Master Thesis
Computer Science
Thesis no: MCS-2013-06
June 2013

Using Multicore Programming on the GPU to Improve Creation of Potential Fields

Hassan Elmir

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information

Author:
Hassan Elmir
E-mail: hassan_elmir@hotmail.com

University advisor:

Johan Hagelbäck, Ph.D.
School of Computing/Blekinge Institute of Technology

School of Computing
Blekinge Institute of Technology
SE-371 79 KARLSKRONA SWEDEN

Internet: www.bth.se/com
Phone: +46 455 385000
SWEDEN

Abstract

In the last decade video games have made great improvements in terms of artificial intelligence and visuals. Researchers have also made advancements in the artificial intelligence field and some of the latest research papers have been exploring potential fields. This report will cover the background of potential field and examine some improvements that can be made to increase the performance of the algorithm.

The basic idea is to increase performance by making a GPGPU(General purpose graphic processing unit) solution for the creation of potential fields. Several GPGPU implementations are presented where focus has lied on optimizing memory access patterns to increase performance. The results of this thesis show that an optimized GPGPU implementation can give up to 18.5x speedup over a CPU implementation.

Keywords: Potential Field, GPGPU, Memory Optimization.

Acknowledgement

I want to thank my supervisor **Dr.Johan Hagelbäck** for his great support and guidance throughout this thesis work.

I would also like to thank my **mom** and **dad** for their never ending support during my studies.

Hassan Elmir
June 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The concept of potential fields was first introduced by O. Khatib in the field of robotics and he called it Artificial potential field[1]. He used the potential field as real time obstacle avoidance for manipulators and mobile robots. Since then other researches have explored the use of potential field (or variations of it) as a navigation and obstacle avoidance tool for robots [2][3].

Potential fields have not become as popular in the game AI research as it has in robotics. There are however some research papers that study the use of potential field in games. The first research paper that used Influence maps (which is similar to potential fields) with games was A.L. Zobrist in 1969 [4]. In 2008 Hagelbäck, J. and Johansson S. J. studied the use of potential field with the Open Real Time Strategy (ORTS) game, where they showed that the potential field can be a good tool to use for both tactical decisions and pathfinding [5]. Potential fields will be discussed in further detail in chapter 2.

The GPU (graphic processing unit) have become very powerful in the last decade. Many researchers have used the parallel processing power of GPGPU (General purpose graphic processing unit) to enhance physics simulations, audio calculations data mining, and cryptography [6] [7]. Some research papers have mentioned the use of potential fields with the GPU to increase performance without presenting implementation details [8]. One research paper that was found went into greater detail of the implementation but the problem that was solved was a very specific pathfinding problem [9].

1.1 Objectives

The objective for this thesis is to examine different potential field implementations on the GPU. Other implementations of the potential field have

been put in a specific context such as pathfinding or crowd control. This implementation instead focuses on a more general use of a potential field that represents space with objects that exert charges around them. This implementation could later be used in a more specific context such as pathfinding. This study focuses on optimizing the creation of potential fields.

1.2 Research Question and Methodology

The research question posed in this thesis is:

How much performance speedup can be achieved when moving the computation of potential fields from the CPU to the GPU when optimizing memory access patterns?

A quantitative approach is used to answer the research question. Several versions of the potential field are implemented:

- CPU Naive
- CPU Memory Copy
- GPGPU Naive
- GPGPU Coalesced Memory
- GPGPU Shared Memory

The CPU memory copy version is based on J. Hagelbäck's optimized implementation that have been proven to be effective and suitable to use in strategy games[10]. The other versions are GPGPU implementations. The first GPGPU implementation is a naive version that is used for comparison with the optimized GPGPU versions. Focus on the optimized versions will lie on optimizing memory access since it is the biggest bottleneck in GPGPU computing [11].

When the implementations are done a benchmark will produce execution times for the various implementations when modifying parameters such as total number of entities, resolution of the potential field and different sizes on entities, where an entitiy represents an object with a specific shape and size.

Chapter 2

Potential Field

A potential field is a 2D grid representing some space. Charges are applied to the potential field to indicate important positions in the space. There are two kinds of charges, positive and negative charges. Objects, like enemy units, create a potential field around their position that is made of positive or negative charges. The size of an objects potential field can vary, depending on the size of the object itself. All charges that are created by objects are summed and put together in a 2D field representing the total potential field of the space. An example is shown in Figure 2.1.

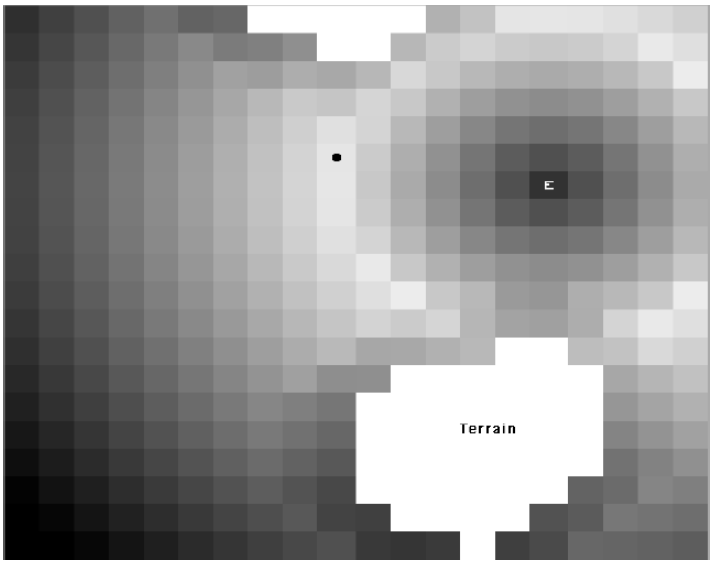

Figure 2.1: Figure of potential field in courtesy of J. Hagelbäck and S.J Johansson[13]. White space represents impassable terrain. E represents an enemy unit.

In a game the positive or attracting charges are objectives that the AI want

to reach. The negative or repelling charges symbolize positions that the AI want to avoid, like impassable terrain and buildings. An object can also create both positive and negative charges around it where negative charges are positioned near the object and positive charges are positioned further away. This enables objects to move in group without colliding with each other. The potential fields that are created by objects are calculated using a potential field function. The potential field function can vary depending on the type of object. J. Hagelbäck presents a number of potential field functions used in his studies [10], an example is shown in Figure 2.2.

$$u(x) = \begin{cases} 5.25 \cdot d - 37.5 & \text{if } d \leq 4 \\ 3.5 \cdot d - 25 & \text{if } d <\in [4, 7.14] \\ 0 & \text{if } d > 7.14 \end{cases}$$

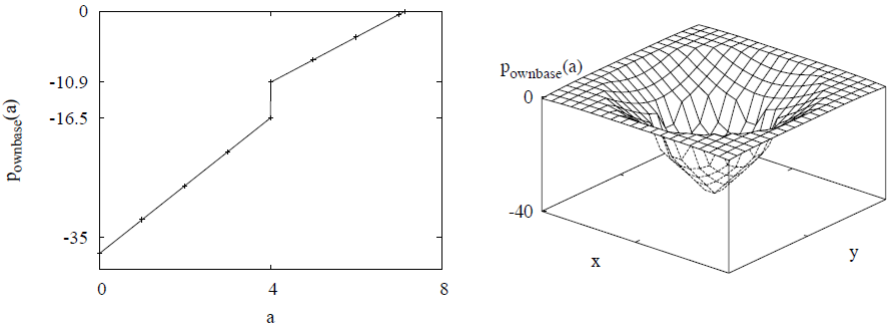

Figure 2.2: The potential generated by a base given a distance d.

There can be more than one potential field used in an AI where each potential field has its own purpose. J. Hagelbäck describes three different potential fields that were used when making the AI for ORTS: Field of Navigation, Strategic Field, and Tactical Field [10]. Each one of the potential fields had a specific purpose and was used for different parts of the AI. Potential fields can also be added together to get different types of overview of the space

2.1 Difference between Influence maps and Potential Fields

Influence maps are similar to Potential fields in some aspects but are different in others. They are both a grid-based representation of some space, but the value of each cell is calculated differently in both techniques [12]. In potential fields the value of a particular cell is calculated using some sort of distance evaluation method such as the Euclidean distance or Manhattan distance between the cell and the charge. An influence map however calculates cells values by letting the initial value from the charge propagate to neighboring cells [4]. An example of how Killzone uses potential fields and influence maps is shown in Figure 2.3. A description on how these techniques are related and how they are used can be found in [12].

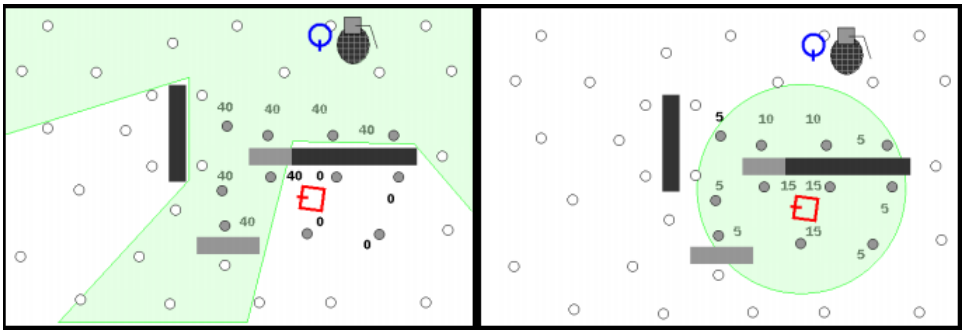

Figure 2.3: Example of mapping threat in Killzone[17]. In the left image influence maps are used for detecting line of fire. In the right image potential field is used for calculating waypoints within blast radius.

2.2 Usages of Potential fields

Potential fields have been utilized in different areas of artificial intelligence, mostly in the field of robotics but also in games. This section will cover some of the usages of potential fields in the academic world as well as in the game industry.

2.2.1 Pathfinding

One of the studied usages of potential fields in games is pathfinding. A* have been the most commonly used method for calculating navigation paths in games. Some research papers have experimented with several methods of optimizing the algorithm to make it more useable in real-time environments.

Potential fields have however been proven to be feasible to use for pathfinding in a real-time environment when the algorithm is combined with A* [13]. Another study made in 2006 shows how one can use a hybrid of both techniques [14]. The study suggests that after finding all possible actions the AI can take, an A* algorithm could be used to evaluate the paths to these actions and calculate the feasibility of reaching them. The A* in this case being the last step in the process of deciding which actions to take and which to discard.

2.2.2 Tactical Decisions

Potential fields can also be used as a basis for making tactical decisions. The spatial nature of the potential field makes it an intuitive tool to use when deciding how a group of units should be positioned on a map. It has been shown that potential fields can produce good results in performing unit formation planning with subgroups of units [15]. Potential fields have also been used to efficiently coordinate units to carry out attacks on an enemy while evading damage from the enemy when units are unable to fight (for example when they are reloading weapons) [10]. The study showed that the potential field can be an effective tool when micromanaging units.

2.2.3 Game Industry

Information about the usage of potential fields in games is sparse, and it is mostly academic projects that explore different ways to use it in games. There are however a few sources where we can see the use of the technique in the game industry.

Influence maps have been used in the game series Age of Empire[16]. It was an important tool for terrain analysis. They used it to analyze the terrain to detect the best positions to put resources on a map. The article also describes an interesting technique that they call Multiple Layer Influence Map, where each layer would describe a specific aspect of the terrain. The size of a cell in the Multiple Layer Influence Map was one byte. They used each bit in that byte to represent a different layer of the total influence map thus reducing the total required memory of the technique.

A combination of both potential fields and influence maps were used for the AI in Killzone[17]. They were used for analyzing the positions around a unit and then determine which position was most favorable to be at. They also used these techniques to enable the AI to stay out of enemys line of fire. In Killzone the terrain can change and covers can be blown away. The

reason why they used these two techniques was because of their capability of adapting to changes in a dynamic world.

The developers did not use influence maps in the most common way where the influence map represents the whole map in the game. Instead they chose to have several smaller influence maps to represent smaller portions of the terrain where there are units. They did not use any pre-calculated maps either, all influence maps were calculated in run-time when they were needed.

Chapter 3

GPU Architecture

The developments of GPUs have been driven by the video games market and they were created for the purpose of real-time rendering to the computer screen. In the last decade the GPU have evolved from being a fixed function pipeline to a programmable parallel processor that can outperform a modern multicore CPU in large scale parallel computing. This increase in performance boost has lead researchers to start use the GPUs computing power for non-graphical purposes which lead to the creation of a new programming field called GPGPU. Nowadays it is possible to use GPU for non-graphical programs without the need to go through the traditional graphics pipeline stages (vertex, pixel etc.). There are several APIs that can be used for GPGPU applications such as CUDA, OpenGL or DirectCompute.

To fully understand how to optimize an application for GPGPU one should have a good understanding of its architecture. This section will provide an overview of the GPUs processor architecture.

3.1 Thread Hierarchy

When using the DirectX[1] or OpenGL[2] API developers write shader programs(e.g. Pixel shader or vertex shader) that will run on the GPU. A shader is a program that describes how to process each thread during execution. Similarly, a compute shader (when using DirectCompute) or kernel program (when using CUDA[3]/OpenGL) is used to run code on each thread during execution without having to go through any stages in the graphics pipeline.

[1] $http://windows.microsoft.com/sv-se/windows7/products/features/directx-11, accessed 2013-04-14$

[2] $http://www.opengl.org/, accessed 2013-03-04$

[3] $http://www.nvidia.com/object/cuda\_home\_new.html, accessed 2013-02-11$

Before executing the GPGPU program the host (CPU program) must transfer data from the CPU memory to GPU memory. After that the host is ready to execute the shader/kernel. There is a specific thread hierarchy of grids, blocks and threads that must be taken into consideration when executing the shader/kernel. Figure 3.1 shows the different levels in the thread hierarchy.

