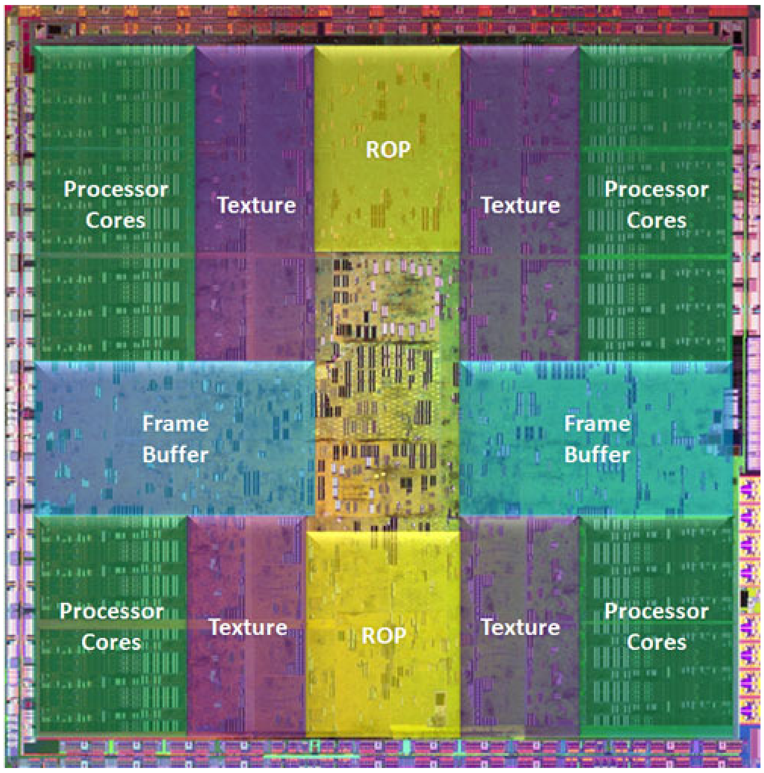

Figure 4.6: *Die diagram with overlay of the GT200 processor core*

scheduling and also fetches the instructions to be performed on each SP. The GT200 are able to run a total of up to 30,720 threads per chip, indicating extreme parallelism possibilities. These devices have several different types of memory. On-chip memory includes local registers and shared memory; while the larger Dynamic Random Access Memory (DRAM), or frame buffer, resides off-chip. The size of the frame buffer depends on the card manufacturer. New to the GT200 chip is also double-precision, 64-bit floating point computation support. Each SM incorporates a double-precision floating-point math unit, this means the GT200 chips have 30 double-precision processing cores. The floating-point math unit shares registers with the eight SPs. While the SPs can run eight threads simultaneously, the floating-point unit can only run one at a time, resulting in slower execution of double precision operations.

### 4.3.4 General-Purpose Computation Using Graphics Hardware

The original task for the GPU was to offload the CPU from computationally heavy graphical processing. From the original fixed-function graphics pipelines, the new generations of GPUs have given increasingly amounts of flexibility concerning the different stages in the graphics pipeline [4]. Several APIs have been designed for programming what the GPU

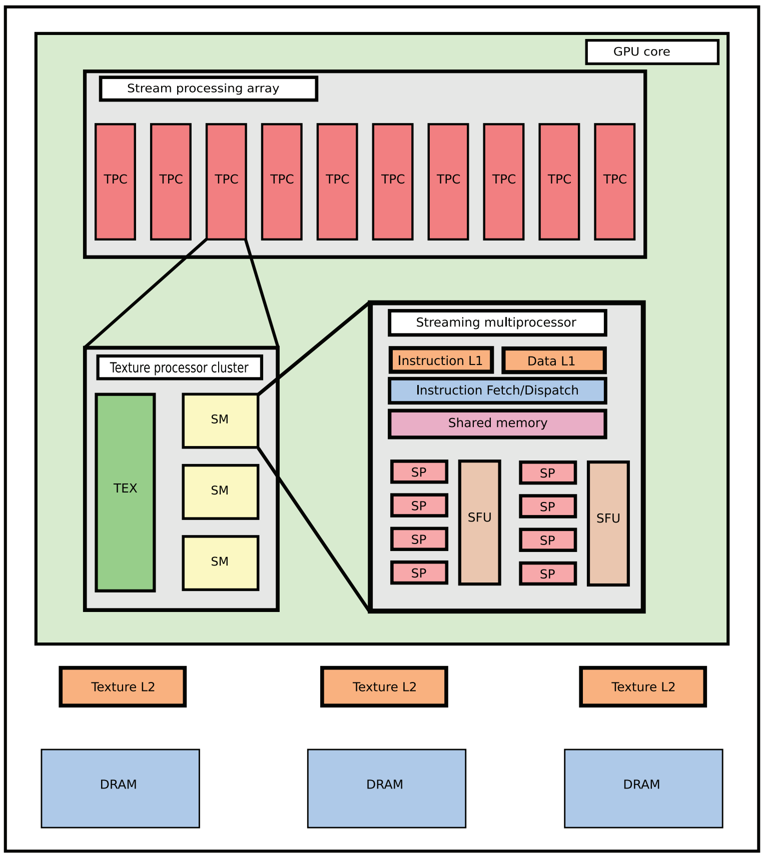

Figure 4.7: *Overview of the GT200 [6]*

is going to process to produce rasterized 2D images. These APIs mainly consider vertex data that is processed by the graphics pipeline to produce the desired graphical output. The two main APIs for graphical programming, OpenGL and DirectX, have already been mentioned.

The performance of GPU hardware, measured in GFLOPS[6], is roughly doubling every six months [3,4]. The discrepancy between the development of the GPU and that of the CPU is caused by the architectural differences between the two. Because of the large number of processing units available, the GPU excels when dealing with problems that can be expressed as parallel computations with high arithmetic intensity - the ration between arithmetic and memory operations.

Development in the field of GPU has led to better utilization of the processing power of the GPU with the advent of GPGPU abilities. GPGPU was originally made possible by exploiting shader languages. However, the programmer was forced to view the problem in terms of geometric primitives and fetching textures, which rarely maps easily to the

[6]Giga Floating-Point Operations Per Second. Often a measurement used for performance comparisons between different processors.

nature of the problem being addressed. Because of this, GPGPU programming was in many cases more complicated than it needed to be. To simplify GPGPU development, GPU producers begun to develop new technology to narrow the gap between shader languages for the GPU, and programming standards for CPUs.

We have already named CUDA from NVIDIA as a GPGPU framework. AMD with the brand name ATI is another major vendor of GPU hardware who has released an API for GPGPU development. Firestream is the name of their architecture and SDK [53]. Firestream uses AMD's own implementation of the open source C/C++ compiler Brook, called Brook+. The AMD Compute Abstraction Layer provides direct communication with the device, allowing programmers to avoid learning graphics-specific languages. The Compute Abstraction Layer also provides low-level access to code, which opens for fine-tuning device performance. The SDK also aims to provide a math performance library (AMD Core Math Library), which is currently under development [54]. Additional tools allow for profiling and debugging of GPU code, to simplify software development.

OpenCL [55] is another API designed for GPGPU development. It is under development by the Khronos group in collaboration with many industry-leading companies, including NVIDIA and AMD [55]. The goal of OpenCL is to support a wide range of applications through a low-level, high-performance, portable abstraction. OpenCL is not restricted to GPGPU, but will ease development on most multi-core architectures. Other advantages with OpenCL is that it will be the first open, royalty-free standard for this kind of development. OpenCL will support portable code designed to effectively utilize the processing power of multi-core architectures. In August 2008, AMD announced that they will support OpenCL and aid the effort of developing OpenCL into an open standard [56]. NVIDIA has also announced that they will fully support the OpenCL 1.0 specification alongside CUDA, on all its upcoming platforms [57].

We have chosen CUDA and NVIDIA GPUs as the underlying architecture for this thesis. We find CUDA to have the best documented API and also to be a well received API in the scientific field. The remaining part of this chapter will focus on CUDA.

## 4.4 Compute Unified Device Architecture (CUDA)

We have discussed several different types of multi-core architectures in the first part of this chapter. We gave a more in-depth description of the newer generation of GPUs from NVIDIA supporting CUDA. We dedicate the rest of this chapter to the CUDA framework, which is intended for GPGPU application development.

A CUDA application is created from mainly two parts, one for the main CPU and one for the GPU. The GPU that runs the CUDA specific parts of an application will be referred to as the CUDA device, or just the device, from this point on. Such hardware devices mainly include the newer NVIDIA GPUs described earlier and the NVIDIA Tesla [52] products. The CPU that interacts with the CUDA device will simply be referred to as the host.

Code written for the host is like any other CPU-specific code, and the programmer has access to all libraries he or she would normally have access to. The device-specific code is the code intended to run on the CUDA device. How CUDA applications are compiled and run will be discussed further in section 4.5.

### 4.4.1 Programming Model

The CUDA programming model is an extension to the ANSI C programming language. By supplying a familiar development environment, starting out with GPGPU programming is considerably easier. CUDA also allows developers to program in a similar way to what they are used to, instead of having to think in graphics API terms.

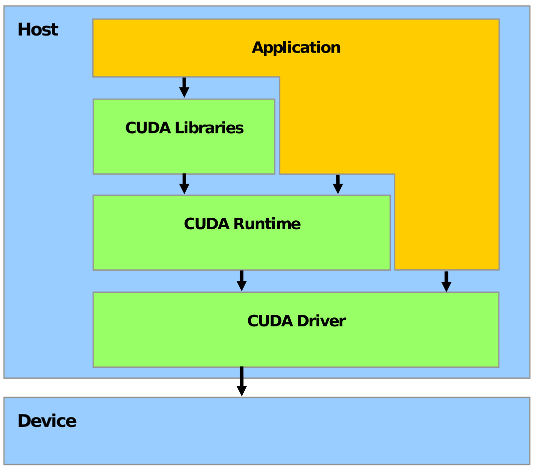

Figure 4.8: *The CUDA software stack [3]*

Figure 4.8 illustrates the layers in the software stack of CUDA. CUDA applications rely on a device driver. On top of the driver, you find the CUDA runtime environment and then the dynamic libraries. A CUDA developer has access to every layer of the software stack through individual APIs. The driver and runtime API expose similar functionality, but the driver API gives somewhat more control to the developer while the runtime API reduces complexity of several operations. The two APIs are mutually exclusive, meaning once the decision to use on of them has been made, the application is restricted to use that

API only. The dynamic library consists of several convenience functions and macros to ease development and increase stability of CUDA applications. The library also consists of standard structures used in 3D based applications such as matrices and quaternions. All CUDA code used in and developed for this thesis use the runtime API.

### 4.4.2 Kernels

A CUDA application can not be run entirely on a CUDA device, it is dependent on the host CPU. All interaction with the device is done through the device driver, and the host is responsible for all interaction with this driver. The device only receives device-specific instructions on what operations to perform and executes these operations. The portion of the code written for execution on the CUDA device is referred to as a kernel.

Kernels define the behavior of the smallest logical structure in CUDA, the thread. The CUDA device is able to run thousands of concurrent threads, each executing the code specified in the kernel. This implies that the device executes the same kernel code over a large number of software threads, which allows for a highly parallel programming model. The number of threads that should execute the kernel on the device is specified by the programmer as a hierarchy of threads, described in section 4.4.3.

CUDA threads can not access the main memory of the system, so the host must copy the proper data to the memory of the device before a kernel launch. For the same reason, all resulting data must be fetched from the device by the host after the completion of the kernel execution. We will take a closer look at the CUDA memory model in section 4.4.5.

CUDA kernel code focuses mainly on the elements related to serving the application purpose, or a part of the application purpose. Since the host handles synchronization and interaction between itself and the device including inter-device memory operations, the kernel code should mainly consist of algorithm-specific operations[7], and as few memory operations as possible.

When a kernel is invoked from the host, control is returned to the host as soon as the device begins executing the kernel. The host is able to continue executing code while the device runs the kernel. This asynchronous launch reduces the amount of time the host is idle while the device executes a kernel. Synchronization calls are available that blocks the host until the device is finished executing.

[7]Operations with a high arithmetic intensity, meaning they are not memory-bound.

### 4.4.3 Threads, Blocks and Grids

An abstraction defining a thread hierarchy specifies how many threads will be run when invoking a kernel. Figure 4.9 illustrates thread organization in blocks and grids. Threads are grouped together into a unit called a block. The threads in a block will run on the same SM, and the SM is responsible for mapping one software thread to one of its SPs. The thread block can be arranged in up to three logical dimensions. The identification of a thread is calculated from the x, y and z components of the thread's position in the block. Several thread blocks can again be combined in a logical grouping called a grid. The grid only supports one and two dimensions of blocks. In each kernel execution there can only be one grid, but with little restraint on the number of blocks in the grid.

