A Qualitative Comparison Study Between Common GPGPU Frameworks.

Adam Söderström

Department of Computer and Information Science
Linköping University

This dissertation is submitted for the degree of
M. Sc. in Media Technology and Engineering

June 2018

Acknowledgements

I would like to acknowledge MindRoad AB and Åsa Detterfelt for making this work possible.
I would also like to thank Ingemar Ragnemalm and August Ernstsson at Linköping University.

Abstract

The development of graphic processing units have during the last decade improved significantly in performance while at the same time becoming cheaper. This has developed a new type of usage of the device where the massive parallelism available in modern GPU's are used for more general purpose computing, also known as GPGPU. Frameworks have been developed just for this purpose and some of the most popular are CUDA, OpenCL and DirectX Compute Shaders, also known as DirectCompute. The choice of what framework to use may depend on factors such as features, portability and framework complexity. This paper aims to evaluate these concepts, while also comparing the speedup of a parallel implementation of the N-Body problem with Barnes-hut optimization, compared to a sequential implementation.

Table of contents

List of figures **xi**

List of tables **xiii**

Nomenclature **xv**

1 Introduction **1**

1.1 Motivation 1

1.2 Aim 3

1.3 Research questions 3

1.4 Delimitations 4

1.5 Related work 4

1.5.1 Framework comparison 4

1.5.2 N-Body with Barnes-Hut 5

2 Theory **9**

2.1 Background 9

2.1.1 GPGPU History 10

2.2 GPU Architecture 11

2.3 Frameworks 13

2.3.1 CUDA 13

2.3.2 OpenCL 15

2.3.3 DirectCompute 16
2.4 Algorithm evaluation 17
2.4.1 Parallel QuickSort 17
2.4.2 Distance transform 18
2.4.3 N-Body 19
2.4.4 Choice of algorithm 20
2.5 N-Body 20
2.5.1 Base problem 20
2.5.2 Barnes-Hut algorithm 23

3 Method 25

3.1 Implementation 25
3.2 Visualization 25
3.3 Sequential 27
3.3.1 All-pairs implementation 27
3.3.2 Octree construction 28
3.3.3 Barnes-Hut force calculation 32
3.4 CUDA 33
3.5 OpenCL 37
3.6 DirectCompute 38

4 Results 43

4.1 Performance 43
4.2 Features 51
4.3 Portability 52
4.4 Complexity 55

5 Discussion 57

5.1 Performance 57

5.2 Features 59
5.3 Portability 60
5.4 Complexity 61
5.5 Encountered problems 62

6 Conclusion 65

6.1 Future work 66

References 69

Appendix A CUDA Vector Addition 73

Appendix B OpenCL Vector Addition 75

Appendix C DirectCompute Vector Addition 79

C.1 Vector Addition Compute Shader (.hlsl) 79
C.2 Vector Addition main (.cpp) 79

List of figures

1.1 Performance comparison of GPU’s over time. [14] 3

1.2 N-Body performance increase as N grows. [33] 6

1.3 Runtime per simulated step in M. Burtscher et. al. CUDA implementation of the N-Body problem. [18] 7

2.1 Development of CPU’s. [31] 10

2.2 Theoretical giga-floating point operations per second (GFLOPS) performance of CPUs versus GPUs. [36] 12

2.3 Area use of CPU (a) and GPU (b) 13

2.4 Popularity (based on Google Trends) over time of CUDA, OpenCL, Direct-Compute. The numbers represent search interest relative to the highest point on the chart for the given region and time. 13

2.5 Hierarchical CUDA model for grids, blocks and threads. 15

2.6 Vector art encoded in a 64x64 texture using (a) simple bilinear filtering (b) alpha testing and (c) Green’s distance field technique 19

2.7 Barnes-Hut recursive subdivision, visualized in 2D for simplicity 24

3.1 Instanced quad rendering, $3.5*10^6 instances$ 27

3.2 Particle system subdivided into an octree 30

3.3 Order of the flattened tree when a breadth-first flattening algorithm is applied to a binary tree 34

4.1 Total execution time 45

4.2 Execution time of the GPGPU step 46
4.3 CUDA execution time 47
4.4 OpenCL execution time 48
4.5 DirectCompute execution time 48
4.6 Sequential execution time 49
4.7 Execution time, CUDA Struct vs Class object buffers 50
4.8 Execution time, CUDA recursive force calculation vs iterative force calculation 50
4.9 Desktop Windows Version Market Share Worldwide 53
4.10 Operating system market share (Q1 2018) [8] 54
4.11 AMD and Nvidia market share (Q4 2017) [6] 55