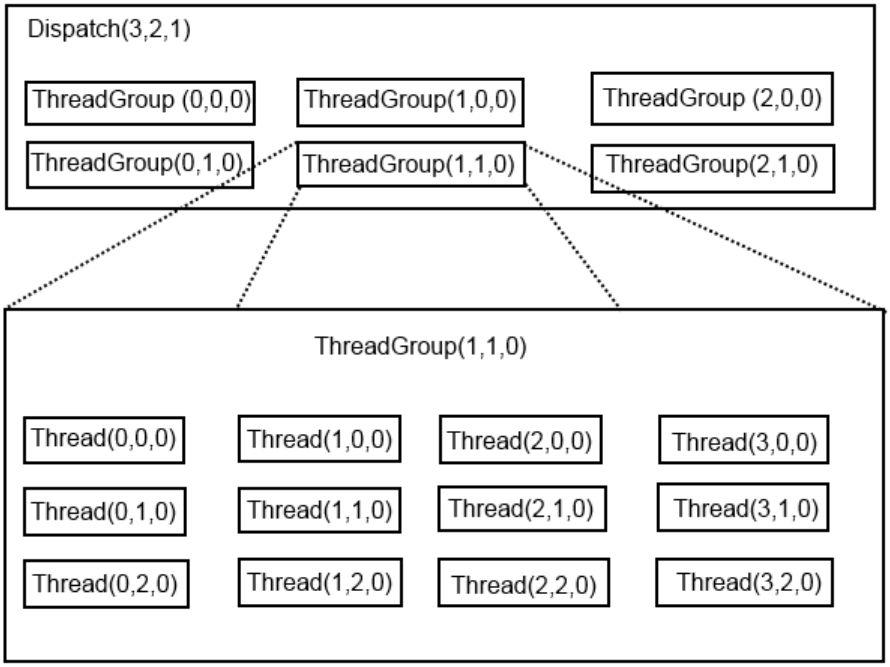

Figure 3.1: Example of thread hierarchy when dispatching 3x2 threadgroups where each threadgroup has 4x3 threads

3.2 Streaming Multiprocessor

The SM (Streaming Multiprocessor) is the processing unit on the GPU. On a modern GPU each SM has 48KB of shared memory, 64k registers, access to off chip memory and 8 SPs (streaming processors). The SP is the thread processor in the SM and it performs the fundamental floating point operators such as add or multiply.

The hardware for an SM is specifically designed for multithreaded purposes. Each SM can manage a large number of threads with very low scheduling overhead. The SM uses SIMT (single instruction multiple thread) to execute groups of threads called warps [18]. In Figure 3.2 we can see a basic overview

of the GPU architecture.

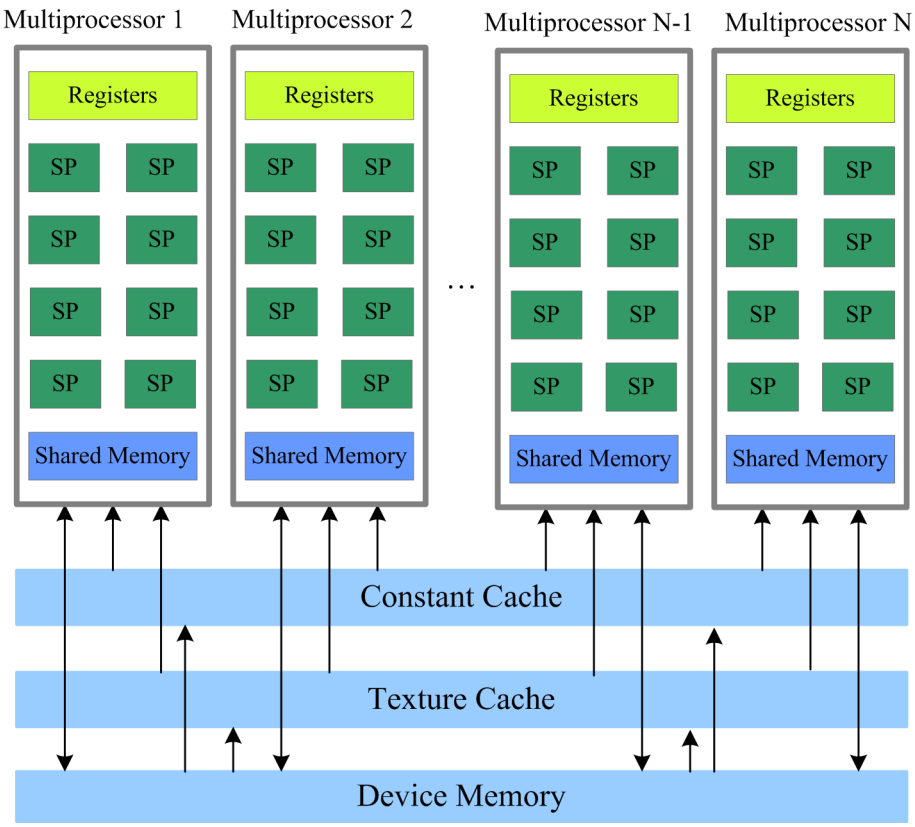

Figure 3.2: Overview of the GPU processor architecture

3.3 SIMT

To manage a large number of threads NVidia uses what it calls the SIMT model. Much like when writing a c program the programmer can basically ignore the underlying cache structure and still get correct results and behaviors. Knowing the attributes of the underlying architecture can however lead to better code structures that can improve the performance of the application.

The SIMT architecture is similar to SIMD (Single instruction multiple data) which performs the same operation on multiple data simultaneously to exploit data level parallelism. The difference is that SIMT performs the same instruction over multiple parallel threads as well as controlling the branching behavior of threads.

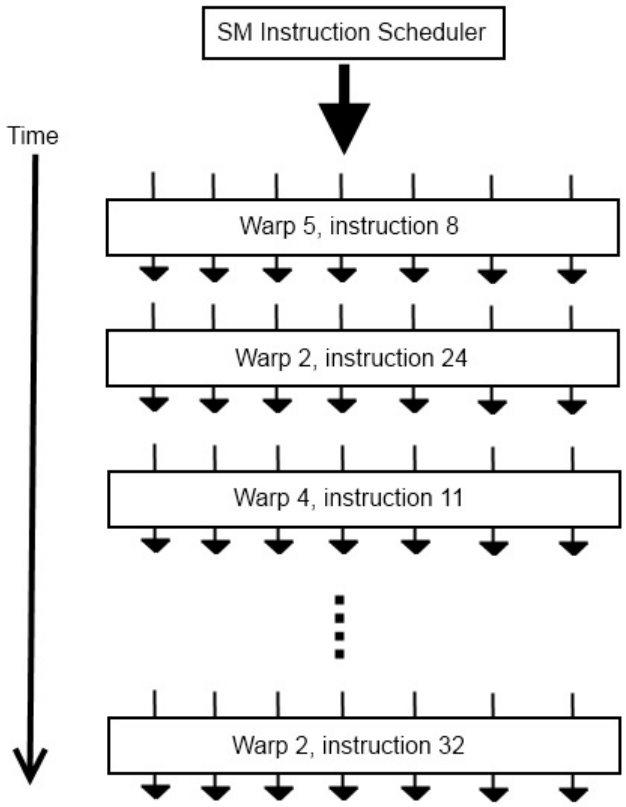

Figure 3.3: Illustration over warp scheduling in a SIMT model

A warp is the basic unit of scheduling in a SIMT model and each warp contains a group of threads that starts executing at the same time. The SM maps the threads in a warp to SP cores and they can execute simultaneously. The threads are free to branch and execute separate code paths but it is however unfavorable. If a thread in a warp diverges via a conditional branch the warp starts to serially execute each branch taken, disabling the threads that are not in that path, and when all paths complete, the threads reconverge to the original path. This means that the SIMT is the most effective when all threads of a warp take the same execution path.

The SP stalls whenever a thread executes a memory access instruction. When this happens the SM will schedule another warp to start executing on the SP as Figure 3.3 shows. This type of scheduling can effectively hide memory access latency since the overhead of scheduling warps is low.

Chapter 4

Implementation

This chapter will cover the different potential field implementations that have been done for this thesis. Each subchapter covers one implementation, and all implementations use circles as entities that exert charges around them.

4.1 CPU

Two CPU versions were implemented. The first version is a naive implementation of the algorithm with no optimizations. The second version is an optimized memory copy implementation as described by J. Hagelbäck. The memory copy implementation was chosen because it have been proven to be efficient enough to be used in strategy games[10].

4.1.1 Naive CPU Implementation

Algorithm 1 shows the naive CPU implementation. It consists of two outer for loops that loop through all the cells in the potential field. The total amount of potential exerted from each entity on a cell is then computed in an inner loop and written to the potential field. This is a brute force implementation since all cells in the potential field will calculate the potential of all entities on them.

ALGORITHM 1: Naive CPU Implementation

1: **Input:** vector<CircleEntity*>_EntityList
2:
3: float $DimRatioX \leftarrow WorldDimX/PFDimX$
4: float $DimRatioY \leftarrow WorldDimY/PFDimY$
5: $nbEntities \leftarrow EntityList.size$
6: **for all** y to y<PFDimY **do**

```
7:   for all x to x<PFDimX do
8:     float charge = 0
9:     Float2 worldPosition(x * DimRatioX, y * DimRatioY)
10:    for all i to i<nbEntities do
11:      charge += EntityList[i].GetChargeAtPosition(worldPosition)
12:    end for
13:    int chargePos ← x + (y * PFDimX)
14:    PotentialField[chargePos] ← charge
15:  end for
16: end for
```

4.1.2 Memory Copy CPU Implementation

This CPU version is an improved implementation that uses pre-calculated charges to speed up the creation phase of the potential field. In the first phase the potential of a temporary entity positioned at (0,0) is calculated, as seen in Figure 4.1, and the result is stored in two arrays. The first array contains the charges exerted by the entity and the second array contains the position of each charge. The number of charges that are being pre-calculated depends on the resolution of the potential field as well as the size of the entity. The higher the potential field resolution or size of the entity the more charges will be pre-calculated. In phase two the charges of this temporary entity are copied to the positions of actual entities residing within the potential field.

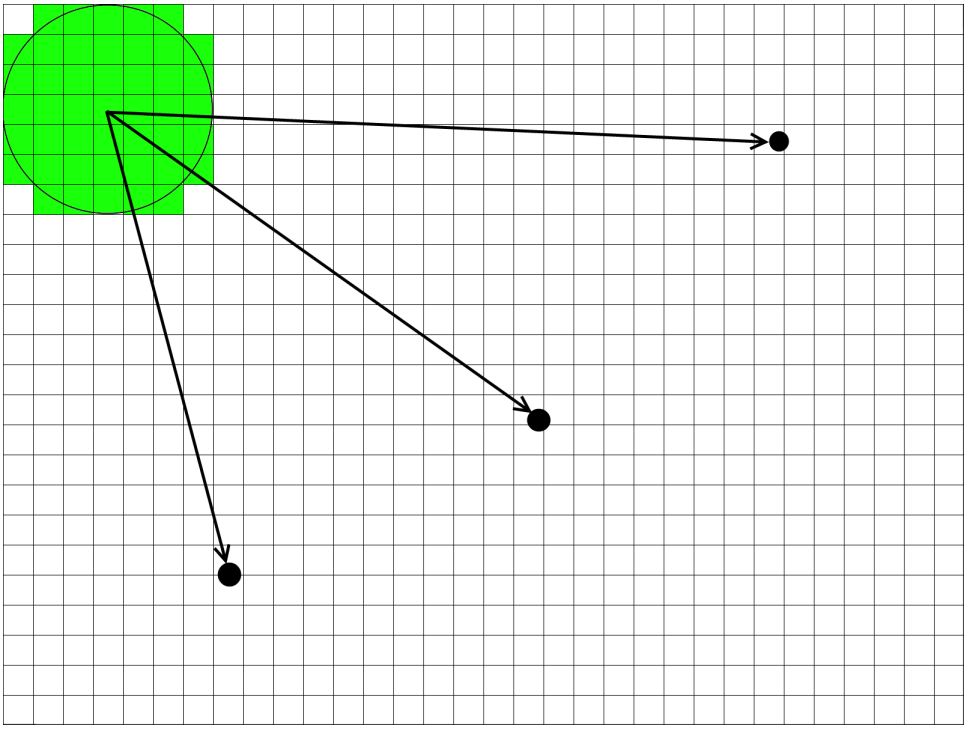

Figure 4.1: Charges of the temporary entity are calculated at position (0,0). These charges are then copied to the positions of actual entities represented by black dots.

As we can see in Algorithm 2 little computation is required in the second phase, when creating the potential field. At line 12, each entity offsets the pre-calculated charge positions by its own position to get the actual charge position for the entity. A safety check is then done to make sure the charge position is inside the potential field before adding it.

ALGORITHM 2: CPU mem copy

1: **Input:** vector<float>_Charges
2: vector<Float2>_ChargePositions
3: vector<CircleEntity*>_EntityList
4:
5: $nbEntities \leftarrow \_EntityList.size$
6: $nbCharges \leftarrow \_Charges.size$
7: $dimRatio \leftarrow WorldDim/PFDim$
8: **for all** i to i<nbEntities **do**
9: $\quad entityPos \leftarrow \_EntityList[i].GetPos$
10: $\quad entityPos/ = dimRatio$
11: $\quad$ **for all** c to c<nbCharges **do**
12: $\quad\quad chargePos \leftarrow entityPos + \_Charges[i].GetPos$

```
13:       if ChargeIsInSidePF(chargePos)== false then
14:         continue
15:       end if
16:       pfPos ← chargePos.X + (chargePos.Y ∗ PFDim.X)
17:       PotentialField[pfPos] ← _Charges[c]
18:     end for
19: end for
```

4.2 GPGPU

Three GPGPU versions were implemented. The first version is a nave implementation serving as a baseline to show the performance speedup gained from the improved implementations. The other two implementations use memory optimization techniques to improve performance. The optimization techniques used are coalesced memory and shared memory which will be discussed more deeply in the next sections.