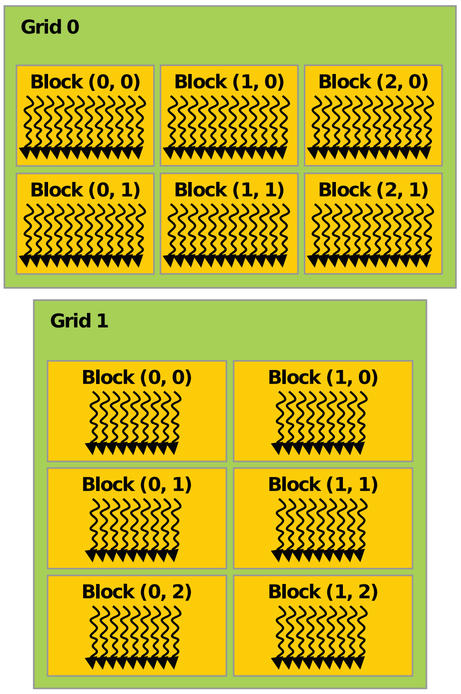

Figure 4.9: *Example of CUDA thread organization*

Since a block is executed on one SM, all threads within the block have access to the same shared memory. This memory is not shared between different blocks, communication between these, will have to go through the global device memory. The programmer has no control over the scheduling of the blocks within the grid. Furthermore, the block execution order should be perceived as random, as there is no guarantee of the order of which blocks are scheduled to run. Sequential dependencies will severely cripple the application design, and should if possible, be avoided. This is a common problem when adapting sequential algorithms to parallel algorithms.

There is absolutely no performance benefit of multiple threads computing the exact same thing and placing the same calculations in the same place, therefore it is important for each thread to know where to find input data and where to place output data, specific to that one thread. To achieve this, each thread will have to have, and know, a thread id. As mentioned, CUDA allows threads to be arranged in up to three logical dimensions in each thread block, and blocks in two dimensions in the grid. For the kernel, a couple of implicit variables are available to distinguish which thread should access which part of the memory.

### 4.4.4 Single-Instruction, Multiple-Threads (SIMT)

In CUDA, when every thread follows the same kernel execution path, the execution will have similar behavior to regular Single-Instruction, Multiple-Data (SIMD) operations. SIMD operations are exactly what the name indicates; an instruction performed over multiple data in only one operation. Whenever one or more threads take a different execution path when evaluating a conditional statement the behavior of CUDA differs from regular SIMD. Threads in CUDA are assigned to an executional unit called a warp. A warp is defined as a set of threads assigned to a SM. Figure 4.10 illustrates how warps of threads belonging to different blocks can be assigned to an SM. The warp consists of 32 concurrent threads belonging to the same block. The threads that make up the warp begin at the same starting address in the code, but are from there on free to branch in different directions and execute independently from each other. When threads of a warp diverge on a branch, they will form two warps, one for each alternative of the branch. The term half-warp is defined as either the first or second half of a warp.

The reason for the existence of warps is that each SM can only execute one instruction for all its SPs at a time. Since each SP handles one instruction for only one thread at a time, all the SPs must handle threads that share a common instruction, which would be threads within the same warp. Branching of code results in threads following different paths of execution. The SM serializes execution of threads diverging on a branch, reducing performance. Whenever the threads in a warp that diverged on a conditional statement return to the same path of execution, they are joined to form one warp again so that all the threads can continue to execute in parallel. Because of this possible deviation between different threads and the usage of different warps we state that CUDA is not a SIMD architecture, but a SIMT architecture.

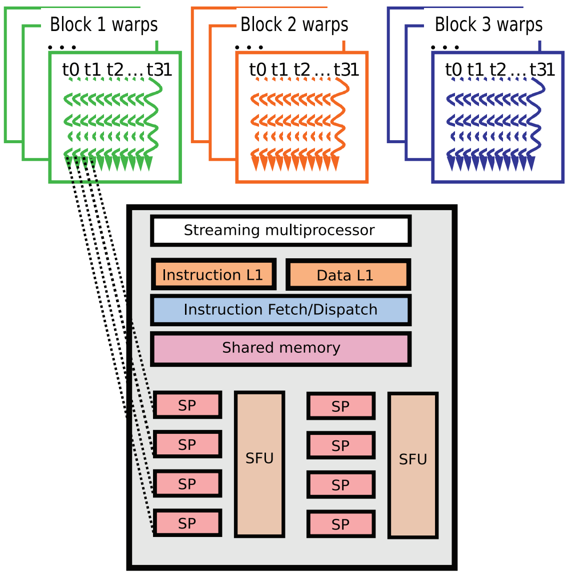

Figure 4.10: *Warps of threads assigned to an SM [6].*

### 4.4.5 Memory Hierarchy

There are several types of memory on the GPU that each have unique properties. Private to a single thread, are registers and local memory. Registers are fast accessible on-chip memory, while local memory is off-chip DRAM and considerably slower. If a thread needs to store more local data than what is possible in registers, it will store it in local memory.

**Shared memory** is an on-chip memory, which means it is very fast. Shared memory can in fact, when there are no bank conflicts, be accessed just as fast as local registers [3]. Shared memory is also shared between all the threads in a block. This memory is used for communication between threads of a single block. It also offers much faster access times than local memory. As mentioned above, memory which does not fit into local registers will be placed in the local memory. Shared memory should therefore be used for as much local storage as possible.

Global memory is the main portion of off-chip DRAM. The DRAM is divided into three different types of memory: *global*, *constant* and *texture* memory. The name *global memory* is a bit deceiving, as it can refer to either the DRAM entirely, or just the part of DRAM which is not constant or texture memory. We will refer to the off-chip DRAM as DRAM only, to avoid confusion, and global memory is the remaining part of DRAM not classified as any of the other memory types residing in DRAM.

**Global memory** is the only writable part of the DRAM, as constant and texture memory are read-only. Global memory reads and writes are equally slow as the thread local memory, but its advantage is that its accessible for every thread running on the device.

It is also persistent over consecutive kernel launches, meaning threads in a kernel launch might leave data in global memory for a subsequent kernel. The host does not have access to thread registers, local memory or shared memory, so global memory is the only way for threads to leave resulting data for the host. Because of the high costs of global memory access, it is important to follow the right access patterns to get maximum memory bandwidth.

**Constant and texture memory** is read-only, as previously stated. The host is responsible for classifying parts of DRAM as either constant or texture memory and filling it with data. The advantages of constant and texture memory is that each SM has its own cache for these two memory types, which results in some performance increase. This cache, however, is not very large, so the increased performance might be limited when using scattered, random memory accesses. Texture memory has another advantage over the other DRAM memory types, as it allows for multi-dimensional fetches. A texture fetch can broadcast packed data into several separate variables in a single operation. It also supports data type conversions during fetches (8- and 16-bit integer input data converted to 32-bit floating-point values). Texture fetches are also not affected by the constraints of memory access patterns that applies to global and constant memory accesses.

## 4.5 CUDA Compiler and Programming tools

In section 4.4, we introduced the programmatic features of CUDA. In this section, we will look closer at how CUDA programs are compiled and some tools available to ease development. CUDA requires its own compiler, but it also makes use of the native compiler for C/C++ of the development system. There is also an assembly-like language which all kernel code is compiled to. We will also briefly mention the available debugger, profiler and the emulation mode.

### 4.5.1 NVCC: The CUDA Compiler

CUDA code is compiled with the CUDA compiler, called NVCC. The common file extension given the code files of CUDA programs is *.cu*. A file containing CUDA code might contain both host- and device-specific code. CUDA programs are written with certain function type qualifiers that notifies NVCC whether the function should execute on the host or the device. Code written for the host is passed to the native C/C++ compiler on the system while the code written to execute on the device is handled by NVCC.

Figure 4.11 illustrates the way CUDA applications are compiled.

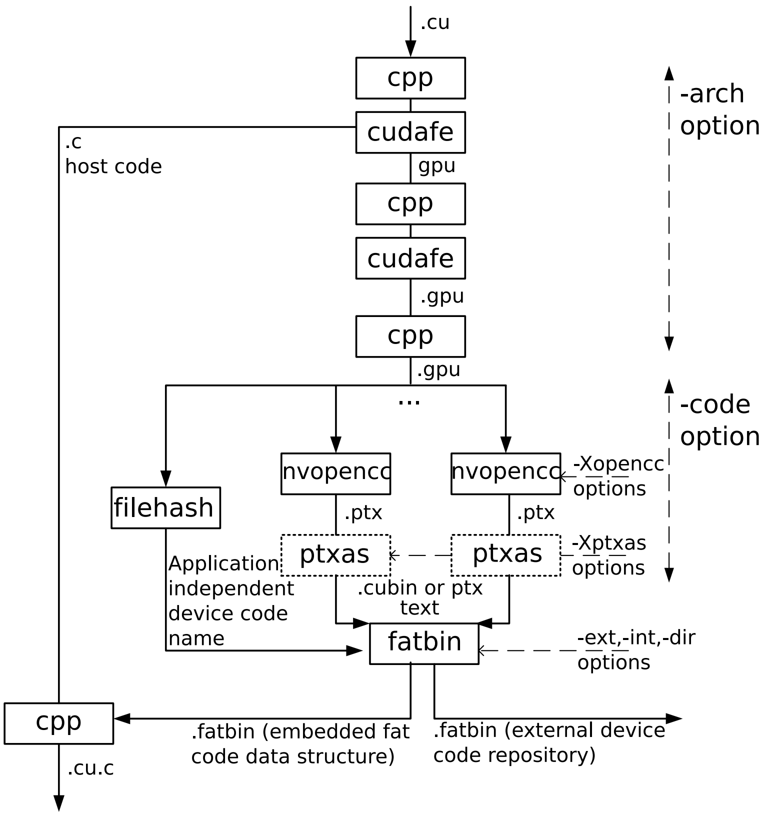

Figure 4.11: *The compiler flow of NVCC [7]*

The front end of NVCC (CUDAFE), separates the host and device code. The host code is passed on to the native compiler of the system, while the device code makes another pass through CUDAFE. Different compiler options will specify the direction of the compilation steps. At the end of compilation, the device code can be embedded in the host binary or be an external resource.

The device code is not GPU machine code, but rather an intermediate object code, similar to code read by Virtual Machines (VMs). This shader assembly code is converted at runtime into GPU-specific machine instructions by the GPU driver, using a Just-In-Time (JIT) compiler. The reason for the JIT compiler is to allow CUDA applications to be compatible with several generations of CUDA supporting devices. The JIT compiler allows the run-time environment to optimize the CUDA object code for the device on the system where the application will be executed. This object code is what is embedded in the final host binary.

### 4.5.2 Additional tools for CUDA development

There are several existing tools created by NVIDIA that aims to simplify and enhance CUDA applications and aid in application debugging. The debugger for use with CUDA applications is called CUDA-GDB [58]. Its purpose is to provide developers with ways to debug their applications directly on the hardware. With simulation and emulation environments, correction variations from the hardware are often introduced. CUDA-GDB is a ported version of GDB: The GNU Debugger, version 6.6 [58]. The debugger allows for a unified environment for application debugging, both for device and native host code. The debugger is currently only in beta version.

There is also a profiler available that allows the developer to investigate the efficiency of the device specific code. The profiler find the occupancy of the device during code execution, as well as timing of kernel launches and memory operations.

## 4.6 Summary

In this chapter we have discussed several different multi-core architectures beginning with homogeneous and heterogeneous architectures. We then gave a more in-depth description of a heterogeneous system, the GPU. With focus on NVIDIA and their line of modern GPUs we introduced some properties of the processors. We have given a brief introduction of several frameworks for development of GPGPU applications and described CUDA in further detail, a framework and API for GPGPU programming. We have examined how the hardware of the CUDA device relate to programmatic features. We have also seen how a developer would structure a CUDA application by specifying host and device specific code. CUDA gives developers access to a massively parallel architecture by using a familiar programming language and programming environment.

In chapter 5 we will go on to introduce the cheat detection solution where we use CUDA kernels running cheat detection code over many clients. Our implementation illustrates how CUDA applications might fit in with a larger software system.

# Chapter 5

# Implementation

We have used the former chapters to present the work and motivations of reducing cheating in on-line multi-player games. We have investigated hardware options for our solution along with possible GPGPU frameworks. In this chapter, we present our game simulation framework. We also describe how the cheat detection mechanisms we have created work.

## 5.1 Introduction

Throughout the thesis, we have listed several important points to consider when implementing cheat detection mechanisms in a game system. As clients in a game often have the same parametric properties, the cheat detection mechanism would in most cases be identical for every client it should check. There might of course be varying values for these parametric properties associated with each game object (i.e. the size or shape of game objects might change resulting in different physical behavior), but the motion of every object will be calculated with the same operations. This indicates that cheat detection is a problem solvable through parallel execution. The same code should be run on the different data coming in from all the clients. To allow the mechanism to run within games with many clients and high traffic, the hardware should be massively parallel. GPGPU applications have proved effective for highly parallel tasks, and we want to investigate how our solution for cheat detection performs on this hardware.

Because of the way we have intended to perform tests of the system (see chapter 6), we have implemented a game simulation rather than a real working game. We have created a physics engine to enforce the physical model we have defined. We have attempted to implement several cheat detection mechanisms, and for each of these, we have created a CPU version and a GPU version. The GPU version is implemented with the CUDA framework and runs on NVIDIA GPUs. We introduce the simulation by giving a general overview of its structure before describing the different relevant parts of the system in further detail.