4.2.1 Naive GPGPU Implementation

The naive implementation is a simple implementation with no optimizations. Each thread corresponds to a cell in the potential field. Every thread calculates the potential that each entity exerts on the corresponding cell and writes the result to the output buffer, as seen in Algorithm 3.

Algorithm 3: GPGPU Naive

```
1: struct CircleInfo
2: {
3:     float2 Position;
4:     float RadiusSquared;
5: };
6:
7: RWTexture2D<float> output;
8: StructuredBuffer<CircleInfo> EntityCircles;
9:
10: void main(uint3 dispatchThread: SV_DispatchThreadID )
11: {
12:     float potential = 0;
13: for all i to i<nbEntities do
14:     CircleInfo c = EntityCircles[i];
15:     potential += CalculateCirclePotential(c,dispatchThread.xy)
16: end for
17:       output[dispatchThread.xy] = potential
```

18: }

4.2.2 Coalesced Memory

Since global memory access have high latency(costs hundreds of clock cycles), coalescing memory access patterns is one of the most important performance optimization that can be done on a GPGPU application. On newer graphic cards global memory is accessed via 128 byte memory transactions as shown in Figure 4.2. When a warp of threads accesses global memory the data will be coalesced into one or more 128 byte memory transactions.

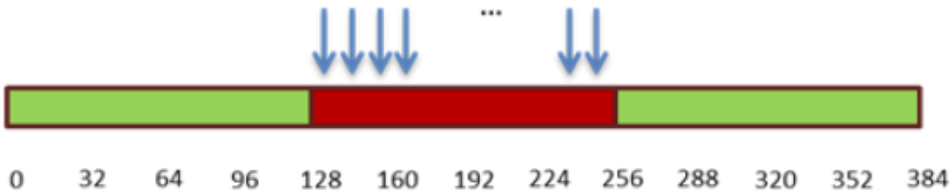

Figure 4.2: When sequential threads in a warp access memory residing in the same cash line the memory transaction will be coalesced i.e fetched in one transaction[19].

Global memory instructions write and read words of sizes equal to 1,2,4,8 or 16 bytes. If the size of a structure in global memory is 1,2,4,8 or 16 bytes the structure could be fetched in one memory transaction. If the structure has a different size, then memory transactions cannot be coalesced into one transaction and will instead be fetched with several transactions [19].

```
struct CircleInfo
{
    float2 Position;
    float RadiusSquared;
    floatPadding;
}
```

Figure 4.3: CircleInfo struct now has a padding element. The total size of CircleInfo is now 16 bytes.

To ensure that memory transactions are coalesced, the size of CircleInfo struct must be aligned. This can be done by adding a padding float to the

struct. Figure 4.3 shows the new CircleInfo struct.

4.2.3 Shared Memory

The final GPGPU implementation uses both shared memory and coalesced memory optimizations. Shared memory is an on chip memory that has lower latency than global memory and is shared between threads within a thread group. Shared memory is divided into several so called memory banks that can be accessed simultaneously by threads within a warp. If threads within a warp access different memory banks in the shared memory, the access can be performed simultaneously. If however two or more threads in a warp access the same memory bank a bank conflict will occur and the accesses are serialized[11][20][21]. The only exception is when all threads access the same bank, which will result in a broadcast. If all threads in a warp want to read the same memory bank, the data will be fetched one time and distributed to all threads in that warp. Figure 4.4 shows examples of how broadcast and bank conflict occur.

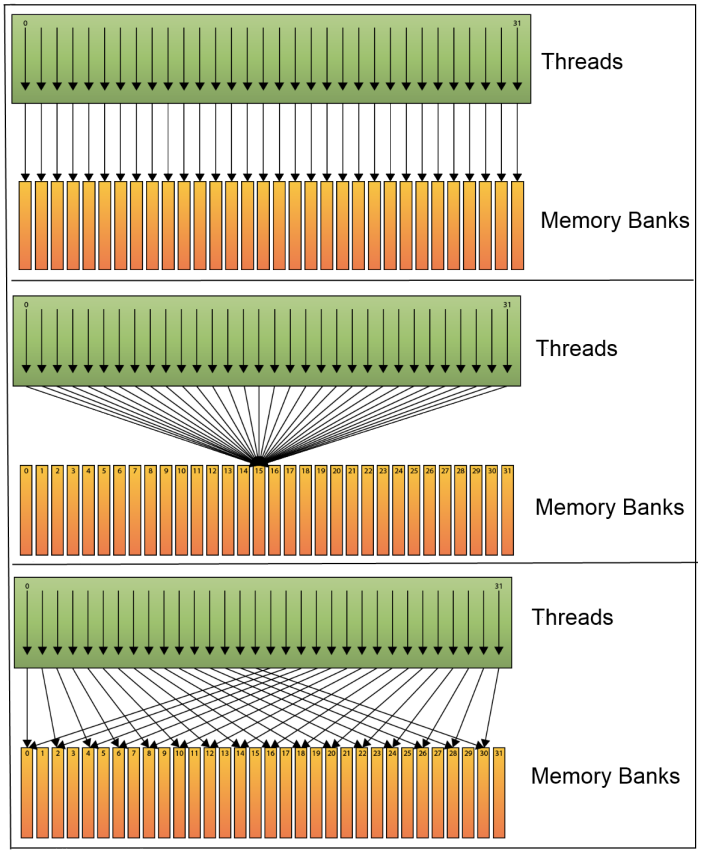

Figure 4.4: Both upper examples show shared memory acces with no bank conflict. The lower example shows a two-way bank conflict.

Algorithm 4 shows the GPGPU implementation of shared memory. The circle entities buffer is logically divided into several segments where each segment is as big as a threadgroup. The potentials of entities are calculated one segment at a time in the outer for loop.

In line 13 the shared memory array is filled with entities from one segment. Since one segment is as big as one threadgroup, each thread in the threadgroup can copy a specific entity into the shared memory array. A group barrier at line 12 ensures that all threads retrieve data from the global buffer at the same time so that memory access is simultaneous.

After the shared memory array is filled, potentials of the entities in the shared memory array are calculated. A group barrier at line 16 makes sure that all threads in the threadgroup access the same entity in the shared memory array at the same time. This will result in a broadcast since all threads access the same entity at the same time.

ALGORITHM 4: GPU Shared mem

```
 1: RWTexture2D<float> output;
 2: StructuredBuffer<CircleInfo> EntityCircles;
 3: groupShared CircleInfo shared_data[ThreadGroupSize]
 4:
 5: void main(uint3 dispatchThread: SV_DispatchThreadID,
 6:              uint3 threadID : SV_GroupThreadID)
 7: {
 8:    uint nbSegments = nbCircles / ThreadGroupSize;
 9:    float potential = 0;
10:
11: for all s to s<nbSegments do
12:   GroupBarrier
13:   shared_data[threadID.x]=EntityCircles[(s*ThreadGroupSize) +threa-
      dID.x]
14:
15:   for all i to i<ThreadGroupSize do
16:     GroupBarrier
17:     potential+=CalculateCirclePotential(shared_data[i],dispatchThread.xy)
18:   end for
19: end for
20:
21:    output[dispatchThread.xy] = potential
22: }
```

Chapter 5

Benchmark

A benchmark was made to collect execution times for the different implementations when varying three parameters: potential field resolution, number of entities and entity size. Potential field resolution start at 256x256 and are incremented by 256 up to 2048. Numbers of entities used are 64, 128, 256, then incremented by 256 up to 2048. The entity type used in this experiment is a circle entity. Three different sizes of circles were used: small with a radius of 4, medium with a radius of 8 and large with a radius of 15.

The benchmark executes all combinations of the parameters and calculates the execution time and speedup achieved for each combination. The speedup refers to performance gains between the CPU Memory Copy and GPU Shared Memory implementations.

The results are presented in three sub-chapters. The first two will show an overview of the execution times when varying potential field resolution and number of entities. The last sub-chapter shows a more in depth comparison between the CPU Memory Copy and GPU Shared Memory implementations.

The CPU Naive implementation had very poor performance and have therefore been omitted from the graphs in the following sub-chapters in order to maintain readability. Execution times and standard deviations for all implementations can however be found in Appendix A.

5.1 Hardware Specifications

The benchmark was executed on a high-end computer. Table 5.1 shows the hardware specifications of the computer.

Operating system	Windows 7 Professional N
CPU	Intel Core i7 920, 267 GHz
RAM	6,00 GB
GPU	Nvidia Geforce GTX 660 2GB GDDR5

Table 5.1: Hardware Specifications

5.2 Validation of Results

Two measures were taken to ensure that the results were stable and valid. The first measure is to execute each setup of parameters 10 times and calculate the average to get the execution time. The second measure is to calculate the standard deviation of the results for each execution in order to make better speedup analysis for the implementations.

5.3 Varying Potential Field Resolution

5.3.1 Small Sized Entities

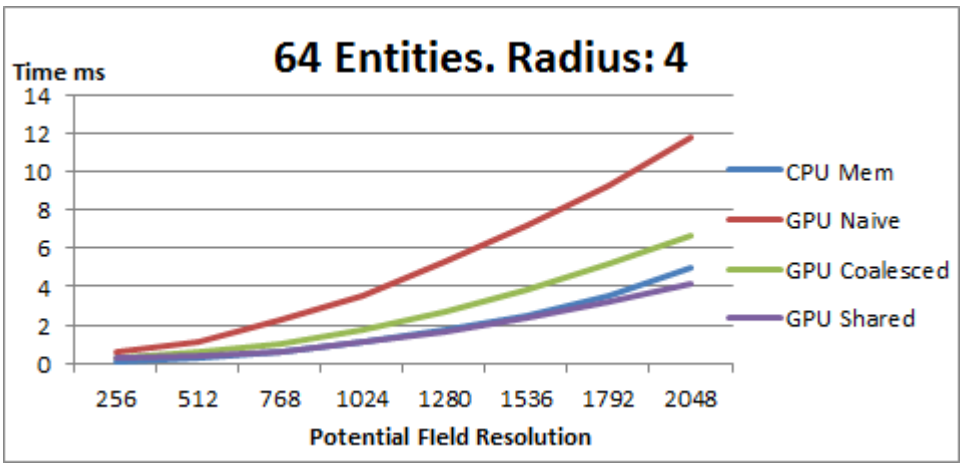

Figure 5.1: Execution times when using 64 small entities and varying the resolution of the potential field.

When the number of entities is 64 the execution times of CPU Memory copy and GPU Shared Memory implementations are very close. In Figure 5.1 we can see that the GPU Shared Memory implementations gets slightly faster than the CPU implementation when the potential field resolution is higher than 1792.

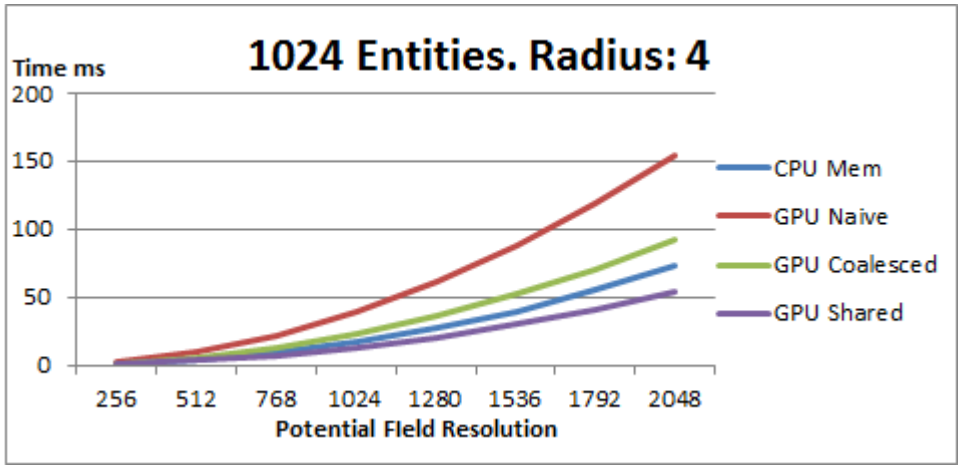

Figure 5.2: Execution times when using 1024 small entities and varying the resolution of the potential field.

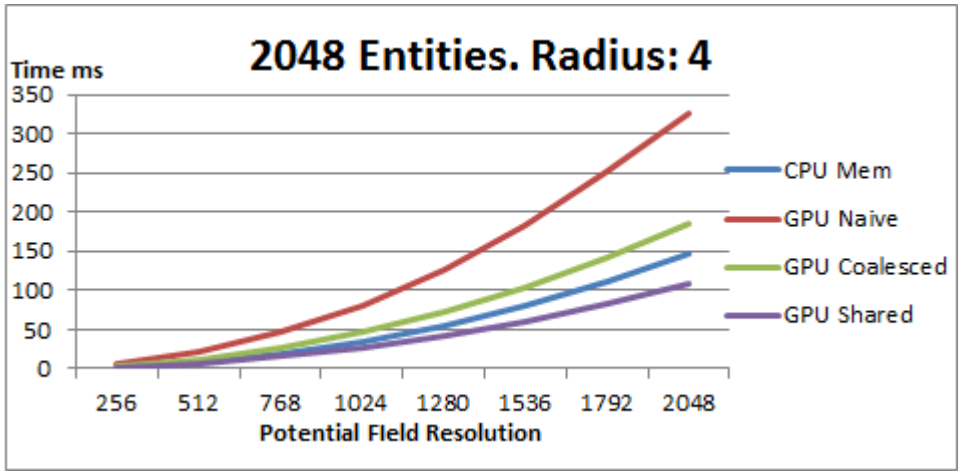

Figure 5.3: Execution times when using 2048 small entities and varying the resolution of the potential field.

When the number of entities increases to 1024 and 2048 the CPU Memory Copy implementation is still faster than the GPU Naive and GPU Coalesced implementations. We can however see in Figures 5.2 and 5.3 that when the potential field resolution exceeds 1024 the GPU Shared Memory implementation starts to get slightly faster than the CPU Memory Copy.

5.3.2 Medium Sized Entities

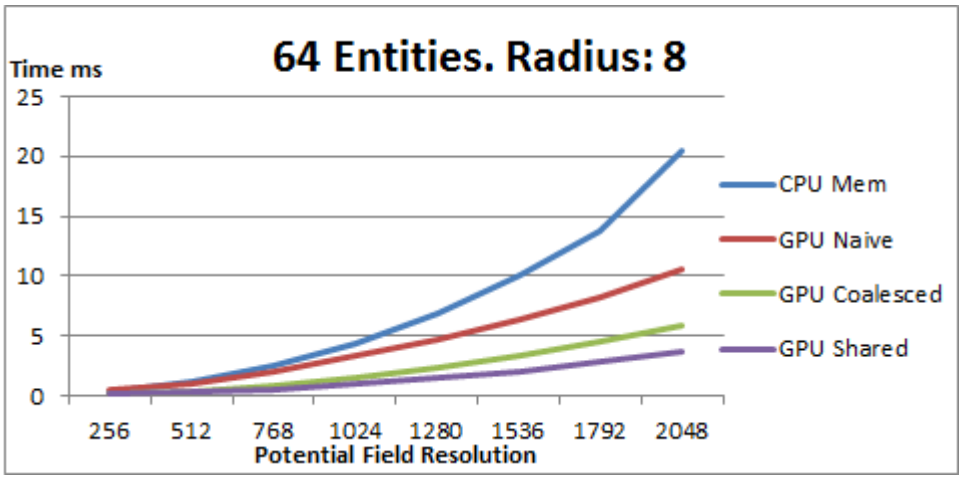

Figure 5.4: Execution times when using 64 medium sized entities and varying the resolution of the potential field.

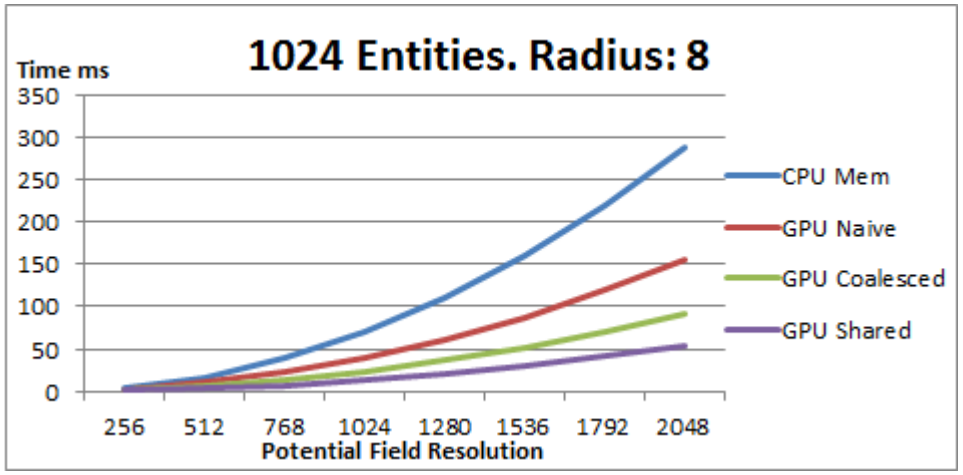

Figure 5.5: Execution times when using 1024 medium sized entities and varying the resolution of the potential field.

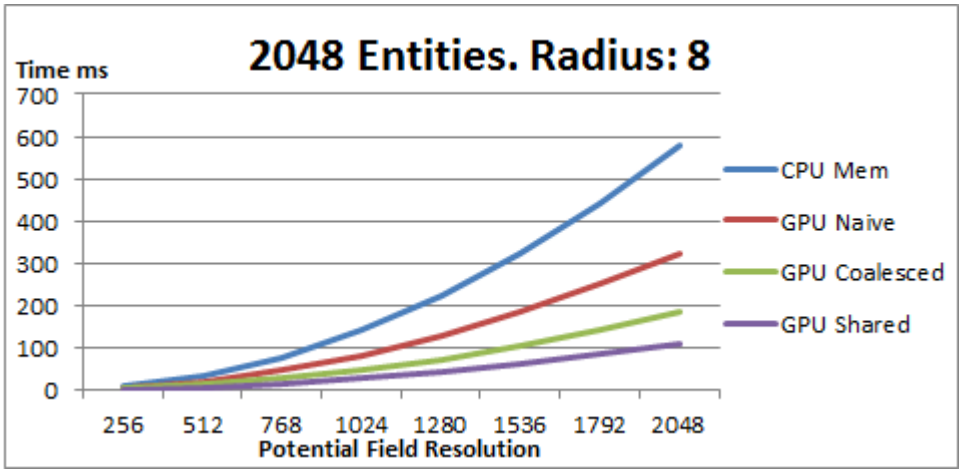

Figure 5.6: Execution times when using 2048 medium sized entities and varying the resolution of the potential field.

In Figures 5.4 to 5.6 we can see that when using entities with radius of 8 the CPU Memory implementation is always slower than all the GPU implementations. The GPU Shared Memory implementation was the fastest implementation for all variations of number of entities.

5.3.3 Large Sized Entities

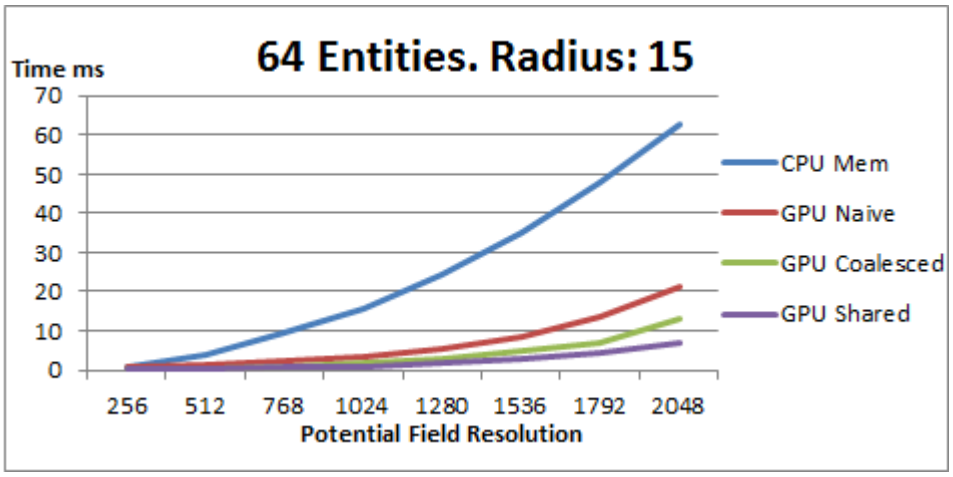

Figure 5.7: Execution times when using 64 large entities and varying the resolution of the potential field.

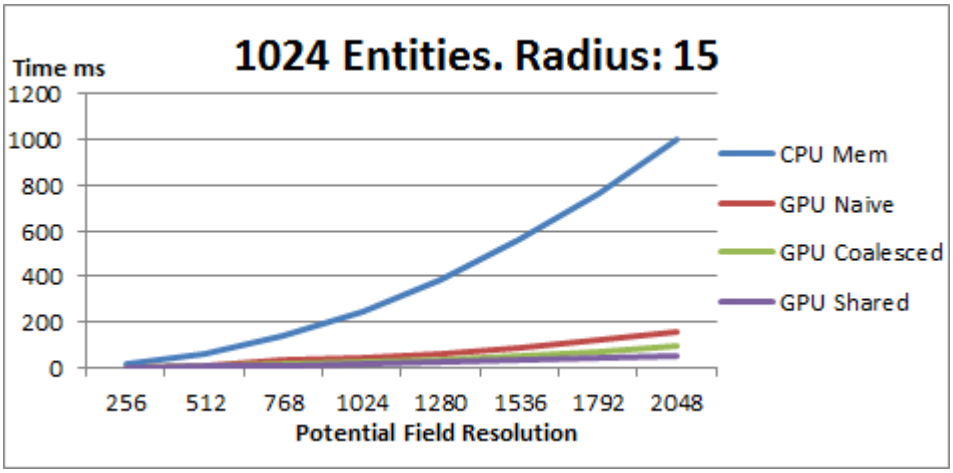

Figure 5.8: Execution times when using 1024 large entities and varying the resolution of the potential field.

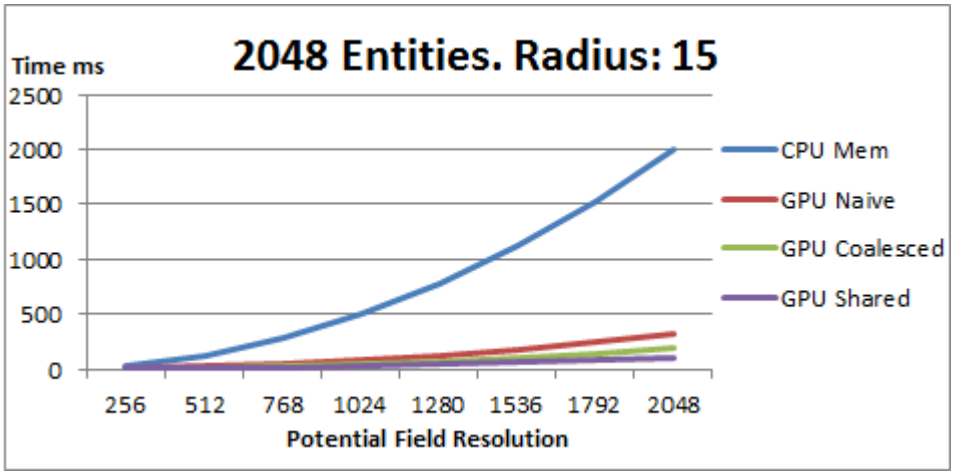

Figure 5.9: Execution times when using 2048 large entities and varying the resolution of the potential field.

In Figures 5.7, 5.8 and 5.9 we can see that when using the largest entities the CPU Memory Copy implementation is slower than all three GPU implementations.

5.4 Varying Number of Entities

5.4.1 Small Sized Entities

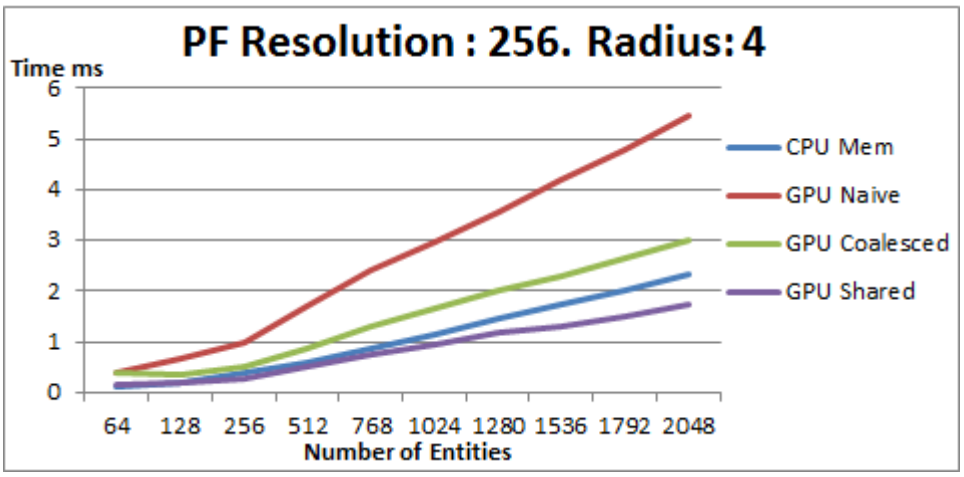

Figure 5.10: Execution times when using potential field resolution of 256x256 with small entities.

As we can see in Figure 5.10 the execution times for the CPU Memory Copy and GPU Shared Memory implementations are close to each other. When the number of entities increases beyond 768 the GPU Shared Memory becomes faster than the CPU Memory Copy implementation.

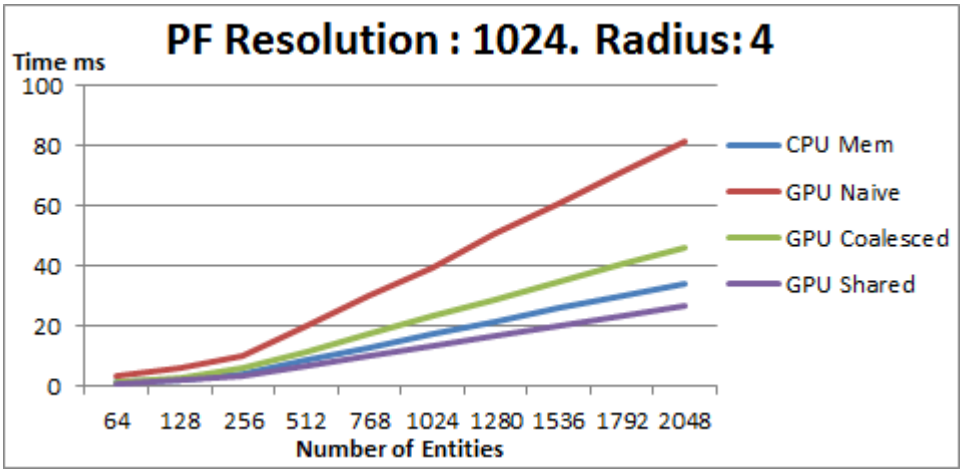

Figure 5.11: Execution times when using potential field resolution of 1024x1024 with small entities.