## 5.2 General Overview

To mimic the properties of a game, we have decided to create a simple game simulation. With a simulation, it is easier to achieve the desired game structure and define a simple enough physics engine for our purpose. The simulation follows a client-server based game architecture, which is chosen for the same reasons as for consumer market game development: ease of development, total control of client communication and a centralized control point. The system is implemented entirely in the C++ programming language and written for a UNIX environment. The main libraries and templates used in the system are pthreads for multi-tread support, the Standard Template Library (STL), the Boost C++ Libraries [59], OpenGL [47] and GLUT [60].

Figure 5.1 illustrates how the simulation is structured during execution of *playback mode*. The definition of playback mode will be given later in section 5.2.3, but the illustration can serve for the purpose of describing how the simulation is generally structured. Even though the illustration indicates it, the simulation does not currently support networked traffic. All clients run on a single machine. However, discrete clients are created within the simulation, and communication follows the same flow which would be normal to a networked multi-player game. To reduce overhead and to avoid having to start many separate processes for each program launch, clients operating on the local machine do not run as separate processes, but within the main server process. We are interested in examining how our solution manages to scale up the number of clients it can handle by investigating how the cheat detection mechanism performs with an increasing number of clients. Network delays and connection limits are not taken into account with our solution. Further details about the topology and structure of the simulation is given in section 5.4.

To allow running reproducible tests, the simulation uses two different modes of operation named *generation mode* and *playback mode*, which we describe individually in sections 5.2.2 and 5.2.3 respectively. The simulation is multi-threaded and figure 5.2 displays the general program flow of the two initial threads of the system. The general program flow is the same for both modes of operation and it is mainly just the update routine

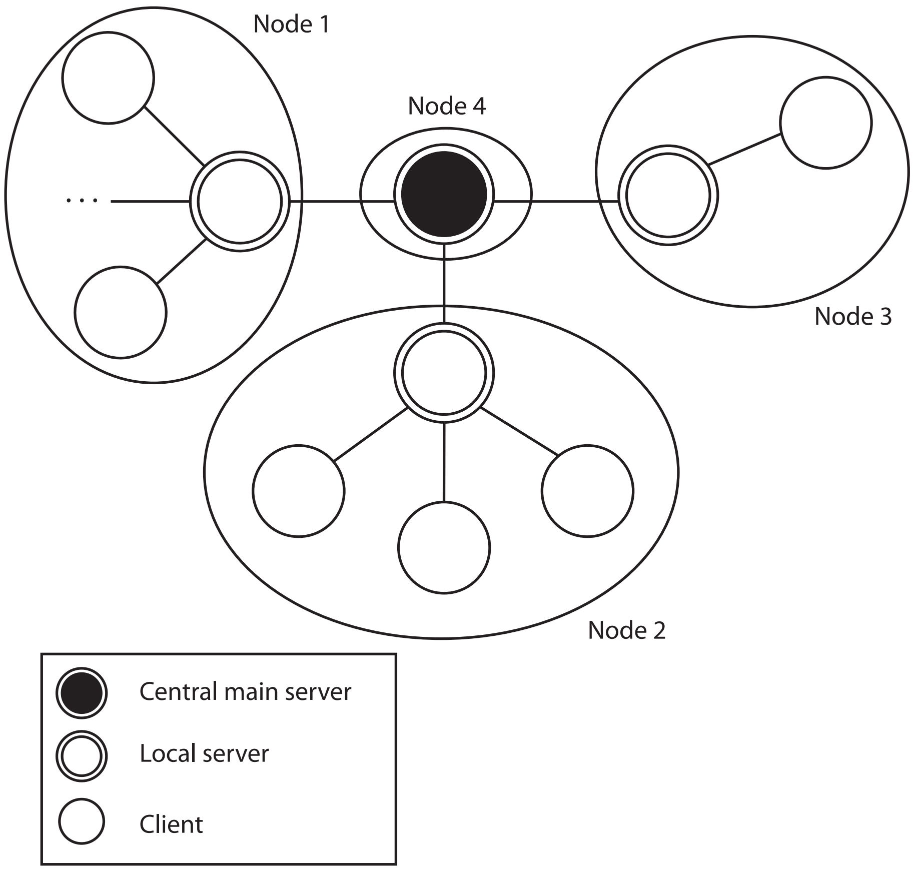

Figure 5.1: *Illustration of the topology of the simulation during playback mode.*

(colored red) which is the main difference between the modes. The start and stop threads methods define which threads should run alongside the control thread. For both modes there is a separate User Interface (UI) thread. There is no user interaction in the simulation besides the possibility to cancel execution, so the UI only displays the current state of the system. The UI thread needs access to the client container to display the current position of the clients, which means the client container must be protected by a mutex to ensure consistency between the main thread and the UI thread.

### 5.2.1 Principles of the game

The game simulator is a space race type of game where the main goal for the clients is to move around in a virtual environment, reaching preset random target locations. These targets are randomly placed out in the game environment. We can say that the first client to reach all of the targets is the winner, but we have not invested any time or effort into the playability of the simulation since all clients are controlled by the computer. The clients playing the game are placed randomly around in the virtual world giving some clients an

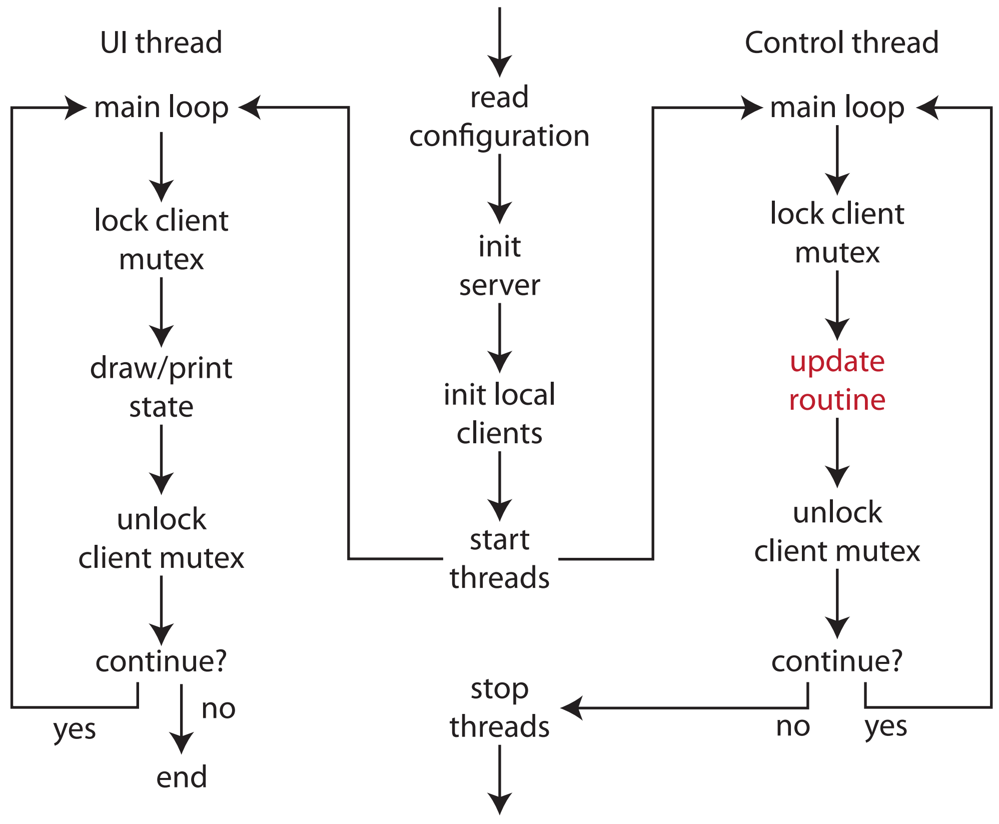

Figure 5.2: *Diagram of the general flow of the two initial threads of the system.*

advantage as they might be placed closer to the first target. Figure 5.7 is a screenshot of the Graphical User Interface (GUI) implemented to give a graphical representation of the game simulation. The GUI is described very briefly in section 5.6.

Because all clients of the game are controlled by the computer, some rules must be set for how they behave when trying to reach a target. To give the clients some kind of intelligent way to reach their targets, they require motion planning. We have not implemented any advanced motion planning algorithms for this thesis. They way the clients move are based on actions decided by just a few simple checks, making the clients rather stupid. They may in fact never properly reach the targets. Whether or not the clients reach their targets, the important thing is that the movement of the clients are restricted by the physical model. Honest clients will not break the rules of the model, while cheating clients will.

There are different ways to perform a cheat in our simulation. We have made clients cheat by either modifying the power of their thruster temporarily or by modifying the values of their current state: their position, velocity and rotation. If a cheating client temporarily increases the thrust capabilities of one of its thrusters, it will be able to accelerate faster in a direction to perform quicker turns or pick up speed faster. Cheaters who change their state can position themselves closer to a target or change their rotation to point towards a target. They might also increase or decrease the magnitude of their velocity vector when either dashing for a target or slowing down to prevent passing a target.

Because we wanted to design our simulation independent of real time, we have used an artificial timeline based on game ticks. A tick is basically a theoretical time duration specified in the configuration of the system. The duration of the tick is specified in our implementation in microseconds. A tick can also be thought of as the duration of a discrete game frame. Physical calculations are done over one tick and updates from the clients would arrive each game tick. So if an execution of the simulation manages to process ticks faster than their theoretical duration, the simulation is able to meet real-time requirements.

### 5.2.2 Generation mode

The generation mode uses the principles of the game (section 5.2.1) to determine its actions. The generation mode places a given number of clients randomly in a virtual environment. From these positions, the clients want to reach the different targets and using a thruster they propel themselves around. External forces, such as gravity, might affect the clients to make their paths a bit more complicated. While this is happening, each client writes periodically its location to a file, generating a list of positions. These files are used by the playback mode.

The flow of the update routine of the generation mode is illustrated by figure 5.3. The update routine iterates over all the clients and allows them to perform course changes to try and reach their target. The clients do this by turning on or off the different thrusters of the game object. After all clients have performed their actions, the physics engine is updated by adding up all forces from thrusters and external forces and calculating the integral of the forces over time interval defined by the tick duration.

The generation mode is also responsible for generating movement for cheaters. The cheaters are generated by using the normal checks of the honest clients, but regularly they perform unrealistic motions. This can be actions like halving the angle between the current orientation of the object and the target within a single game tick, or doubling the speed of the object within a unreasonably short time span. The files generated for cheaters and honest players are placed in the target motion directory under the subdirectories *cheat* and *clean* respectively.

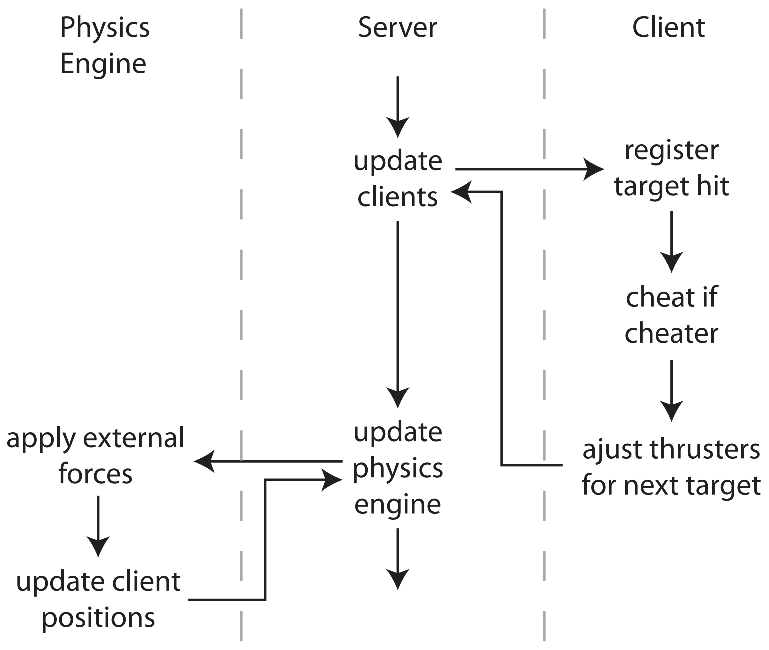

Figure 5.3: *Diagram of the flow of the update routine in generation mode.*

### 5.2.3 Playback mode

Similar to the generation mode, the playback mode initializes the given number of clients. The difference is that the clients running in playback mode find and open the files generated by the generation mode. For each game tick, client state information is read from the files and the clients report this state to the server. Clients open files from the same directory, so if there are more clients running than there are unique files there will be several clients reporting the same data from the same file to the server. The server samples the state information updates from every client, putting the samples in a sample queue. This occurs in the update routine of the server running in the control thread. The update routine follows the flow illustrated by figure 5.4. The sample queue is read by another thread, the *cheat detection* thread. The cheat detection thread runs the cheat detection mechanism on the samples in the sample queue when the queue is full.