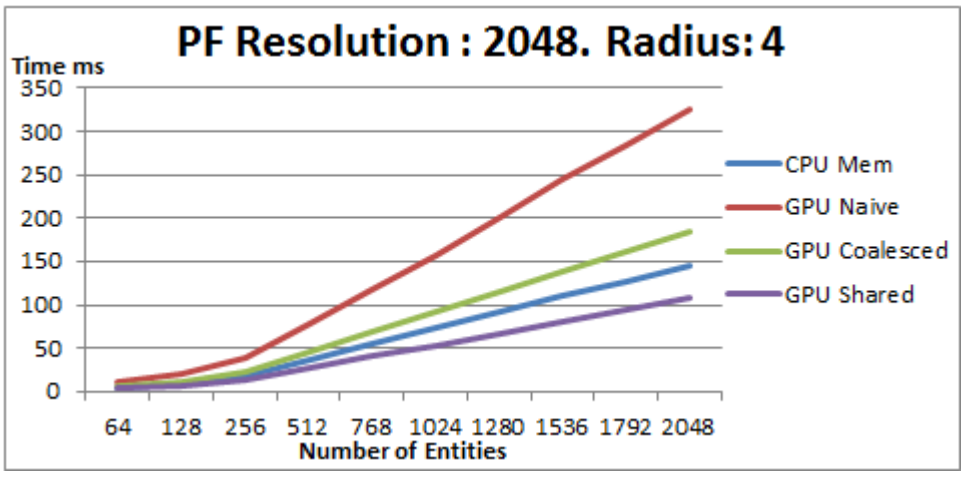

Figure 5.12: Execution times when using potential field resolution of 2048x2048 with small entities.

As we can see in Figures 5.10 to 5.12 the CPU Memory Copy implementation is faster than the GPU Naive and GPU Coalesced implementations. In Figures 5.11 and 5.12 we can see that when the number of entities increases beyond 512 GPU Shared Memory becomes faster than the CPU Memory Copy implementation.

5.4.2 Medium Sized Entities

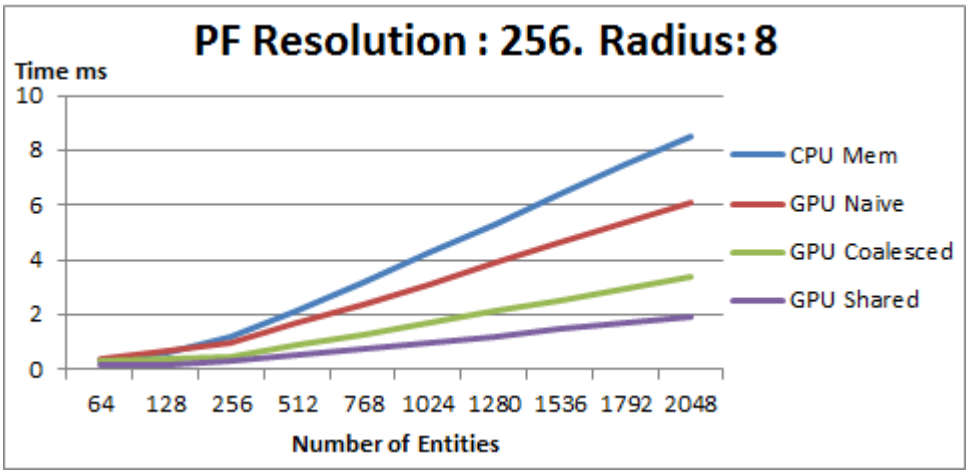

Figure 5.13: Execution times when using potential field resolution of 256x256 with medium sized entities.

In Figure 5.13 we can see that the execution time for CPU Memory Copy implementation is close to the GPU Naive implementation. When the number of entities is bigger than 256 the GPU Naive implementation becomes faster than the CPU Memory Copy.

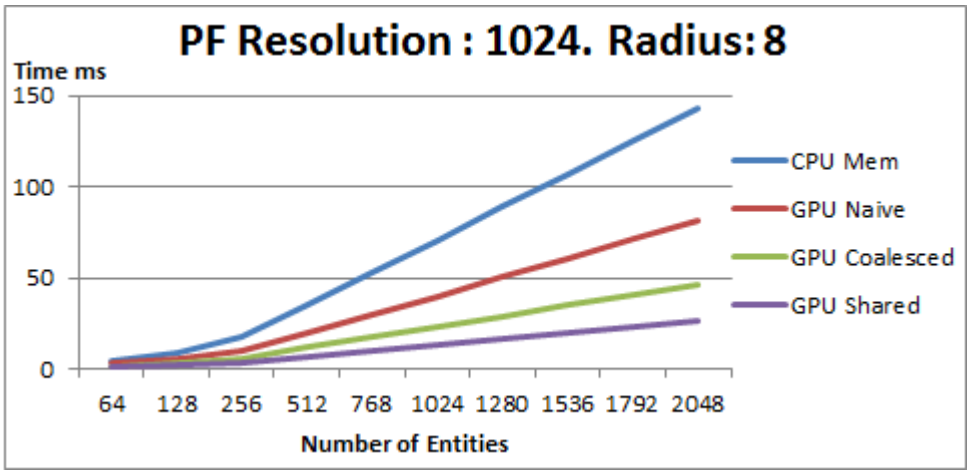

Figure 5.14: Execution times when using potential field resolution of 1024x1024 with medium sized entities.

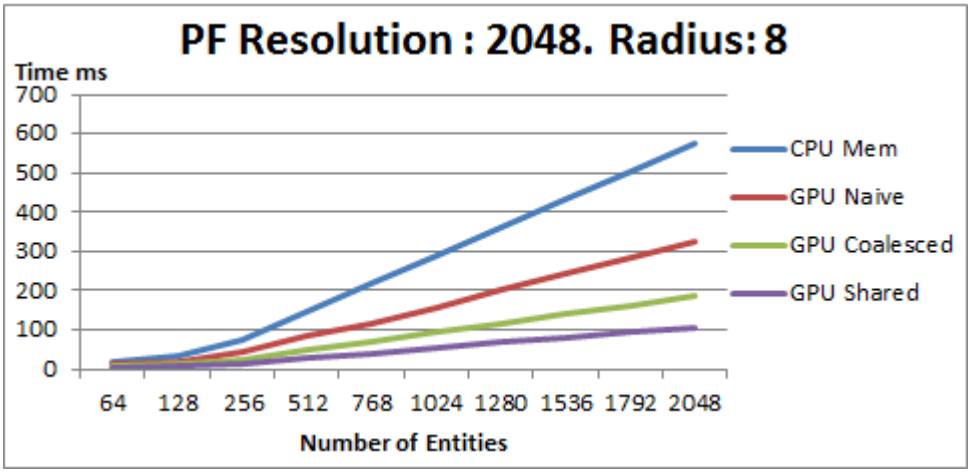

Figure 5.15: Execution times when using potential field resolution of 2048x2048 with medium sized entities.

In Figures 5.13 to 5.15 we can see that when the size of entities is increased to 8 the CPU Memory Copy implementation becomes slower than all GPU implementations. It is also clear that the GPU Shared Memory implementation is the fastest in all cases.

5.4.3 Large Sized Entities

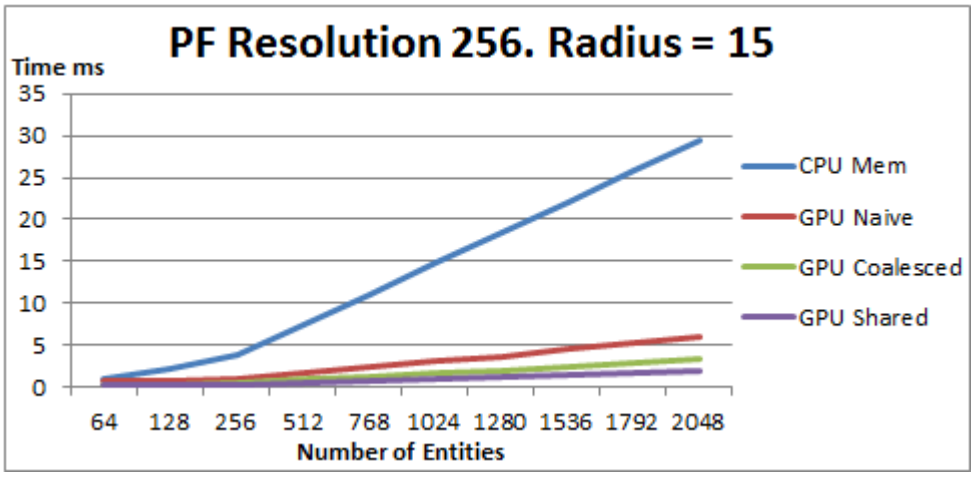

Figure 5.16: Execution times when using potential field resolution of 256x256 with large entities.

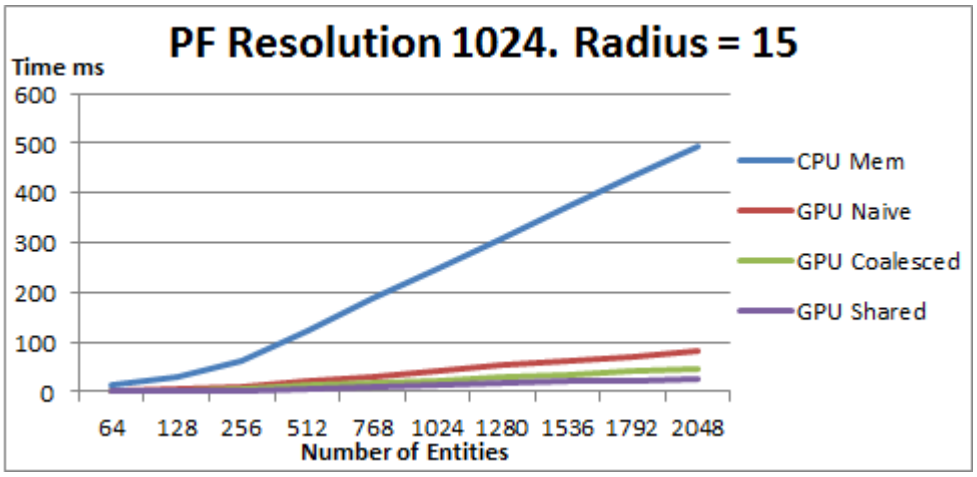

Figure 5.17: Execution times when using potential field resolution of 1024x1024 with large entities.

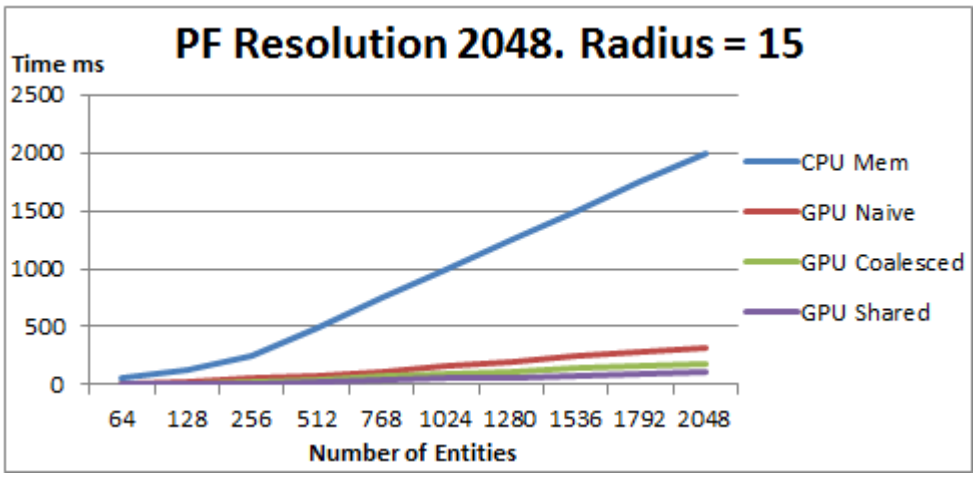

Figure 5.18: Execution times when using potential field resolution of 2048x2048 with large entities.

As we can see in Figures 5.16 to 5.18 all three GPU implementations are faster than the CPU Memory Copy implementation.

5.5 Calculating Speedup With Different Parameters

A range calculation has been made in order to get a correct estimate of the speedup achieved with using the GPU Shared Memory implementation. The benchmark runs every set of parameters 10 times and produces a mean execution time as well as the standard deviation. Figure 5.19 shows an overview of the basic idea of how the range is calculated. Three standard deviations are added and subtracted from the mean values of the CPU Memory Copy and GPU Shared Memory implementations. Three standard deviations was chosen to capture 99.7% of all values in accordance with the 68-95-99.7 rule. As we can see in Figure 5.19 S1 represents the smallest difference in execution time between the CPU Memory Copy and the GPU Shared Memory implementations, and S2 represents the biggest difference. In Figures 5.20 to 5.22 the lines in the graphs represent the mean value of each execution, and the error bars represent the speedup range.

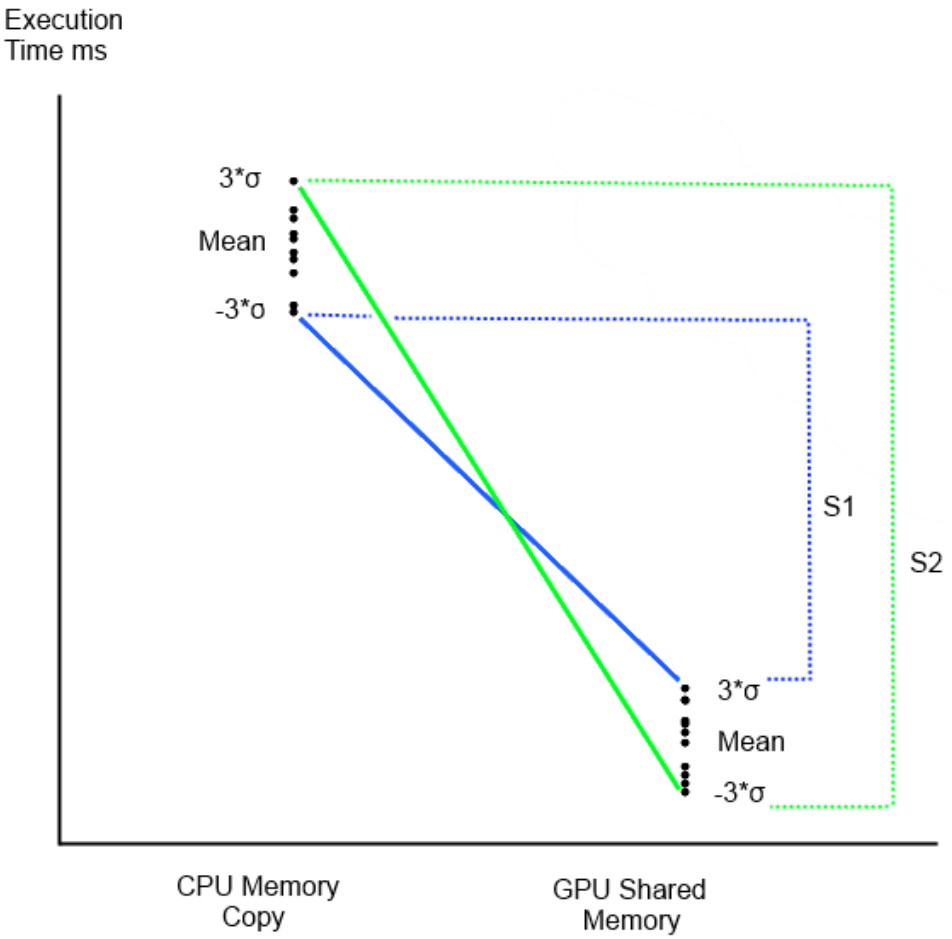

Figure 5.19: Mean values of execution times are off-setted by three standard deviations to capture the full range of speedup achieved.

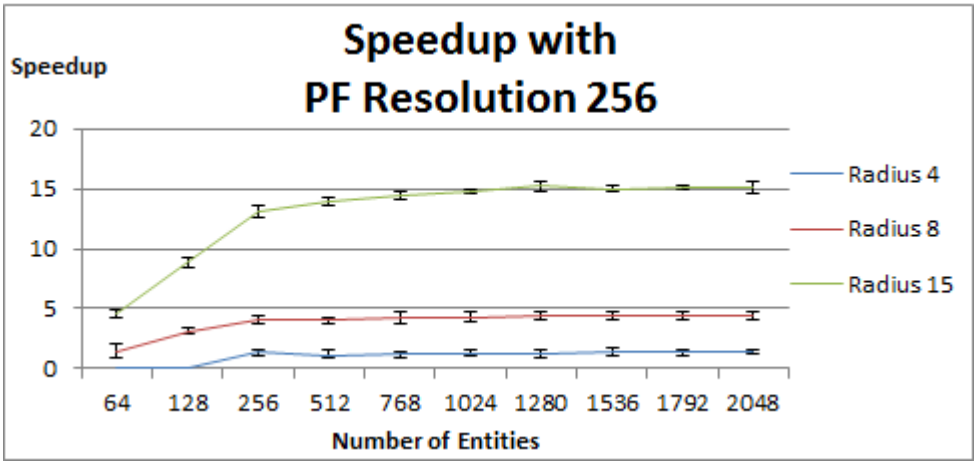

Figure 5.20: Speedup achieved for different entity sizes with potential field resolution of 256x256.

In Figure 5.20 we can see that when the number of entities is 64 or 128, no speedup is achieved for entities with radius of 4. When the radius is 8 or 15 the speedup is small for smaller number of entities. When the number of entities increases beyond 128 the speedup increases and stabilizes at some point for the different radius's.

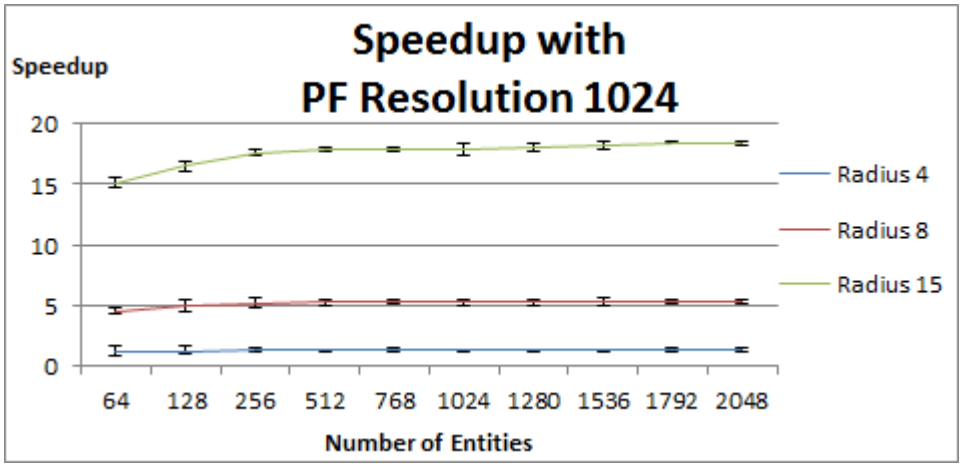

Figure 5.21: Speedup achieved for different entity sizes with potential field resolution of 1024x1024.

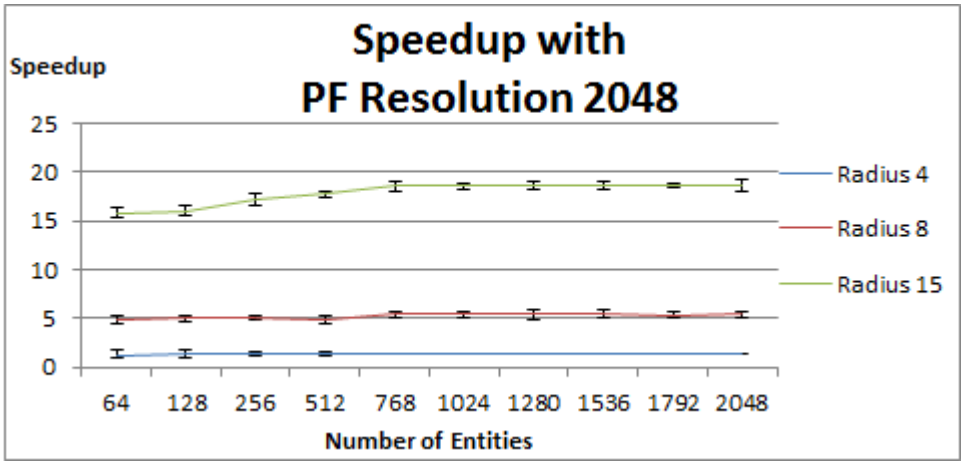

Figure 5.22: Speedup achieved for different entity sizes with potential field resolution of 2048x2048.

The biggest speedup gains can be seen when varying the size of the entities used. Figures 5.20 to 5.22 show that when using a specific potential field resolution while varying the number of entities the speedup is relatively constant, but the different radius sizes give big differences in speedup.

Chapter 6

Discussion

In chapters 5.3 to 5.5 we saw the impact that the different parameters had on the performance of the various implementations. It was clear that when using the smallest entity size the CPU Memory Copy implementation performed better than both the GPU Naive and GPU Coalesced memory implementations. The GPU Shared Memory implementation performed slightly better than CPU Memory Copy when using small entities.

The Figures 5.20 to 5.22 clearly shows that the biggest performance speedup was gained when entity size increased. The speedup more than doubled between each step of radius increase (from radius of 4 to 8, and from 8 to 15). The results show that the size of entities used is a more important factor than potential field resolution or number of entities.

When the potential field resolution was 256x256 and having 128 small entities, CPU Memory Copy was marginally faster than the GPU Shared Memory implementation. The highest speedup that was achieved ranged between 18.1x 19.3x with potential field resolution of 2048x2048 and with 2048 large entities. It is important to note that the biggest performance gains were achieved with entity radius of 15 which is quite big for a world with a size of 256x256. But even with smaller entities with 8 in radius, a speedup that ranged between 4.5 and 5.5 was achieved.

Potential fields have been used in games, mostly in the RTS genre [5]. In Age of Empires the potential field was used as a terrain analysis tool and could be slow at times [16]. A GPGPU based implementation could speed up the creation phase of potential fields and ultimately lead to faster iteration times for level designers that use potential fields as a terrain editing tool.

The GPU implementations presented in this thesis are fairly simple to implement and it is easy to add new types of entities to use in the application.

This implementation serves as a base for other areas to build on such as crowd control, terrain editing or terrain debugging.

6.1 Conclusion

The research question posed in this thesis was : *How much performance speedup can be achieved when moving the computation of potential fields from the CPU to the GPU when optimizing memory access patterns?* To answer the question five different implementations for potential field have been implemented and benchmarked. Two of them were CPU and three where GPGPU implementations. The first CPU implementation was a naive implementation. The second CPU implementation was an improvement which copied a set of charges onto the potential field instead of looping through all potential field cells and calculating each charge. The first GPGPU implementation was a naive version, and the other two used optimized memory access patterns(coalesced memory transactions and usage of shared memory).

Three different parameters were modified when benchmarking the implementations, potential field resolution, number of entities and size of entities. For low potential field resolution with low number of small entities the CPU Memory Copy implementation was slightly faster than the GPU Shared Memory implementation. The highest speedup that was achieved ranged between 18.1x and 19.3x. This speedup was achieved when the potential field had a resolution of 2048x2048 and 2048 entities with 15 in radius.

Whether or not it is worth implementing a GPGPU solution depends on the entities that are going to be used in the application. In this experiment a circle is used as an entity, but the entity can be of any form and size and using different entities will affect the performance. Another important factor to take into consideration is whether the application is CPU or GPU bound. If the CPU is overburdened it can be beneficial to use a GPGPU implementation even for cases when the CPU implementation is faster in order to make better use of the computer resources.

6.2 Future Work

The experiment in this thesis was done with large potential fields that covered big portions of space. There are other ways one could use potential fields and one way is to use several smaller potential fields that only cover specific areas of the total space. As mentioned in chapter 2, Killzone uses

potential fields in this way. Only small potential fields are created to cover the areas where the AI is at the time.

It would be interesting to see if a GPGPU implementation for several smaller potential fields would be effective. In this thesis data was sent to the GPU only once to create the whole potential field. Having several smaller potential fields would mean executing the GPGPU program several times and send smaller amounts of data each time. Since sending data to the GPU have some overhead it may cause the GPGPU implementation to become slower than the CPU.

Bibliography

[1] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *The international journal of robotics research*, 1986.

[2] R. C. Arkin, "Motor schemabased mobile robot navigation," *The International Journal of Robotics Research*, 1989.

[3] J. Borenstein and Y. Koren, "Real-time obstacle avoidance for fast mobile robots.," *Systems, Man and Cybernetics, IEEE Transactions on.*, 1989.

[4] A. L. Zobrist, "A model of visual organization for the game of GO.," *Proceedings of the May 14-16, 1969, spring joint computer conference*, 1969.

[5] J. Hagelbck and S. J. Johansson, "The rise of potential fields in real time strategy bots.," *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE).*, 2008.

[6] J. D. Owens, "A survey of generalpurpose computation on graphics hardware," *Computer graphics forum*, 2007.

[7] O. Gervasi, D. Russo, and F. Vella, "The AES implementation based on opencl for mult/many core architeture cryptography.," *Computational Science and Its Applications International Conference on, IEEE*, 2010.

[8] X. Jin, J. Xu, C. C. Wang, S. Huang, and J. Zhang, "Interactive control of large-crowd navigation in virtual environments using vector fields.," *Computer Graphics and Applications, IEEE*, 2008.

[9] J. Guivant and S. Cossell, "Parallel evaluation of a spatial traversability cost function on GPU for efficient path planning.," *Open Journal of Urology*, 2011.

[10] J. Hagelbck and S. J. Johansson, "Using multi-agent potential fields in real-time strategy games.," *In Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, 2008.

[11] Nvidia, "Cuda c best practices guide." http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. Accessed 2013-02-11.

[12] S. J. Johansson, "A survey of the use of artificial potential fields and influence maps in game AI research.," *Accepted for publication in IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

[13] J. Hagelbck, "Potential-field based navigation in starcraft.," *Computational Intelligence and Games (CIG), 2012 IEEE Conference*, 2012.

[14] C. Miles and S. J. Louis, "Towards the co-evolution of influence map tree based strategy game players.," *Computational Intelligence and Games,*, 2006.

[15] P. H. Ng, Y. J. Li, and S. C. Shiu, "Unit formation planning in RTS game by using potential field and fuzzy integral.," *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*, 2011.

[16] D. C. Pottinger, "Terrain analysis in realtime strategy games.," *InProceedings of Computer Game Developers Conference.*, 2000.

[17] W. Straatman, R.and van der Sterren and A. Beij, "Killzones AI: dynamic procedural combat tactics.," *Game Developers Conference.*, 2005.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture.," *Micro, IEEE*, 2008.

[19] Nvidia, "Cuda c programming guide." http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 2013-02-11.

[20] M. Harris, S. Sengupta, and J. Owens, *GPU Gems 3: Chapter 39.* Addison-Wesley Professional, 2007.

[21] J. Fung, "Directcompute lecture series 210: Gpu optimizations and performance." http://channel9.msdn.com/Blogs/gclassy/DirectCompute-Lecture-Series-210-GPU-Optimizations-and-Performance. Accessed 2013-02-13.

Appendix A

Results

The Results are shown in the tables on the following pages. There are three tables, one table for each entity size that was used. The first five columns of each table show the mean execution time as well as the standard deviation for each individual implementation in milliseconds seperated by "/" . The last two columns show the combination of potential field granularity and number of entities for each given execution.

Table A.1: Results for entities with radius = 15.