The update routine of the playback mode updates clients just like the generation mode, but the clients get their positions, velocities and rotations from file, rather than generating these on the fly through the physics engine. After a client has been updated, the server samples the state of the client and places the sample in the sample queue. There is a minimum two sample queues in the system. One for adding new samples, and one that is used by the checking mechanism. The sample queues are protected by mutexes which are opened when either the sampler has filled a queue or the checker has performed a cheat detection check. Because samples need to be gathered from the clients, there is a natural

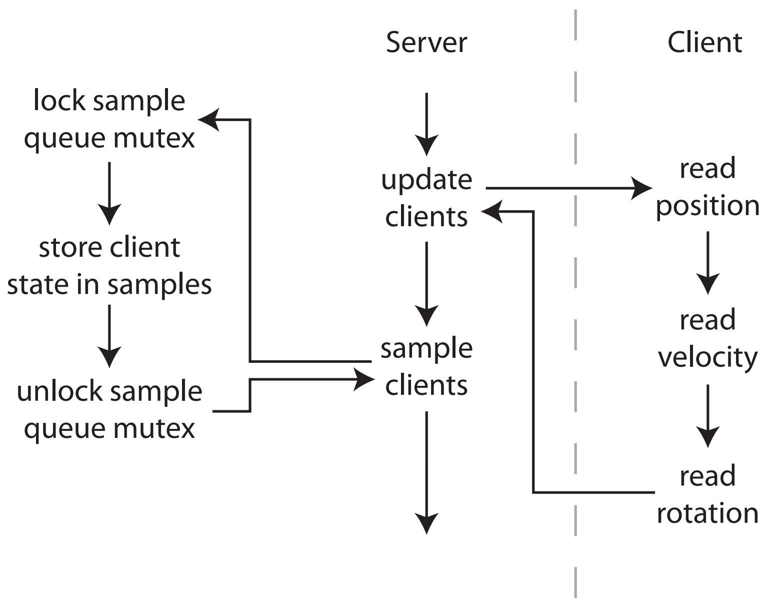

Figure 5.4: *Diagram of the flow of the update routine in the control thread in playback mode.*

delay before cheaters will be discovered. This means that cheating clients are discovered some time after the time the cheating behavior started. This is a very small delay though, which is rarely langer than a few seconds.

## 5.3 Physics Engine

The physics engine is one of the main parts of the simulation. The engine is responsible for calculating the sum of all physical forces acting on all objects and updating their positions accordingly. The physics engine is controlled by configuration parameters that allow for changing physical properties quickly, even during runtime. Objects are registered with the physics engine so it maintains a pool of objects to manage. Updates of the parameters of an object, such as throttle, is handled by the individual clients. The integrations of the time steps from a game tick to the next is done by the engine. The physics engine does this by updating every object in the object pool.

We presented some of the basic theory behind physical models in chapter 3. All physics calculations are dependent on time. To give reliable results, the simulation does not use real system time, but logic ticks to represent a timeline. The timeline is maintained by the external system, so the physics engine works just as well with real-time as it does with simulated time. The physics engine is mainly run off-line, or more correctly, as fast as possible. It does not have any real-time dead-lines that it needs to fulfill. The main implementation of the physics engine runs on the host and is only used during the generation mode. During playback mode, the cheat detection mechanisms in the different CUDA kernels act as a reverse physics engine, in the sense that they try to determine if positions generated by the physics engine are valid with the current physical model.

Collisions between objects is implemented in the physics engine, but it is not turned on. With the way the cheat detection mechanism works, collisions would introduce dependencies between the different checks, reducing the possibilities for parallel execution. We have not researched how collisions might be included in a cheat detection mechanisms like ours. The physics engine can also add boundaries to the virtual environment so that the clients do not wander off into oblivion, but rather bounce off the boundry walls.

## 5.4 Simulation Structure

Because our simulation uses a C/S communication model, there are two main node types in our simulation; the server and the clients. Even though the server and the clients do not communicate over a real network, they do exchange packets similar to network packets used in networked games. A packet is either generated by the generation mode or read from file in playback mode by the clients each game tick.

### 5.4.1 Server

Servers are the backbone of the entire simulation. No clients can run without a server because clients are running within the server process. Figure 5.1 illustrates how the simulation is planned to be run in playback mode with network support. Node 4 runs only a server. This is the server that is responsible for running the cheat detection mechanism. Node 1 to 3 run local servers that again handle a number of local clients. The servers in nodes 1 through 3 report the status of all their clients to the main server. These servers act like proxy servers. The reason we have chosen this solution is to allow any number of clients to run on any node.

These local clients are just internal object instances of a client class, thus existing only within the server. To distribute client state information files to be used by the playback mode, servers should be run in generation mode on any node that should be included in the playback mode. With regards to figure 5.1 this would mean node 1 through 3 would run generation mode before playback mode can be executed. It is also possible for the node 4 to run local clients while administering remote clients.

The server samples all incoming data from the clients. When a cheater is reporting erroneous positional data, the calculations should indicate that the movement of the player does not follow the rules and restrictions of physical parameters of the game. The server does not currently support clients to be on remote machines. However, the design is easily expandable to extend to the use of network communication for clients to send movement data to a remote server.

### 5.4.2 Client

The responsibility of the clients in the system depends on the execution mode of the program. During the generation of movement files, clients act as writers for the server, writing their locations and other appropriate data to file periodically. When the program is run in playback mode, clients read from the generated files and report the data written during the generation mode. This way, the system allows for reproducible tests as the test data is the same for each test run. If the simulation is to be expanded to support network communication, clients would be managed under one of the proxy servers on the node they are running, as illustrated in figure 5.1. How the main server running the cheat detection mechanism will manage the remote clients is not yet thought of.

We mentioned under section 5.2.1 how clients might cheat in our simulation. We have no motion planning in our implementation, but clients evaluate their position in the environment and try to adjust their thrusters to orient themselves towards their target. Cheaters might bypass the thruster adjustments by setting their position and orientation manually, manipulating the calculations of the physics engine. If the cheating client observes that its orientation is $x$ degrees of rotation away from pointing towards the target, they might halve this rotation by setting their rotation parameter. Whenever a client performs such a manipulation of its state, the cheat detection mechanism must be able to detect it.

## 5.5 Cheat Detection Mechanisms

We always implement two versions of a cheat detection mechanism. One is written for the host CPU, while the other is a CUDA version, written for the CUDA device. In essence, they execute the exact same operations on the same type of data structure supplied by the sampler. The CUDA device supports the kernel to run on a large number of threads to achieve massive parallel execution, while the host implementation only uses basic looping structures to simulate the same behavior. The host implementation is not threaded in any way.

To determine legal client movement patterns, the cheat detection mechanisms is given samples of the movement of each client. Every sample has a positional vector with three values: x, y and z, the three-dimensional axes. Another similar tuple holds the velocity of the game client object. The velocity is calculated in the first implementation, so it is not necessary to include this in the samples used in the first implementation. There are also a few other parameters mostly related to the physical properties of an object used in the mechanism, the object mass and maximum allowed acceleration. The sample structure is considered a simple model of a theoretical game packet structure.

There has been developed several cheat detection mechanisms for our system to test the performance difference of various approaches. The cheat mechanisms needed to be implemented for the diversity of existing games would have to vary considerably. Testing several possible solutions would wise to determine how the relation between memory and arithmetic operations impact the system. Throughout the thesis we have rewritten the cheat detection mechanism several times and attempted different approaches. We ended up with one solution in the end that we decided to test the performance of. We also made one that was not very successful, so we do not include that implementation when we present our results in the next chapter. We will, however, describe both of the two ideas we came up with and implemented.

The way the mechanisms are implemented for the host, mirror the way they are implemented in the CUDA kernels, except that the threads (labelled th*1* to th*n* in the illustrations below) are serialized when executing on the host. Apart from this, the host implementations should be identical in the general order of operations and which operations are performed. We will not focus on how the mechanisms are implemented on the host.

### 5.5.1 Stable Implementation

The first initial implementation idea of the cheat detection mechanism, deduces the velocity and acceleration of an object from the state information contained in three consecutive samples at a time. After finding the acceleration of the object as a three dimensional vector, all external forces existing in the physical model are subtracted. The resulting acceleration is the result of the forces the object itself has applied, which in the simulation would mean its thrust. If the thrust the object has applied is greater than the maximum thrust the object could be capable of applying, the client would most likely be a cheater.

In a way, the implementation reverses the physical behavior calculated by the physics engine.

Each block of the mechanism checks one client. The size of the block is determined by how many samples the mechanism should check for that client. Because each check requires three samples like we mentioned above, the total number of checks to be performed in a block is `num_samples - 2`. Each check is performed by one thread, resulting in `num_samples - 2` threads in a block. Each thread needs three consecutive samples to perform a check and so the reading and execution pattern of each thread is as illustrated by figure 5.5.

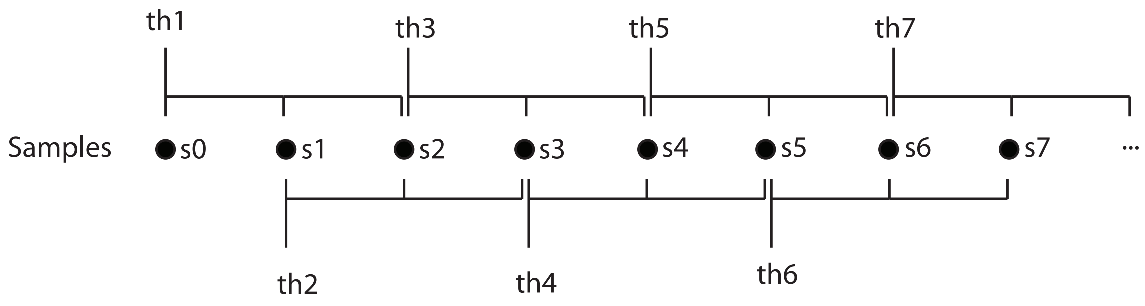

Figure 5.5: *Sample reading and execution pattern of the first CUDA kernel.*

In the beginning of our research, this implementation proved to be very inefficient. The physical model was too simple, which resulted in the GPU version being slower than the CPU version in almost every test. When we increased the complexity of the physics engine and the way the playback mode handled the buffering of the samples from the clients, the GPU implementation increased its throughput. See the next chapter for the benchmarks and results we obtained with this implementation.

The mechanism is able to detect cheaters using this implementation. However, it is not very accurate, marking some honest players as cheaters and missing some cheating participants. The implementation results in a scalar value indicating the thrust that must have been present to achieve the acceleration the object has had over the three samples checked. If the motion generated by the physics engine is close to the maximum acceleration the object can achieve in the physical model we have defined, the object might be marked as a cheater. Also, the motion of the cheating clients generated by the generator mode, might not always generate motion breaking the rules of the physical model. This results in cheating clients actually performing legal motion. To create a more accurate cheat detection mechanism, we must have invested more time into research of the problem of reversing the physics generated by the physics engine.

### 5.5.2 Discarded Implementation

The second implementation of the cheat detection mechanism uses its information about where an object is by checking a sample which contains a positional and a velocity vector. Based on these values, the kernel calculated several possible extremes for where the next sample might be positioned, and then checked if the next sample was within these constraints by interpolation. This implementation is a computationally heavier solution. It is also dependent of more information about the state of the clients, because it does not calculate the velocity of the clients like the first implementation. This means that more data has to be transferred to the device for the GPU implementation.

Figure 5.6 illustrates the sample reading and execution pattern of the second implementation. Where threads in the first implementation read and check three samples, the second implementation only reads two. To improve memory reads, the threads performs two checks each, so that the read operations do not overlap.

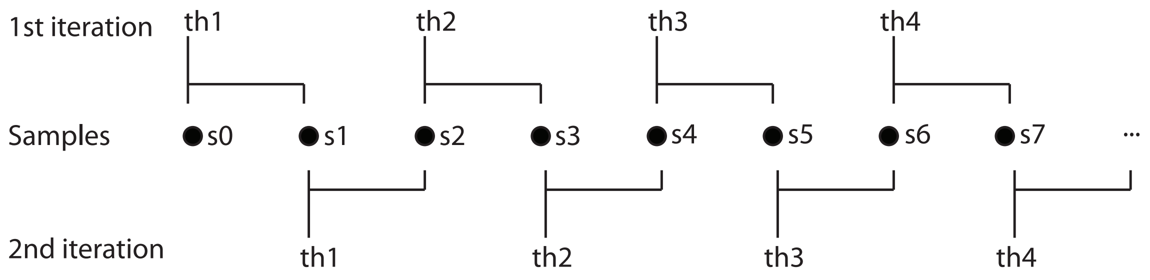

Figure 5.6: *Sample reading and execution pattern of the second CUDA kernel.*

Each thread in the kernel of the checker knows two positions of an object. By knowing the position, velocity and rotation of the object, the threads calculate a number of possible positions for the object to end up with accelerations in several directions: along the velocity vector, the opposite direction, right, left, up and down. When these positions are calculated, the thread checks if the second sample is withing these bounds by creating a bounding sphere through these points. Cheating has likely occurred if the second sample is outside of this bounding sphere.

## 5.6 Other parts of the system

Earlier in this chapter we have described the most important discrete parts of our game simulator. There are several more minor features that might be worth mentioning.

**GUI**

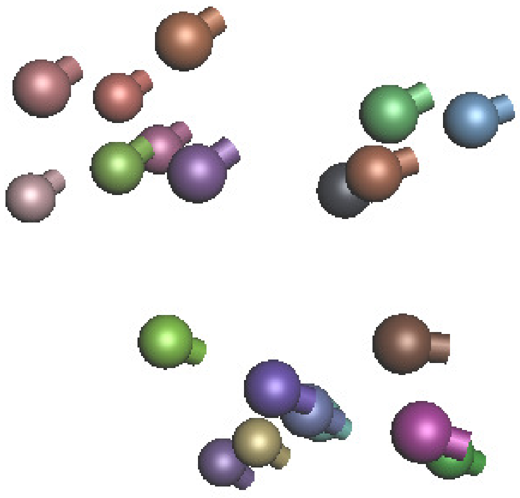

Figure 5.7: *Screenshot of the graphical representation of the game state.*

To see the result of the moving objects in the system a simple GUI was added. Figure 5.7 is a screenshot of the graphical representation of the game state. The GUI uses the GLUT library for windowing in a UNIX environment and OpenGL for the three-dimensional rendering. The GUI allows us to see the result of the moving objects. It is very simple and only intended to serve as a development aid.

### Configuration system

Most of the parts of the system are configurable through either command-line or an INI-like [61] file. The configuration is read at start-up and eases modification of parametric parts of the system (i.e., the physics engine) to avoid recompilation between changes. Table 5.1 lists a few of the most important configuration parameters used in the simulation.

### Templates

For ease of further development, two simple templates has been made: a three-dimensional vector and a quaternion representation. Both templates are complete with constructors, operator overloads and other utility functions. In theory, both of the templates should work in the GPU kernel code, but for some reason, a compile-time error occurs when trying to use the quaternion template for the GPU. The vector template supports to be used both in the host-specific code and in the CUDA kernel.

| | |
|---|---|
| **simulator** | |
| tick_duration_us | How long a game tick lasts in microseconds. |
| local_clients | How many local clients to instanciate |
| move_dir | Directory containing movement files for reading or where to place generated files |
| cheat_percentage | How large of a percentage of the clients should be cheaters |
| **generator** | |
| duration | The duration of the game session to generate movement files for |
| targets | How many targets to create |
| **physics** | |
| gravity_[xyz] | Three fields of a vector describing the gravitational acceleration |
| bounds_[xyz] | Set the physical bounds of the environment |
| collisions | Turn on/off collisions |
| **object** | |
| minimum_radius | The minimum radius of a game object |
| maximum_radius | The maximum radius of a game object |
| mass_density | The mass density of the game objects |
| max_engine_thrust | The engine thrust of the objects |
| max_bow_thrust | The maximum thrust of the bow thrusters |

Table 5.1: The most essential configuration parameters used in the simulation.

## 5.7 Summary

In this chapter, we have described how we have chosen to implement and create a cheat detection system. We have described the different parts of our implementation and how all the different parts fit together. Our simulation uses two modes of operation; generation mode and playback mode. The first generates movement files that can be used by the latter as input. When the simulation runs in playback mode, the cheat detection mechanism is active and attempting to detect cheaters. We have described how our cheat detection mechanism calculates the same operations the physics engine did in the generation mode to determine if the movements of the clients are consistent with our physical model. We have also described how the cheat detection mechanism is implemented both on the CPU and the GPU so that tests can be run to compare the two. In the next chapter, we present our results of the CPU and the GPU versions of the cheat detection mechanisms. We describe how we have performed tests of the system and explain our results and what is the cause of the results we have obtained.

# Chapter 6

# Testing and Results

In the previous chapter, we gave an overview of our game simulation with cheat detection. We described how the simulation uses two modes of operation; one to generate files containing data similar to that of packets in multi-player games, and one to gather data from game clients and run the cheat detection mechanism on samples of the data gathered from the clients. The two modes of operation was chosen to ensure that repeated tests with the same data, could be run on different cheat detection mechanisms and with different mechanism launch parameters.

In this chapter, we describe the performance of our solution by presenting the results of different benchmarks we have run on the system. In the previous chapter, we described that we attempted two different approaches at implementing cheat detection. Because of errors and lack of stability in the second implementation, we are not able to present any numbers of its performance in this chapter. There are many things to consider when optimizing CUDA applications for best performance. We try to explain our results and we give our viewpoints on how our system might be improved. We will also briefly discuss what we have learned from our implementation and how this would apply to similar systems.

## 6.1 Introduction

When testing the efficiency of a simulation like ours, there are several different elements to take into account. A benchmark strategy commonly used is to record the time different parts of a system takes to execute. The result of such a benchmark is only interesting if we compare the results of one execution with the results of another. Often the two executions would differ in one way or another, because identical executions should yield identical results, which in most cases is not very interesting. The difference between two executions can be the number of clients connected to a server, which is interesting to test in a simulation like ours. Such a test would give an indication of the scalability of a system. As an example, if the execution time of our cheat detection mechanism is four times as long with twice as many clients, the solution might not be very scalable. But if the alternative solution takes eight times as long, the first solution might be pretty good after all.

In a multi-core application, it is important for all the processing cores of the processor to be occupied with work. If cores are idle, the processor is not performing optimally. Occupancy is a term that is very interesting when benchmarking CUDA applications. Occupancy can be defined as a scalar value indicating how many cores of a multi-core device is busy processing data at any given time. A low occupancy indicates that processing power is wasted, while a high occupancy would indicate that the entire multi-core processor is active. With regard to CUDA applications, the occupancy is calculated as the ratio between the number of active warps and the maximum number of supported warps of the CUDA device being used. Results of an occupancy check can be interesting by themselves, as opposed to most timing-based benchmarks, because the check gives a good indication of how efficient the kernel is.

## 6.2 Detect the Cheaters

We have no results indicating exactly how many of the cheating clients our mechanism is able to detect. The success rate of our implementation is not what should be considered the important part of the game simulator. We are able to detect quite a few cheaters, but the mechanism we have created also seem to point out honest players from time to time. Also, cheating participants may not always be discovered. There are several reasons for this. Our mechanism calculates the acceleration an object has had over the three samples in a check. If the object is having an acceleration close to the maximum allowed by the physical model, the object might be marked as a cheater. The cheat detection check returns a value indicating the thrust required to achieve the acceleration reported by the clients. This value should be adjusted with a tolerance value so it does not include honest players. A consequence might be that more cheaters go unnoticed.

## 6.3 Test Setup

Before any tests can be run, it is necessary to generate a large number of data files. This is done by the generation mode of the simulation, as stated in the chapter 5. Generated files are placed in subdirectories of a target directory. The subdirectories are named *cheat* and *clean* and all files within these directories simulate either cheaters or honest players. If there is a desire to start more clients than there are generated files, the playback mode makes several clients share files. This means they report identical data to the server.

The cheat detection mechanism we have tested is implemented for both the GPU using CUDA and for the CPU. To compare the execution time of the two versions up against each other, we have to run the simulation with the same data twice; once for the GPU version and once for CPU version. To automate tests, we have written simple test scripts in the Python scripting language. They run the simulation in playback mode with various configurations to aggregate results of each execution. All timing internally (i.e., taking the time of a cheat detection mechanism launch) is done within the simulation and not by these scripts. The results of the internal benchmarks are written to standard output and collected by the test scripts.

## 6.4 Mechanism Execution Time

Determining how long the cheat detection mechanisms takes to execute is one of the main interests we have when testing the efficiency of our system. This can lead to a diverse number of tests, because each launch of the mechanism implies numerous variable parameters affecting performance. Because we are comparing the execution time of the mechanism on the CPU up against the execution time on the GPU, the launches will have to have identical parameters set. Because the GPU requires us to think of a launch as a number of threads contained in a number of blocks, we will use this terminology to describe the different factors affecting a launch.

**The number of clients** is probably the most important point to consider because we are checking for scalability. First of all; the mechanisms we have created try to check all clients in one launch. We have not looked into the possibility of checking subsets of the clients for each check. A major reason for this is that we attempt to find how well the system performs as we scale up the number of clients to check. If the launch of the mechanism checks just a subset of the clients, then we could have run the system with a smaller number of clients to achieve the same effect.

**The number threads in a block** is relevant as there is an optimal number of threads contained within a block for every given kernel giving the best throughput. This number should ideally be multiple of 16, stated in the CUDA Programming Guide. This relates to the warps and half-warps we discussed in section 4.4.4. If this number is not a multiple of 16, there is a risk of warps being underpopulated which can result in idle processing cores.

**The number of blocks in a grid** is another relevant factor affecting the performance of a launch. The SM performs context switches when threads of a warp wait for reads or writes to main memory. It is important that when a warp is switched out of context another is ready. If the launch consists of several blocks, then there are always warps ready to be switched into context, meaning the launch benefits from a high block count. Another benefit of high block counts, is the support of future devices with a higher number of processing cores.

We run tests changing the different values of the three major factors affecting a kernel launch that we mentioned above. This is automated by the test script we have written that also collects the information and calculates the average time of each mechanism launch.

## 6.4.1 Number of tests

Because of the many different values that we can set for the different parameters we listed above, we need to run a large number of tests. The test script calls the simulation with all the different values we want to use sequentially. What we are trying to show with the following tests is the difference in mechanism launch times between the CPU and the GPU. Both mechanisms are given the exact same data placed in identical buffers.

All tests are run on data generated by the generator mode over two minutes of "game time"[1]. For these tests we have set the tick duration to $1/40s$, meaning there are 40 updates per second. This means there are 4800 samples per client. We have used client ranges from 200 to 1200. For the largest tests, we have therefore a total of $4800 * 1200 = 5760000$ samples being used over the duration of the playback mode. The first execution, illustrated by the graph in figure 6.1, uses a much smaller buffer size than the last execution, illustrated by the graph in figure 6.4. A smaller buffer size means there are fewer samples to check and the mechanism runs quicker. Larger buffer sizes means more checks per kernel launch, giving longer execution times.

[1]Because we execute the simulation with a best-effort launch, it has no concept of time. The game time mentioned here is basically just what the physics engine simulates.

Be aware that in the following graphs, the scale of the y-axis may change quite considerably. The difference in the axis means that deviations in the plots in the graphs with a smaller scale will look more drastic as opposed to the graphs with a larger scale.

### 6.4.2 Results

In this section, we present graphs illustrating the results we have obtained from the benchmarks testing different number of connected clients, minimum block count and block size. In the first set of graphs we have calculated the average execution time of the CPU and the GPU implementations.

Figure 6.1 and 6.2, illustrate two launches with two different values for the block size. Both graphs illustrate executions where the minimum number of blocks in a mechanism launch is 16. The graph in figure 6.1 uses a block size of 256 threads, while the other uses a block size of 448 thread. The number of clients range from 200 to 1200. Figure 6.3 and 6.4 illustrate two graphs of a executions where the number of blocks used has been increased to 64. We have the same number for the block size as in figure 6.1 and 6.2, 256 threads for the first and 448 for the second. The clients range of clients is the same, 200 to 1200.

From the four graphs it becomes quite apparent that both the number of blocks and the size of the blocks greatly affect the performance of the GPU. GPU is able to outperform the CPU with almost every configuration we have tested. However, when the block size and the number of blocks is small, the GPU performs considerably worse than with larger numbers. In the tests with the smallest block sizes and the smallest number of blocks, the GPU runs approximately twice as fast as the CPU. When the block size and the number of blocks is increased the GPU runs three to four times as fast as the CPU.

There are two reasons for these results. The first is the memory transaction overhead associated with moving sample data to the GPU. If the data to be moved is too little the overhead of the memory transaction becomes a large part of the overall execution time of the mechanism. In these cases, the CPU will benefit from the fact that it does not require a memory transaction. The other reason for the low performance of the GPU when both the block size and number of blocks is small, is that large parts of the GPU will be idle.