CPU Naive	CPU Mem	GPU Naive	GPU Coalesced	GPU Shared	Resolution	Entities
59.47/0.015	0.96/0.002	0.84/0.099	0.31/0.079	0.21/0.006	256	64
232.64/0.337	4.15/0.038	1.12/0.093	0.47/0.066	0.31/0.007	512	64
521.11/0.507	9.32/0.024	2.17/0.223	0.99/0.086	0.62/0.003	768	64
954.57/0.170	15.43/0.033	3.58/0.181	1.70/0.099	1.06/0.008	1024	64
1517.76/0.285	24.34/0.400	5.37/0.380	2.71/0.260	1.69/0.184	1280	64
2230.73/0.447	35.18/0.140	8.67/0.241	4.76/0.194	2.65/0.198	1536	64
2847.58/0.289	48.04/0.089	13.71/0.993	6.91/0.250	3.67/0.231	1792	64
3783.88/0.439	62.71/0.189	21.31/1.526	13.10/0.594	4.79/0.309	2048	64
119.99/0.171	1.88/0.053	0.81/0.033	0.32/0.526	0.60/0.005	256	128
493.23/0.230	7.72/0.023	1.80/0.094	0.87/0.043	0.54/0.043	512	128
1107.24/0.388	17.43/0.046	3.79/0.344	1.96/0.178	1.18/0.028	768	128
1961.65/1.285	35.33/0.190	8.21/0.545	4.33/0.155	2.60/0.110	1024	128
3062.39/0.254	48.28/0.400	16.75/2.680	9.90/0.057	5.38/0.094	1280	128
4039.96/0.223	70.49/0.134	33.73/1.632	18.85/0.891	11.25/0.052	1536	128
5968.61/0.559	95.69/0.083	49.90/0.365	28.21/0.224	16.80/0.247	1792	128
7796.91/0.481	125.71/0.225	23.83/7.172	13.80/0.237	8.20/0.444	2048	128
244.18/0.065	3.69/0.013	1.00/0.051	0.47/0.020	0.29/0.020	256	256
922.81/0.071	15.37/0.038	3.18/0.092	1.69/0.029	1.01/0.028	512	256
2061.00/0.126	34.70/0.056	6.74/0.118	3.74/0.083	2.21/0.082	768	256
3636.97/0.106	62.03/0.128	11.74/0.137	6.58/0.098	3.89/0.097	1024	256
5587.90/0.142	97.70/0.313	17.89/0.095	10.24/0.090	6.03/0.094	1280	256
8716.00/0.533	141.08/0.668	25.33/0.148	14.73/0.160	8.66/0.156	1536	256
12057.19/4.746	191.09/0.497	34.19/0.188	20.03/0.195	11.78/0.194	1792	256
14772.60/6.145	250.31/0.545	44.42/0.294	26.14/0.237	15.35/0.236	2048	256
473.48/0.066	7.36/0.023	1.71/0.057	0.88/0.022	0.53/0.020	256	512
1907.14/0.105	30.81/0.049	5.93/0.082	3.34/0.031	1.95/0.033	512	512
4392.49/3.766	69.75/0.108	15.13/0.587	8.75/0.931	5.10/0.535	768	512
7310.89/5.509	123.98/0.263	31.89/0.383	18.31/0.797	10.67/0.451	1024	512
12045.14/1.261	197.90/0.375	40.62/3.974	23.86/0.358	14.81/0.275	1280	512
16414.42/1.605	282.46/0.212	49.91/0.661	29.41/0.576	17.51/0.604	1536	512
23189.52/1.767	382.62/1.076	60.44/1.552	35.76/1.017	20.76/0.383	1792	512
30782.90/0.258	500.44/0.844	78.08/0.286	46.30/0.221	26.94/0.227	2048	512
683.22/0.081	11.04/0.029	2.17/0.052	1.16/0.060	0.69/0.062	256	768
2783.49/0.183	46.27/0.094	7.74/0.089	4.43/0.062	2.58/0.062	512	768
6203.32/0.229	104.82/0.547	22.56/1.974	13.20/1.136	7.73/0.540	768	768
11805.23/1.300	185.53/0.389	48.89/0.674	28.49/0.165	16.65/0.262	1024	768
18295.94/0.240	291.26/0.782	52.89/0.342	31.15/0.186	18.13/0.521	1280	768
26396.77/0.125	473.03/1.573	66.42/1.674	39.26/0.899	22.77/0.288	1536	768
34196.75/1.131	574.02/1.284	89.54/0.176	53.19/0.189	30.90/0.183	1792	768
47108.58/7.476	750.93/1.300	116.66/0.047	69.40/0.226	40.33/0.222	2048	768

CPU Naive	CPU Mem	GPU Naive	GPU Coalesced	GPU Shared	Resolution	Entities
955.24/0.114	14.73/0.012	2.79/0.068	1.53/0.021	0.89/0.019	256	1024
3831.37/0.803	61.63/0.102	10.75/1.437	6.33/0.925	3.65/0.054	512	1024
8135.31/0.239	140.17/0.412	33.80/1.493	19.85/0.805	11.48/0.168	768	1024
14570.81/0.321	247.41/0.364	45.02/0.445	26.46/0.930	15.37/0.215	1024	1024
22816.93/0.326	388.54/0.841	61.77/1.489	36.52/0.970	21.19/0.465	1280	1024
34343.75/0.359	562.92/1.664	88.01/1.946	52.23/0.364	30.68/0.842	1536	1024
44101.41/1.602	766.16/1.166	119.20/0.316	70.91/0.231	41.13/0.179	1792	1024
59103.40/1.491	1000.75/1.498	155.38/0.454	92.56/0.166	53.74/0.225	2048	1024
1128.58/0.018	18.43/0.027	3.52/0.051	1.91/0.064	1.11/0.062	256	1280
4649.93/0.307	76.75/0.107	15.22/0.561	8.59/0.464	4.94/0.084	512	1280
10803.60/0.229	201.50/0.438	43.12/0.371	24.53/0.346	14.16/0.076	768	1280
19335.90/0.316	308.80/0.690	54.66/0.249	31.02/0.301	17.93/0.376	1024	1280
30604.32/3.105	491.94/0.272	79.47/0.849	45.35/0.499	26.31/0.322	1280	1280
42201.04/0.360	789.94/2.725	113.60/0.092	65.07/0.154	37.75/0.146	1536	1280
55249.77/0.913	956.87/1.730	154.89/0.798	88.75/0.351	49.44/0.198	1792	1280
73298.16/3.845	1252.98/2.094	201.88/0.960	115.69/0.115	64.15/0.261	2048	1280
1406.87/0.104	22.09/0.065	4.21/0.051	2.30/0.069	1.34/0.063	256	1536
5612.73/0.129	92.03/0.162	18.76/2.718	10.61/1.564	6.10/0.089	512	1536
12157.93/0.495	242.00/0.545	47.14/0.142	26.93/0.105	15.54/0.092	768	1536
23229.90/1.408	370.94/0.556	62.11/0.288	35.36/0.398	20.50/0.483	1024	1536
33631.77/1.236	584.02/0.864	94.96/0.119	54.25/0.128	31.46/0.121	1280	1536
51134.13/1.870	945.24/2.869	136.46/0.556	78.13/0.221	48.30/0.163	1536	1536
66061.47/2.297	1149.01/2.308	186.30/1.553	106.39/0.362	60.70/0.235	1792	1536
86510.30/1.167	1501.94/2.134	242.66/1.021	138.77/0.079	79.55/0.163	2048	1536
1592.86/0.125	25.75/0.046	4.83/0.121	2.63/0.010	1.52/0.029	256	1792
6546.98/0.267	107.60/0.155	19.11/1.295	10.84/0.740	6.23/0.042	512	1792
15130.07/1.040	282.52/0.335	52.93/1.295	30.23/0.764	17.45/0.466	768	1792
25514.16/1.343	432.71/0.767	72.59/0.484	41.36/0.510	23.96/0.519	1024	1792
40287.54/1.054	681.22/1.390	110.78/0.137	63.26/0.132	36.68/0.122	1280	1792
59769.79/0.324	1104.43/2.855	159.54/1.051	91.24/0.372	52.83/0.164	1536	1792
78623.58/2.069	1339.86/2.134	218.00/1.757	124.10/0.257	72.15/0.320	1792	1792
102588.00/2.984	1751.46/2.026	283.55/1.294	161.88/0.067	94.13/0.323	2048	1792
1895.78/0.103	29.50/0.049	5.47/0.146	3.00/0.025	1.73/0.023	256	2048
7831.87/1.088	124.25/0.678	20.89/0.959	11.86/0.627	7.35/0.184	512	2048
16605.14/2.218	278.73/0.449	52.34/0.480	29.89/0.024	16.23/0.200	768	2048
29454.26/1.978	495.08/0.999	81.64/0.953	46.56/0.156	26.95/0.576	1024	2048
46295.56/1.838	778.60/2.794	126.74/0.462	72.38/0.293	41.95/0.167	1280	2048
65378.24/15.292	1123.80/2.841	182.69/1.353	104.18/0.346	60.39/0.181	1536	2048
103790.37/12.716	1530.87/2.276	250.12/2.310	141.69/0.185	82.56/0.357	1792	2048
137131.60/17.618	2000.29/2.975	324.64/2.517	185.02/0.231	106.66/0.392	2048	2048

Table A.2: Results for entities with radius = 8.

CPU Naive	CPU Mem	GPU Naive	GPU Coalesced	GPU Shared	Resolution	Entities
57.60/0.019	0.30/0.012	0.50/0.030	0.21/0.002	0.12/0.005	256	64
234.13/0.285	1.10/0.046	1.05/0.079	0.41/0.028	0.28/0.006	512	64
514.23/0.386	2.45/0.067	1.95/0.118	0.86/0.015	0.57/0.071	768	64
933.77/0.196	4.41/0.076	3.33/0.174	1.53/0.098	0.96/0.051	1024	64
1518.35/0.324	6.95/0.015	4.62/0.062	2.38/0.199	1.49/0.094	1280	64
2118.66/0.566	10.02/0.011	6.31/0.153	3.43/0.241	2.10/0.088	1536	64
2838.36/0.233	14.80/0.036	8.20/0.031	4.48/0.089	2.85/0.031	1792	64
3839.67/0.523	20.50/0.065	10.59/0.287	5.96/0.364	3.71/0.037	2048	64
118.67/0.191	0.59/0.037	0.62/0.079	0.25/0.045	0.17/0.036	256	128
485.20/0.200	2.18/0.011	1.66/0.080	0.79/0.048	0.49/0.016	512	128
1048.81/0.449	4.93/0.018	3.31/0.117	1.70/0.072	1.04/0.069	768	128
1880.20/1.574	8.92/0.016	5.74/0.170	2.98/0.107	1.80/0.104	1024	128
2925.13/0.253	13.86/0.041	8.82/0.532	4.83/0.308	2.95/0.237	1280	128
4360.94/0.284	20.12/0.049	13.31/0.184	7.56/0.269	4.55/0.261	1536	128
5891.99/0.463	27.71/0.067	17.77/0.270	10.26/0.324	6.17/0.361	1792	128
7206.43/0.484	40.86/0.067	22.88/0.029	13.38/0.410	8.02/0.400	2048	128
227.39/0.050	1.17/0.038	1.03/0.075	0.47/0.031	0.30/0.014	256	256
963.14/0.088	4.34/0.061	3.23/0.076	1.73/0.051	1.03/0.052	512	256
2032.05/0.154	9.82/0.043	6.82/0.130	3.81/0.091	2.25/0.072	768	256
3936.11/0.090	17.78/0.036	11.96/0.171	6.68/0.112	3.95/0.107	1024	256
5755.68/0.135	27.84/0.091	18.09/0.075	10.36/0.017	6.16/0.214	1280	256
8361.78/0.453	40.26/0.076	25.68/0.175	14.99/0.310	8.89/0.300	1536	256
11117.01/3.910	55.26/0.136	34.63/0.272	20.42/0.349	12.00/0.320	1792	256
15371.90/5.234	71.98/0.119	45.04/0.368	26.62/0.427	15.65/0.397	2048	256
452.83/0.081	2.15/0.073	1.75/0.119	0.89/0.046	0.54/0.034	256	512
1849.30/0.133	8.51/0.010	6.00/0.085	3.39/0.049	1.98/0.050	512	512
4166.44/4.640	19.34/0.028	13.05/0.124	7.54/0.147	4.45/0.018	768	512
7712.56/5.466	35.56/0.046	23.06/0.106	13.29/0.172	7.81/0.197	1024	512
11711.02/0.982	55.64/0.252	34.73/0.429	20.35/0.266	11.87/0.307	1280	512
17661.14/1.638	80.30/0.179	49.22/0.178	29.11/0.264	16.93/0.201	1536	512
23213.12/1.957	110.26/0.273	61.33/2.826	36.35/1.820	21.17/0.046	1792	512
30547.85/0.239	144.23/0.281	78.13/0.345	46.44/0.389	27.03/0.367	2048	512
740.89/0.080	3.21/0.023	2.18/0.091	1.16/0.011	0.69/0.033	256	768
2839.64/0.154	12.78/0.012	7.74/0.099	4.44/0.048	2.61/0.066	512	768
6169.74/0.216	29.06/0.069	16.94/0.131	9.86/0.070	5.73/0.068	768	768
11757.54/1.341	53.32/0.062	29.97/0.091	17.44/0.160	10.16/0.155	1024	768
17872.85/0.234	83.48/0.180	46.17/0.094	27.14/0.016	15.89/0.224	1280	768
25263.01/0.117	120.33/0.247	66.03/0.132	39.13/0.260	22.88/0.310	1536	768
36238.79/1.131	164.68/0.360	89.58/0.287	53.29/0.332	31.02/0.332	1792	768
45980.03/7.063	215.89/0.148	116.77/0.376	69.55/0.392	40.48/0.394	2048	768