Because the buffer size is direct consequence of the minimum number of blocks and the block size, for all the different number of clients we tested above, the buffer size will be the same. This is why the graphs are so straight, as it is natural to think that there would be some increase in execution time with a larger number of clients. When the

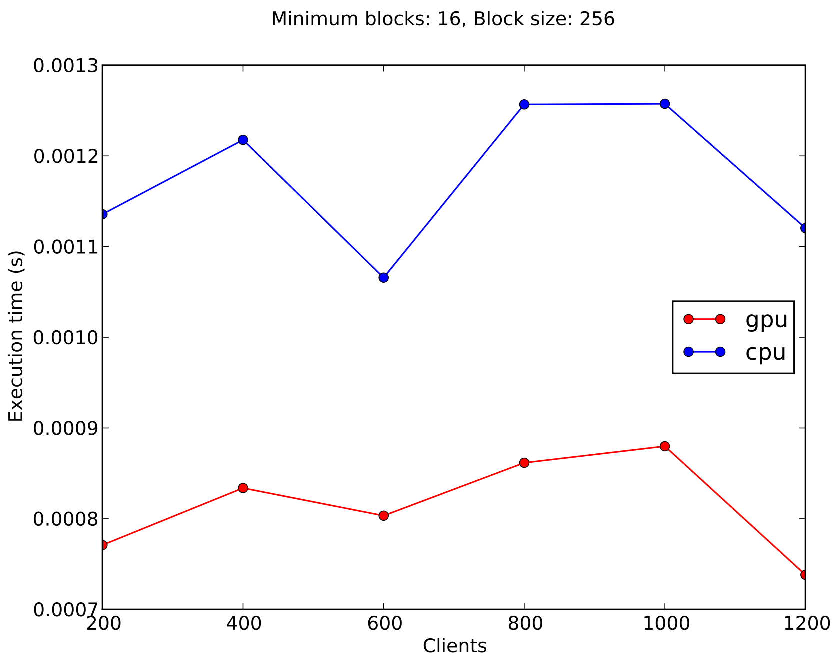

Figure 6.1: *The average cheat detection mechanism execution time of the GPU (red) and the CPU (blue). Minimum number of blocks is 16 and a block size of 256 threads. Lower is better.*

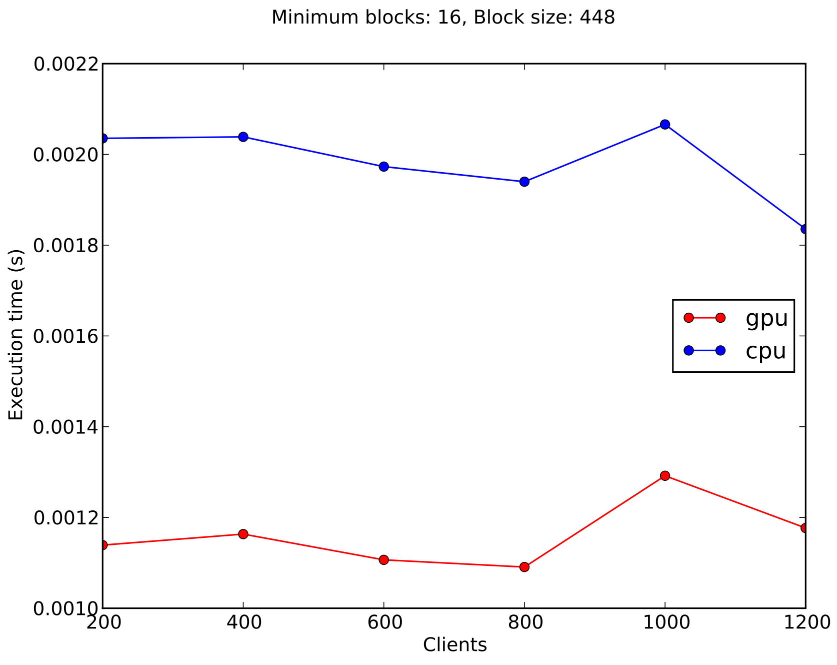

Figure 6.2: *The average cheat detection mechanism execution time of the GPU (red) and the CPU (blue). Minimum number of blocks is 16 and a block size of 448 threads. Lower is better.*

buffer size is the same over over the different number of clients, it means that the same number of checks are being performed by each launch of the mechanism. The difference

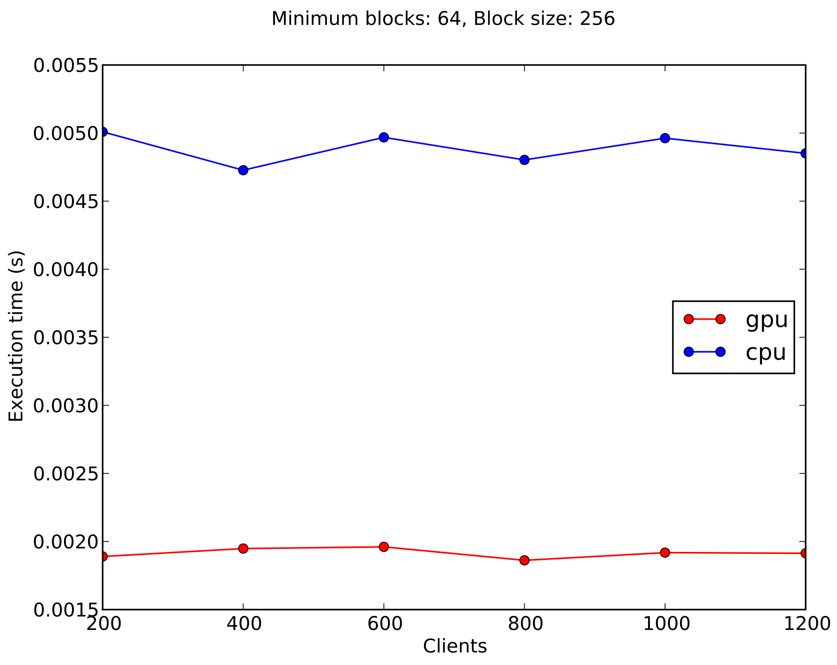

Figure 6.3: *The average cheat detection mechanism execution time of the GPU (red) and the CPU (blue). Minimum number of blocks is 64 and a block size of 256 threads. Lower is better.*

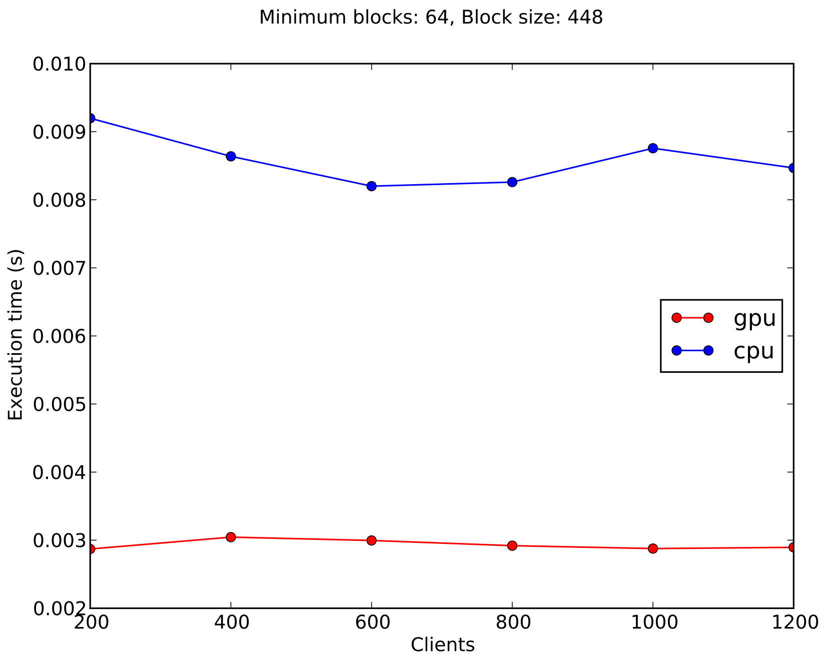

Figure 6.4: *The average cheat detection mechanism execution time of the GPU (red) and the CPU (blue). Minimum number of blocks is 64 and a block size of 448 threads. Lower is better.*

is that with fewer clients, the system will fill up its buffer slower than when there are more clients. This would mean that when more clients are connected to the system, the cheat detection mechanism is launched more frequently. To illustrate this, we present the graphs illustrated by figure 6.5 to 6.8. These graphs are generated from the same test data as the graphs in figure 6.1 through 6.4. But this time we have multiplied the average execution time of the mechanism with the number of mechanism launches for each test run. This gives us an estimate of how much time is actually being spent within the cheat detection mechanism of both the GPU and the CPU implementation.

There are two things that are apparent from the graphs in figures 6.5 to 6.8: most importantly the GPU is considerably faster than the CPU, and that the performance of the GPU is very dependent of the dimensions of both the grid and the blocks of a launch. With a careful look at all of the four graphs in the two figures, it is possible to see that the CPU has the same execution time independent of the number of blocks and the size of the blocks. This is because the CPU code is serial. For the GPU, however, it is quite clear that when we use a small number of blocks and a small block dimension, the performance is reduced. This can be explained by idle processing cores on the GPU wasting processing resources. From the first graph to the last, the GPU approximately doubles its performance, while the CPU is stagnant.

If we were to speculate on the projections presented by these graphs, it seems like the GPU also scales considerably better than the CPU. If we look at the graph in figure 6.8, we can see that the incline of the graph of the GPU is much gentler than that of the CPU. Alas, we have not any consistent test results with more clients. Without a simulation supporting network, all data must be read from the disk, resulting in an exponential increase in test execution time because of file I/O. This is a problem we see a bigger effect from in the next section.

## 6.5 Offloading Effect

When the cheat detection mechanism runs on the GPU, the CPU is relieved of performing cheat detection tasks and can instead work on other game relevant computation. Even if our results would have revealed that the execution of the GPU implementation took longer time than the CPU version, the GPU would still offload the CPU, allowing the CPU to perform other tasks. This offloading effect might be all that is necessary to implement a cheat detection mechanism in a game. As long as the CPU has enough resources to administer sampling of the movements of the clients and launch the GPU cheat detection mechanism, then the addition of a such a mechanism to a game will not have a large performance impact on the system.

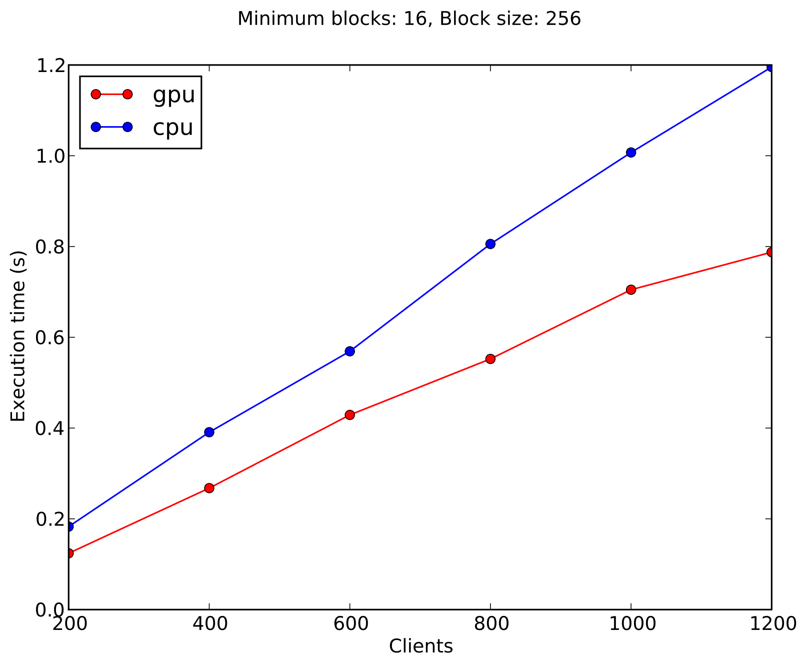

Figure 6.5: *A graph of the product of the average execution time and the number of launches of the cheat detection mechanism on the GPU (red) and the CPU (blue). A minimal number of 16 blocks of size 256 for each launch of the mechanism. Lower is better.*

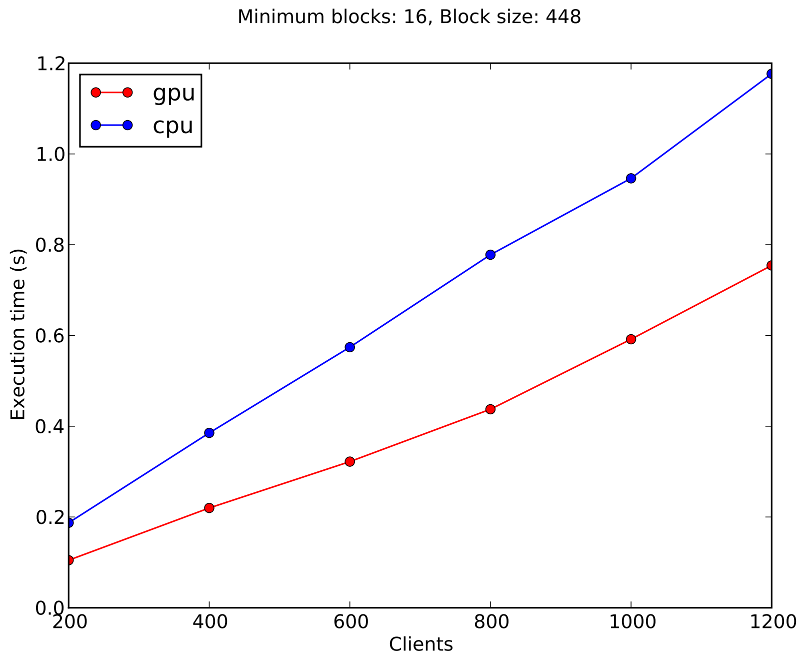

Figure 6.6: *A graph of the product of the average execution time and the number of launches of the cheat detection mechanism on the GPU (red) and the CPU (blue). A minimal number of 16 blocks of size 448 for each launch of the mechanism. Lower is better.*

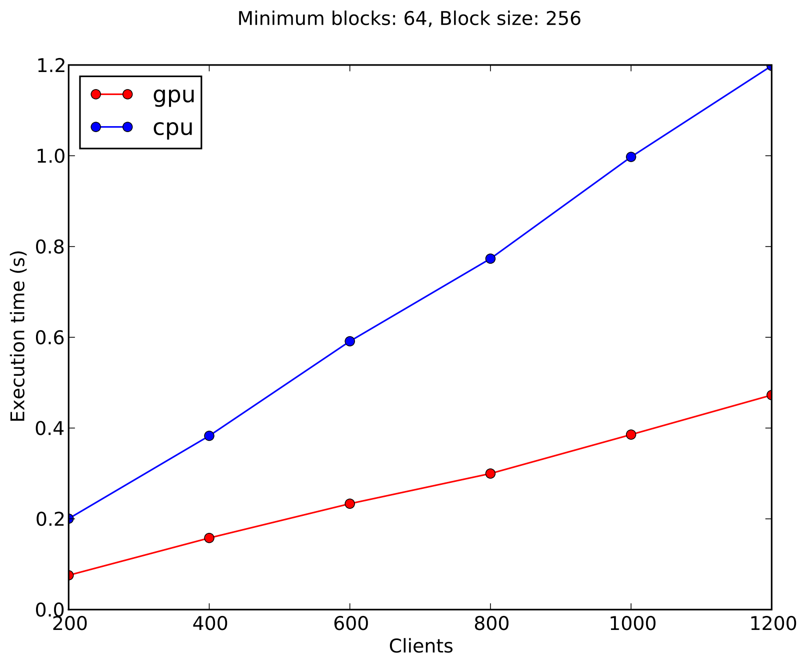

Figure 6.7: *A graph of the product of the average execution time and the number of launches of the cheat detection mechanism on the GPU (red) and the CPU (blue). A minimal number of 64 blocks of size 256 for each launch of the mechanism. Lower is better.*

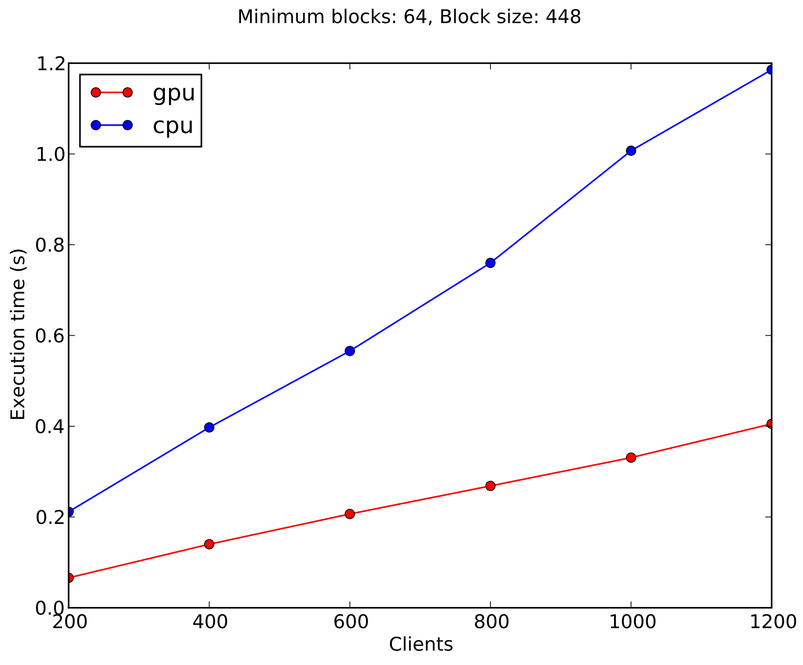

Figure 6.8: *A graph of the product of the average execution time and the number of launches of the cheat detection mechanism on the GPU (red) and the CPU (blue). A minimal number of 64 blocks of size 448 for each launch of the mechanism. Lower is better.*

To determine the offloading effect the GPU has for the CPU, we have measured the different CPU-times of user space execution of the two implementations. We have only included the user space time in our graphs because it is the most relevant, as the code executes in user space. We know from the test results we presented in section 6.4.2, that the GPU implementation is faster than the CPU implementation. We now want to see exactly how much the GPU relieves the CPU.

Figure 6.9 and 6.10 illustrate two graphs where we have used similar values for the block size and the minimum number of blocks to the ones we used in section 6.4.2. We can tell by the graphs in figure 6.5 that the GPU version takes less CPU time than the CPU implementation. We quickly notice that the difference between the CPU and the GPU implementations is not as great as we might have guessed. First of all, these graphs illustrate the time of the entire execution of the simulation. The framework does several other operations than just running the cheat detection mechanism, and all of these are handled by the CPU. When we inspected the profiler output from gprof [62], we saw that the majority of the execution of the simulation was within I/O related methods. Parsing the input data was also a major bottleneck. These time-consuming methods are used when running both the CPU and the GPU implementations. Therefore, even though the GPU version is considerably faster, the difference in execution time between the test runs using the two mechanisms is not that great. If we had been able to discover this sooner, we could have improved the I/O and parsing mechanisms of our simulation to probably speed up its execution considerably.

## 6.6 When to use the GPU

We have seen how the CPU and the GPU implementations of the cheat detection mechanism perform differently in relation to each other when we use different launch models. The difference between the two, is smallest when the number of checks to be performed is small. However, as the number of checks is increased, the increase in execution time of the CPU implementation is much steeper compared to the increase in the GPU implementation. This indicates that the GPU implementation is the most scalable of the two, in addition to being the fastest.

The cheat detection mechanism we have implemented for our system is easy to parallelize because there are no dependencies between what must be calculated for each client. Similar systems that have tasks that can map the operations that must be performed to a large number of threads, would benefit from using a GPU to offload the processing. A thing

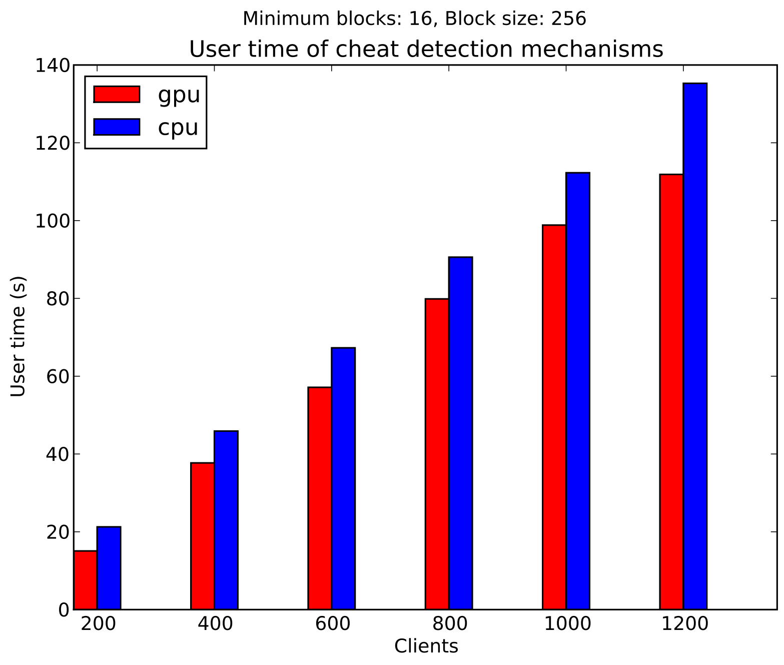

Figure 6.9: *Execution time in user space. Minimum 16 blocks with a block size of 256 threads. Lower is better..*

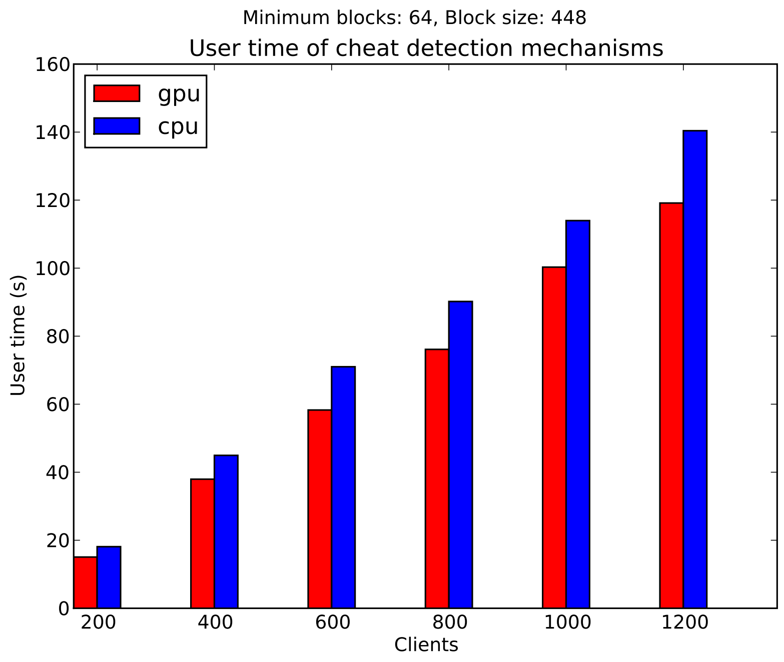

Figure 6.10: *Execution time in user space. Minimum 64 blocks with a block size of 448 threads. Lower is better.*

worth noting, is that the GPU is only effective if it has enough to process. Operations that require only a few calculations over a small number of threads would not be very efficient running on a GPU.

Although we have experimented with cheat detection in a game simulation, the GPU can be used for many additional tasks. If the game uses a physics engine that supports GPU execution, it might be able to perform all physics calculations on the server. If the scale of the game is not intended to be too great, this allows the game providers to perform all physics calculations on trusted hardware. This will reduce the control the clients of the game have and removes the need for a cheat detection mechanism checking for consistency in the movements of the clients entirely.

## 6.7 Summary

In this chapter, we have presented the results of our benchmarks of our cheat detection mechanism. We have seen that the GPU is able to outperform the CPU when performing cheat detection tasks. Offloading a cheat detection mechanism to a GPU not only speeds up the operation, but also frees up CPU cycles that can be used to perform other tasks related to game management. In the next and concluding chapter, we summarize the work that has been done in this thesis and we suggest further work that can be done to improve our existing solution.

# Chapter 7

# Conclusion

In this final chapter, we summarize what research has been done in this thesis, our main contributions and the results we have achieved. After a short summary of this thesis, we suggest a few ideas for the future that might improve, or at least extend the scope and functionality, of our game simulation.

## 7.1 Summary and Contributions

Even though there is an increasing popularity of on-line multi-player games, cheating is prominent. This destructive behavior degrades the gaming experience of honest game players. The game industry has always been a step behind the cheaters, struggling to keep up with new and creative cheating methods. Although the existing solutions are not sufficient to eliminate cheating, there is an increasing amount of research attempting to reduce cheating in on-line multi-player games. Because of the large diversity of the types of existing on-line games, the existing cheats and the cheating mechanism that aim to battle them, are equally diverse. In this thesis, we have presented a game simulator with a cheat detection mechanism aiming to detect cheating that exploits the physical model of a game. Computational resources in modern game infrastructures are already scarce, so the major goal we set for implementing a cheat detection mechanism was to keep its performance requirements low.

An increasing interest in GPGPU development and research, has given system developers easier access to the computational power of the GPU. The GPU is a relatively cheap, but powerful processor ideal for performing computationally heavy tasks that can be parallelized. In this thesis, we have investigated how a GPGPU cheat detection mechanism performs compared to a mechanism implemented on the CPU only. We have used the CUDA framework from NVIDIA to conduct our research on. The CUDA framework is a popular alternative in the GPGPU community and under constant development. It is also relatively well documented, not only because of helpful programming and optimization guides released by NVIDIA, but also for an active on-line forum of developers exchanging experiences with the framework.

We have obtained promising results indicating that a cheat detection mechanism running on a GPU, can outperform the same mechanism running on a CPU. Although cheat detection mechanisms will vary greatly from game to game, a mechanism checking for consistency in physical calculations can be migrated to the GPU to achieve a performance boost. We have also witnessed that by moving a cheat detection mechanism to the GPU, it is able to offload the CPU, allowing the CPU to perform other tasks while the cheat detection mechanism is executing. Even though there are many improvements that can be made to our game simulation framework, we have shown that cheat detection mechanisms can be offloaded to the GPU to relieve the CPU.

## 7.2 Critical Assessments

Implementing an entire game simulator with its own physical engine and cheat detection mechanism, is a time-consuming task. Many quite essential features are missing from our implementation; amongst network communication, a proper user interface, advanced physical model, client motion planning and a more advanced cheat detection mechanism. Especially the lack of network support has limited how we have been able to test the scalability of the system. We saw how much I/O operations affect our simulation in section 6.5. An improvement that probably would have improved the speed of our implementation quite considerably, is a change in the file format we have used for our simulation. The movement files are saved in a human readable format, which greatly increases the amount of parsing that is required to read the files back into the simulation in playback mode.