CPU Naive	CPU Mem	GPU Naive	GPU Coalesced	GPU Shared	Resolution	Entities
985.54/0.145	4.28/0.010	2.80/0.073	1.53/0.044	0.90/0.034	256	1024
3951.82/0.649	16.98/0.026	10.19/0.083	5.92/0.048	3.42/0.049	512	1024
8803.87/0.238	38.73/0.074	22.37/0.120	13.15/0.070	7.62/0.096	768	1024
14470.23/0.377	71.19/0.214	39.55/0.128	23.31/0.237	13.50/0.157	1024	1024
24216.70/0.313	111.59/0.334	61.21/0.102	36.19/0.016	21.06/0.200	1280	1024
33666.01/0.311	160.85/0.288	87.81/0.116	52.14/0.272	30.35/0.287	1536	1024
46010.38/1.427	220.28/0.201	119.46/0.576	71.16/0.395	41.27/0.319	1792	1024
60099.51/1.231	287.49/0.339	155.56/0.603	92.70/0.405	53.90/0.373	2048	1024
1153.09/0.014	5.35/0.016	3.53/0.066	1.89/0.011	1.13/0.102	256	1280
4707.38/0.353	21.22/0.026	13.14/0.085	7.41/0.107	4.31/0.144	512	1280
10557.87/0.241	48.51/0.114	28.90/0.109	16.43/0.069	9.49/0.073	768	1280
19317.81/0.293	89.11/0.165	51.08/0.094	29.07/0.220	16.85/0.160	1024	1280
29234.08/2.506	138.92/0.328	79.18/0.058	45.19/0.012	26.34/0.259	1280	1280
41573.67/0.344	200.82/0.283	113.89/0.471	65.26/0.338	37.97/0.304	1536	1280
60285.46/0.767	275.09/0.599	155.18/1.128	89.02/0.450	51.56/0.427	1792	1280
74511.59/3.030	360.35/0.317	202.07/1.186	115.96/0.400	67.32/0.385	2048	1280
1417.25/0.109	6.41/0.014	4.21/0.092	2.33/0.113	1.33/0.034	256	1536
5881.39/0.159	25.50/0.041	15.86/0.107	8.95/0.050	5.15/0.049	512	1536
12436.71/0.384	58.20/0.131	34.72/0.222	19.76/0.148	11.43/0.171	768	1536
22341.21/1.398	106.54/0.119	61.14/0.091	34.84/0.299	20.19/0.160	1024	1536
35287.31/1.613	167.08/0.362	94.89/0.072	54.26/0.165	31.51/0.204	1280	1536
51403.44/1.931	240.41/0.515	136.57/0.575	78.52/0.508	45.43/0.253	1536	1536
69054.75/1.842	330.70/0.222	186.59/0.928	106.64/0.482	62.16/0.393	1792	1536
88502.90/1.147	431.67/0.328	242.68/0.986	138.97/0.400	80.74/0.357	2048	1536
1667.54/0.108	7.47/0.017	4.83/0.071	2.63/0.011	1.53/0.034	256	1792
6733.32/0.310	29.75/0.050	18.20/0.123	10.37/0.171	5.93/0.047	512	1792
14144.51/1.024	67.91/0.132	40.23/0.150	22.97/0.132	13.27/0.144	768	1792
25873.22/1.540	124.35/0.174	71.24/0.096	40.62/0.201	23.56/0.186	1024	1792
40915.35/0.859	194.43/0.447	110.76/0.098	63.22/0.056	36.76/0.235	1280	1792
61968.01/0.368	281.07/0.620	159.08/0.128	91.17/0.289	52.95/0.303	1536	1792
80674.55/2.007	384.49/0.777	219.10/1.440	124.26/0.369	72.54/0.465	1792	1792
100599.60/2.937	503.47/0.490	283.60/2.211	161.96/0.367	94.25/0.446	2048	1792
1862.86/0.107	8.54/0.015	5.53/0.084	3.03/0.013	1.76/0.035	256	2048
7161.24/0.878	34.00/0.049	20.97/0.111	11.91/0.050	6.85/0.051	512	2048
16812.59/1.858	77.47/0.136	46.45/0.111	26.53/0.130	15.32/0.144	768	2048
28763.84/1.887	142.27/0.160	81.30/0.375	46.47/0.505	26.89/0.248	1024	2048
46346.86/1.760	222.37/0.560	126.93/0.574	72.49/0.309	42.11/0.339	1280	2048
67751.06/13.159	321.57/1.132	183.57/1.411	104.51/0.515	60.72/0.296	1536	2048
101639.18/14.916	440.03/0.822	250.61/2.409	141.98/0.383	82.98/0.325	1792	2048
135648.60/15.517	577.26/2.228	324.68/1.700	185.18/0.418	106.80/0.556	2048	2048

Table A.3: Results for entities with radius = 4.

CPU Naive	CPU Mem	GPU Naive	GPU Coalesced	GPU Shared	Resolution	Entities
61.61/0.018	0.10/0.004	0.58/0.010	0.44/0.006	0.30/0.006	256	64
229.70/0.340	0.32/0.024	1.17/0.002	0.55/0.014	0.34/0.005	512	64
525.06/0.401	0.64/0.012	2.28/0.332	1.01/0.132	0.65/0.013	768	64
975.69/0.162	1.11/0.041	3.54/0.157	1.73/0.164	1.10/0.016	1024	64
1550.34/0.274	1.79/0.010	5.30/0.216	2.67/0.207	1.67/0.109	1280	64
2076.52/0.432	2.49/0.013	7.16/0.187	3.81/0.258	2.38/0.026	1536	64
2928.00/0.274	3.54/0.021	9.26/0.115	5.15/0.323	3.20/0.032	1792	64
3785.54/0.520	5.01/0.038	11.80/0.189	6.70/0.417	4.15/0.040	2048	64
122.32/0.157	0.20/0.042	0.85/0.012	0.40/0.015	0.27/0.012	256	128
469.67/0.221	0.62/0.019	2.60/0.134	1.24/0.134	0.78/0.013	512	128
1110.61/0.471	1.27/0.050	5.33/0.250	2.66/0.162	1.63/0.017	768	128
1842.69/1.243	2.26/0.025	8.65/0.339	4.37/0.267	2.64/0.021	1024	128
2911.97/0.307	3.45/0.012	11.08/0.693	6.12/0.293	3.68/0.024	1280	128
4319.53/0.287	5.00/0.011	13.93/0.858	7.92/0.594	4.71/0.043	1536	128
5883.79/0.467	7.03/0.018	17.85/0.368	10.26/0.330	6.16/0.032	1792	128
7499.33/0.448	9.88/0.036	22.18/0.825	12.84/0.674	7.68/0.056	2048	128
229.01/0.050	0.39/0.038	0.98/0.069	0.48/0.010	0.31/0.107	256	256
941.27/0.085	1.20/0.033	3.02/0.106	1.63/0.107	0.98/0.107	512	256
2202.06/0.126	2.53/0.050	6.29/0.251	3.48/0.178	2.07/0.140	768	256
3694.64/0.104	4.41/0.035	10.44/0.143	5.91/0.193	3.50/0.181	1024	256
6090.33/0.109	6.88/0.044	15.98/0.162	9.15/0.203	5.41/0.208	1280	256
8168.99/0.432	9.98/0.072	22.58/0.081	13.14/0.239	7.76/0.260	1536	256
11967.81/4.814	13.98/0.175	30.52/0.349	17.85/0.311	10.50/0.247	1792	256
14279.07/5.888	18.28/0.032	39.50/0.147	23.29/0.378	13.71/0.369	2048	256
486.65/0.080	0.58/0.050	1.56/0.072	0.79/0.032	0.48/0.032	256	512
1884.61/0.129	2.22/0.010	5.31/0.110	2.98/0.048	1.74/0.047	512	512
4094.29/3.939	4.83/0.015	11.53/0.201	6.60/0.077	3.85/0.068	768	512
7719.74/5.727	8.61/0.099	20.06/0.087	11.64/0.105	6.80/0.102	1024	512
11411.26/0.956	13.53/0.084	31.07/0.169	18.17/0.196	10.61/0.202	1280	512
16095.05/1.436	19.73/0.097	44.31/0.100	26.11/0.190	15.22/0.189	1536	512
23577.02/1.754	27.81/0.066	60.07/0.353	35.55/0.309	20.72/0.317	1792	512
29193.40/0.242	36.76/0.088	78.05/0.139	46.39/0.366	27.06/0.369	2048	512
703.47/0.079	0.87/0.058	2.18/0.069	1.16/0.034	0.69/0.033	256	768
2906.35/0.157	3.31/0.012	7.76/0.110	4.44/0.050	2.58/0.046	512	768
6661.20/0.228	7.21/0.025	16.97/0.202	9.86/0.072	5.73/0.068	768	768
11273.49/1.327	12.90/0.099	29.74/0.084	17.42/0.113	10.14/0.102	1024	768
17219.72/0.239	20.26/0.030	46.24/0.134	27.20/0.198	15.84/0.199	1280	768
26448.32/0.120	29.69/0.146	66.04/0.137	39.13/0.252	22.78/0.247	1536	768
33255.50/0.947	41.82/0.118	89.61/0.344	53.23/0.301	30.99/0.305	1792	768
46293.21/6.908	55.22/0.081	116.65/0.054	69.52/0.382	40.42/0.377	2048	768

CPU Naive	CPU Mem	GPU Naive	GPU Coalesced	GPU Shared	Resolution	Entities
968.33/0.122	1.16/0.093	2.79/0.033	1.54/0.034	0.90/0.033	256	1024
3736.77/0.782	4.42/0.013	10.16/0.114	5.93/0.056	3.42/0.048	512	1024
8204.87/0.284	9.64/0.021	22.43/0.202	13.17/0.131	7.63/0.131	768	1024
15246.40/0.377	17.21/0.015	39.37/0.100	23.18/0.089	13.50/0.158	1024	1024
23150.57/0.376	27.10/0.148	61.32/0.154	36.23/0.205	21.07/0.215	1280	1024
33210.73/0.349	39.51/0.109	87.76/0.082	52.16/0.257	30.29/0.246	1536	1024
46534.59/1.431	55.80/0.046	119.28/0.473	71.01/0.317	41.20/0.322	1792	1024
57364.81/1.237	73.40/0.111	155.25/0.058	92.51/0.012	53.67/0.011	2048	1024
1163.76/0.017	1.45/0.087	3.57/0.182	1.89/0.081	1.10/0.067	256	1280
4837.75/0.380	5.53/0.052	13.13/0.121	7.38/0.049	4.25/0.048	512	1280
10968.10/0.194	12.06/0.058	28.92/0.170	16.39/0.083	9.51/0.136	768	1280
19735.37/0.312	21.85/0.964	50.89/0.083	28.96/0.014	16.84/0.159	1024	1280
28220.46/2.458	33.84/0.129	79.25/0.144	45.26/0.196	26.29/0.192	1280	1280
41801.96/0.360	49.42/0.168	113.64/0.166	65.15/0.260	37.81/0.244	1536	1280
55415.96/0.879	69.80/0.147	154.76/0.838	88.65/0.188	51.46/0.306	1792	1280
76233.73/3.298	92.00/0.096	201.18/0.222	115.62/0.007	67.02/0.007	2048	1280
1428.17/0.129	1.74/0.096	4.17/0.058	2.26/0.004	1.31/0.095	256	1536
5736.67/0.158	6.60/0.030	15.65/0.112	8.83/0.059	5.09/0.051	512	1536
13212.48/0.480	14.43/0.055	34.58/0.171	19.71/0.131	11.39/0.134	768	1536
23434.73/1.219	25.78/0.024	60.89/0.092	34.75/0.079	20.18/0.158	1024	1536
34221.90/1.625	40.60/0.151	94.98/0.115	54.20/0.041	31.50/0.208	1280	1536
51352.42/1.819	59.22/0.070	136.64/0.750	78.32/0.416	45.39/0.249	1536	1536
68297.47/2.306	83.67/0.180	186.36/1.419	106.51/0.357	61.87/0.425	1792	1536
90034.48/1.157	110.26/0.207	242.94/0.930	138.73/0.013	80.59/0.127	2048	1536
1628.94/0.102	2.03/0.013	4.91/0.174	2.66/0.078	1.54/0.062	256	1792
6342.23/0.249	7.73/0.028	18.20/0.043	10.29/0.002	5.93/0.048	512	1792
15239.15/0.815	16.91/0.053	40.28/0.173	22.97/0.131	13.26/0.126	768	1792
25812.23/1.293	30.10/0.029	71.00/0.074	40.53/0.013	23.50/0.106	1024	1792
41581.11/1.107	47.32/0.052	110.77/0.116	63.29/0.199	36.71/0.199	1280	1792
57629.16/0.381	69.16/0.078	160.30/1.354	91.38/0.612	52.94/0.333	1536	1792
79357.90/1.991	97.83/0.153	219.40/1.640	124.17/0.346	72.58/0.354	1792	1792
109135.67/3.135	128.65/0.145	283.80/1.795	161.83/0.010	94.11/0.499	2048	1792
1891.69/0.134	2.32/0.098	5.56/0.223	3.02/0.034	1.76/0.037	256	2048
7566.14/1.055	8.85/0.071	20.72/0.050	11.78/0.050	6.77/0.050	512	2048
17221.18/1.823	19.23/0.065	45.98/0.212	26.24/0.131	15.14/0.131	768	2048
29527.40/1.653	34.49/0.026	81.19/0.119	46.34/0.103	26.84/0.104	1024	2048
44846.67/1.835	54.15/0.173	126.46/0.109	72.33/0.191	41.93/0.204	1280	2048
70018.11/12.268	79.18/0.220	183.81/1.740	104.50/0.524	60.79/0.483	1536	2048
100843.57/15.748	111.56/0.203	250.53/2.618	142.41/0.453	83.46/0.487	1792	2048
131486.43/13.785	147.29/0.139	325.06/2.732	185.08/0.397	106.82/0.309	2048	2048