More time could have been dedicated to optimize and extend our game simulation. The physical model we have used is very simple and contains just a few of the many features used in mainstream physics engines. With a more advanced physical model, our cheat detection mechanisms would have to be extended considerably. This might have given us a different set of results on how the GPU would perform compared to the CPU.

## 7.3 Future work

Many improvements can be made to this cheat detection prototype. This section is devoted to possible additions and modifications to the current system which might lead to interesting results.

**Modular cheat detection mechanisms**

The server system should support modular CUDA kernels in shared libraries that can be connected and disconnected from the main system. This way, server uptime can be maximized as the server does not need to be taken down to add new cheat detection mechanisms. Running mechanisms can also be removed so that they could be changed or deleted if they are obsolete. Shared libraries with detection code would then act as system plug-ins that can also be tested on a staging server before being released onto a main game server. Also, with a plug-in based system, less significant cheat detection mechanisms could be disabled if the server load becomes unmanageable and enabled again when system load drops.

**Use network connectivity**

To test the scalability of the system further, network connectivity should be added. Then several nodes can connect to the checking server, supplying data from a higher number of clients. Solutions including proxies might also be tested if the system supports network communication.

**Change the simulation architecture**

The simulation is currently a client-server architecture. P2P games are emerging to battle scalability issues. Such systems lack the centralized control unit, the main server. To detect cheaters within a P2P system, some network nodes must be performing detection. Several possibilities exists for this problem. The game developer might introduce passive clients into the system to join game sessions searching for cheaters. These clients are not visible to other clients and do not participate in the actual game. They solely sample information about all other clients which they then run through the detection mechanism.

Cheat detection can also be made into a collective effort, where selected or all clients with free system resources perform smaller portions of the detection mechanism at turn. This way, cheaters can be discovered by the other player themselves. To make such a solution reliable, there must be a sufficient number of clients performing the check routines and redundant tests must be run so that clients can compare results. Clients with deviating results are likely a cheater. Both of these type of mentioned clients act as referees in the

P2P system. Webb et al [26] have done research on fair referee selection while minimizing impact on the system. They investigate how it is possible to solve the problem of safely choosing who will serve as checkers, or referees.

### CUDA physics engine

The main physics engine of the simulator currently only runs on the host system. The physics engine is mainly used during the generation mode, where the need for a cheat detection mechanism is not present. To improve the speed of the generator, the physics engine could be ported to CUDA. This is something already achieved by PhysX [31].

# Appendix A

# List of Acronyms

| | |
|---|---|
| **API** | Application Programming Interface |
| **AS** | Asynchronous Synchronization |
| **CBE** | Cell Broadband Engine |
| **C/S** | Client-Server |
| **CPU** | Central Processing Unit |
| **CUDA** | Compute Unified Device Architecture |
| **DRAM** | Dynamic Random Access Memory |
| **FPS** | First Person Shooter |
| **GPGPU** | General Purpose computing on Graphic Processing Units |
| **GPU** | Graphics Processing Unit |
| **GUI** | Graphical User Interface |
| **JIT** | Just-In-Time |
| **MMOG** | Multi-Player Online Game |
| **ODE** | Open Dynamics Engine |
| **P2P** | Peer-to-Peer |
| **PP** | Peer-Peer |
| **PRP** | Peer-Referee-Peer |
| **RACS** | Referee Anti Cheat Scheme |

| | |
|---|---|
| **SDK** | Software Development Kit |
| **SIMD** | Single-Instruction, Multiple-Data |
| **SIMT** | Single-Instruction, Multiple-Threads |
| **SM** | Stream Multiprocessor |
| **SMP** | Symmetric Multiprocessing |
| **SoI** | Sphere of Influence |
| **SP** | Stream Processor |
| **STL** | Standard Template Library |
| **TPC** | Texture Processing Cluster |
| **UI** | User Interface |
| **VAC** | Valve Anti-Cheat System |
| **VM** | Virtual Machine |
| **WoW** | World of Warcraft |

# Appendix B

# Source Code and Test Results

Attached is a CD-ROM with the source code developed during this thesis and the tests results of the benchmarks.

The contents can also be found on the following address:

`http://martinom.at.ifi.uio.no/master/`

# Bibliography

[1] Valve Software. The Orange Box. *http://orange.half-life2.com/*, Accessed July 2009.

[2] Visual Walkthroughs. Half Life 2 Walkthrough - Anticitizen One (Part Two). *http://www.visualwalkthroughs.com/halflife2/anticitizenone2/anticitizenone2.htm*, Accessed July 2009.

[3] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture: Programming Guide. *http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf*, Accessed December 2008.

[4] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, September 2005.

[5] NVIDIA. NVIDIA GeForce 8800 GPU Architecture Overview. *http://www.nvidia.com/page/8800_tech_briefs.html*, Accessed January 2009.

[6] Alexander Ottesen. Efficient parallelisation techniques for applications running on gpus using the cuda framework. Master's thesis, University of Oslo, Norway, May 2009.

[7] NVIDIA. The CUDA Compiler Driver NVCC. *http://www.nvidia.com/object/cuda_programming_tools.html*, Accessed January 2009.

[8] Wentong Cai, Percival Xavier, Stephen J. Turner, and Bu-Sung Lee. A scalable architecture for supporting interactive games on the internet. In *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 60–67, Washington, DC, USA, 2002. IEEE Computer Society.

[9] Jeff Yan and Brian Randell. A systematic classification of cheating in online games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9, New York, NY, USA, 2005. ACM.

[10] Pritchard, M. How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It. *http://www.gamasutra.com/features/20000724/pritchard_pfv.htm*, Accessed May 2008.

[11] Fun-Motion. List of Physics Games. *http://www.fun-motion.com/list-of-physics-games/*, Accessed July 2009.

[12] Steven Daniel Webb and Sieteng Soh. Cheating in networked computer games: a review. In *DIMEA '07: Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*, pages 105–112, New York, NY, USA, 2007. ACM.

[13] Patric Kabus, Wesley W. Terpstra, Mariano Cilia, and Alejandro P. Buchmann. Addressing cheating in distributed mmogs. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–6, New York, NY, USA, 2005. ACM.

[14] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 134–139, New York, NY, USA, 2004. ACM.

[15] Glider. Glider. *http://www.mmoglider.com/*, Accessed July 2009.

[16] Blizzard. World of Warcraft. *http://www.worldofwarcraft.com/*, Accessed December 2008.

[17] Steven Daniel Webb, Sieteng Soh, and William Lau. RACS: a referee anti-cheat scheme for P2P gaming. In *NOSSDAV'07: 17th International workshop on Network and Operating Systems Support for Digital Audio & Video*, pages 37–42, 2007.

[18] Valve Corporation. Counter-Strike: Source on Steam. *http://store.steampowered.com/app/240/*, Accessed May 2009.

[19] Daniel Bauer, Ilias Iliadis, Sean Rooney, and Paolo Scotton. Communication architectures for massive multi-player games. *Multimedia Tools Appl.*, 23(1):47–66, 2004.

[20] L Gautier and C Diot. Design and evaluation of mimaze a multi-player game on the internet. In *Multimedia Computing and Systems, 1998. Proceedings. IEEE International Conference*, pages 233–236. IEEE, 1998.

[21] Christoph Neumann, Nicolas Prigent, Matteo Varvello, and Kyoungwon Suh. Challenges in peer-to-peer gaming. *SIGCOMM Comput. Commun. Rev.*, 37(1):79–82, 2007.

[22] Paul Bettner and Mark Terrano. 1500 archers on a 28.8: Network programming in age of empires and beyond. In *Game Developers Conference 2001 - 22/03/2001*, 2001.

[23] Fabio Reis Cecin, Rodrigo Real, Rafael de Oliveira Jannone, Claudio Fernando Resin Geyer, Marcio Garcia Martins, and Jorge Luis Victoria Barbosa. Freemmg: A scalable and cheat-resistant distribution model for internet games. In *DS-RT '04: Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 83–90, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Christian Mönch, Gisle Grimen, and Roger Midtstraum. Protecting online games against cheating. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 20, New York, NY, USA, 2006. ACM.

[25] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Trans. Netw.*, 15(1):1–13, 2007.

[26] Steven Daniel Webb, Sieteng Soh, and Jerry Trahan. Secure referee selection for fair and responsive peer-to-peer gaming. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 63–71, Washington, DC, USA, 2008. IEEE Computer Society.

[27] Valve. Valve Anti-Cheat System. `https://support.steampowered.com/kb_article.php?p_faqid=370`, Accessed July 2009.

[28] Even Balance, Inc. PunkBuster Online Countermeasures. `http://www.evenbalance.com/`, Accessed July 2009.

[29] Wikipedia. Warden (software). `http://en.wikipedia.org/wiki/Warden_(software)`, Accessed July 2009.

[30] Valve. Steam Message. `http://storefront.steampowered.com/Steam/Marketing/message/837/?l=english`, Accessed July 2009.

[31] NVIDIA. NVIDIA PhysX for Developers. `http://developer.nvidia.com/object/physx.html`, Accessed July 2009.

[32] David M. Bourg. *Physics for Game Developers.* O'Reilly, 1st edition, 2002.

[33] Erin Catto. Box2D Physics Engine. *http://www.box2d.org/*, Accessed July 2009.

[34] Erwin Coumans. Bullet is a professional free 3D Game Multiphysics Library. *http://code.google.com/p/bullet/*, Accessed July 2009.

[35] The Blender Foundation. Blender.org. *http://www.blender.org/*, Accessed July 2009.

[36] Russell Smith. Open Dynamics Engine. *http://www.ode.org/*, Accessed July 2009.

[37] ODE. Products That Use ODE. *http://www.ode.org/users.html*, Accessed July 2009.

[38] Havok. Havok Physics. *http://www.havok.com/content/view/17/30/*, Accessed July 2009.

[39] NVIDIA. NVIDIA Forums - CUDA GPU Computing. *http://forums.nvidia.com/index.php?showforum=62*, Accessed July 2009.

[40] INTEL. The Evolution of a Revolution. *http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf*, Accessed December 2008.

[41] INTEL. INTEL High-Performance Consumer Desktop Microprocessor Timeline. *http://www.intel.com/pressroom/kits/core2duo/pdf/microprocessor_timeline.pdf*, Accessed December 2008.

[42] INTEL. Intel Core 2 Duo Processors. *http://www.intel.com/products/processor/core2duo/*, Accessed December 2008.

[43] IBM. The Cell project at IBM Research. *http://www.research.ibm.com/cell/*, Accessed December 2008.

[44] INTEL. Intel Microarchitecture, Codenamed Nehalem. *http://www.intel.com/technology/architecture-silicon/next-gen/*, Accessed July 2009.

[45] PC Perspective. Nehalem Revolution: Intel's Core i7 Processor Complete Review. *http://www.pcper.com/article.php?aid=634&type=expert*, Accessed July 2009.

[46] NVIDIA. NVIDIA GeForce GTX 200 GPU Architectural Overview. *www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf*, Accessed January 2009.

[47] OpenGL. The Industry's Foundation for High Performance Graphics. *http://www.opengl.org/*, Accessed January 2009.

[48] DirectX. DirectX: Advanced Graphics on Windows. *http://msdn.microsoft.com/en-us/directx/default.aspx*, Accessed January 2009.

[49] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley, 5th edition, 2005.

[50] NVIDIA. Cuda Zone. *http://www.nvidia.com/object/cuda_home.html#*, Accessed December 2008.

[51] AMD. AMD FireStream Stream Processors and FireStream SDK. *http://ati.amd.com/technology/streamcomputing/stream-computing.pdf*, Accessed December 2008.

[52] NVIDIA. High Performance Computing (HPC) - NVIDIA Tesla many core parallel supercomputing. *http://www.nvidia.com/object/tesla_computing_solutions.html*, Accessed January 2009.

[53] AMD. AMD Stream Computing: Software Stack. *http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf*, Accessed January 2009.

[54] AMD. AMD Stream SDK. *http://ati.amd.com/technology/streamcomputing/sdkdwnld.html*, Accessed January 2009.

[55] Khronos Group. OpenCL. *http://www.khronos.org/opencl/*, Accessed January 2009.

[56] AMD. AMD Drives Adoption of Industry Standards in GPGPU Software Development. *http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~127451,00.html*, Accessed January 2009.

[57] NVIDIA. NVIDIA Adds OpenCL To Its Industry Leading GPU Computing Toolkit. *http://www.nvidia.com/object/io_1228825271885.html*, Accessed January 2009.

[58] NVIDIA. CUDA-GDB: The NVIDIA CUDA Debugger. *http://developer.download.nvidia.com/compute/cuda/2.1-Beta/cudagdb/CUDA_GDB_User_Manual.pdf*, Accessed January 2009.

[59] Boost. Boost C++ Libraries. *http://www.boost.org/*, Accessed July 2009.

[60] GLUT. GLUT - The OpenGL Utility Toolkit. *http://www.opengl.org/resources/libraries/glut/*, Accessed July 2009.

[61] Wikipedia. INI file. *http://en.wikipedia.org/wiki/INI_file*, Accessed July 2009.

[62] Jay Fenlason and Richard Stallman. GNU gprof - The GNU Profiler. *http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html*, Accessed August 2009.