List of tables

2.1 CUDA memory types. 14

2.2 CUDA vs OpenCL terms 15

3.1 HLSL index variables. 40

3.2 HLSL buffer types. 41

3.3 HLSL register types. 41

4.1 System specifications on which the tests were performed. 44

4.2 Framework features 4.1. 52

4.3 Metrics for the vector addition examples listed in Appendix A, B and C . . 56

Nomenclature

Acronyms / Abbreviations

API	Application Programming Interface
COM	Center of Mass
CPU	Central Processing Unit
CS	Compute Shader
D3D	Direct3D
DT	Distance Transform
FFT	Fast Fourier Transform
FIFO	First in, first out
FMM	Fast Multipole Method
GFLOPS	Giga Floating Point Operations Per Second (double precision)
GPGPU	General-purpose Computing on Graphics Processing Units
GPU	Graphics Processing Units
HLSL	High-Level Shading Language
ILP	Instruction Level Parallelism
ODE	Ordinary Differential Equations
OpenCL	Open Computing Language
PDE	Partial Differential Equations

SIMD	Single Instruction, Multiple Data
SM	Streaming Multiprocessor
SRV	Shader Resource View
UAV	Unordered Access View
VBO	Vertex Buffer Object
VLIW	Very Large Instruction Word
VRAM	Video Random Access Memory

Chapter 1

Introduction

This report is the summary of a M. Sc. in Media Technology and Engineering master thesis performed at Linköping University at the company MindRoad AB. The paper describes an qualitative comparative study between some of the general-purpose computing on graphics processing units (GPGPU) frameworks that are in use today. The comparison will focus on the differences in GPGPU frameworks in terms of performance, framework features, portability, how easy it is to develop applications in the various frameworks as well as code complexity and understandability of the application developed, with less focus on the performance. The GPGPU frameworks that is evaluated and compared in this study are CUDA [1], Open Computing Language (OpenCL) [7] and DirectCompute [9].

1.1 Motivation

During the last decades, the performance of central processing units (CPU) have kept a steady linear inclination. CPU manufacturers have been able to put more and more components such as micro-transistors on a single chip which is the reason why the development of more and more powerful CPU's have been developed. In a paper from 1965, Gordon Moore made the observation that the number of transistors in a integrated circuit doubles approximately every two years [35]. This observation has gotten the name *Moore's Law* and today almost 50 years later, Moore's observation is still valid and applicable. The number of transistors are still increasing, but the performance of single-core CPU's have started to decline. The development has ran into three walls:

- Instruction Level parallelism (ILP) wall — There is not enough instruction level parallelism to keep the CPU busy
- Memory wall — A growing gap between the CPU speed and off-chip memory access
- Power wall — Increased clock rate needs more power which in turns leads to heat problems

These problems have started a trend among CPU manufacturers to create CPU's that have more than a single core on the chip, and the production of single-core CPU's have drastically decreased. Today all major chip manufacturers produce multicore CPU's and most devices use a multicore chip, furthermore the number of cores available on chips seems to be increasing. This multicore concept is not a new technology, graphics processing unit's (GPU) have been using this technology for a long time, and modern GPU's may contain hundreds of cores. This has started a trend to not just use the computing power within a GPU to render graphics to the screen, but to use this massive amount of parallelism for more general computing. This has led to the development of frameworks specifically intended for GPGPU purposes, and some of the most popular frameworks that are used today are CUDA developed by Nvidia, OpenCL maintained by the Khronos group and backed by a huge variety of companies, as well as DirectCompute developed by Microsoft as a part of DirectX.

While the performance inclination of CPU's have started to decline, the opposite can be said for GPU's. As can be seen in figure 1.1, the performance of GPU's have drastically increased during the last decade. One of the main reasons for this inclination is the gaming industry where the demand for more realistic and immersive graphics are pursued, which has led to GPU's becoming cheaper and at the same time more powerful.

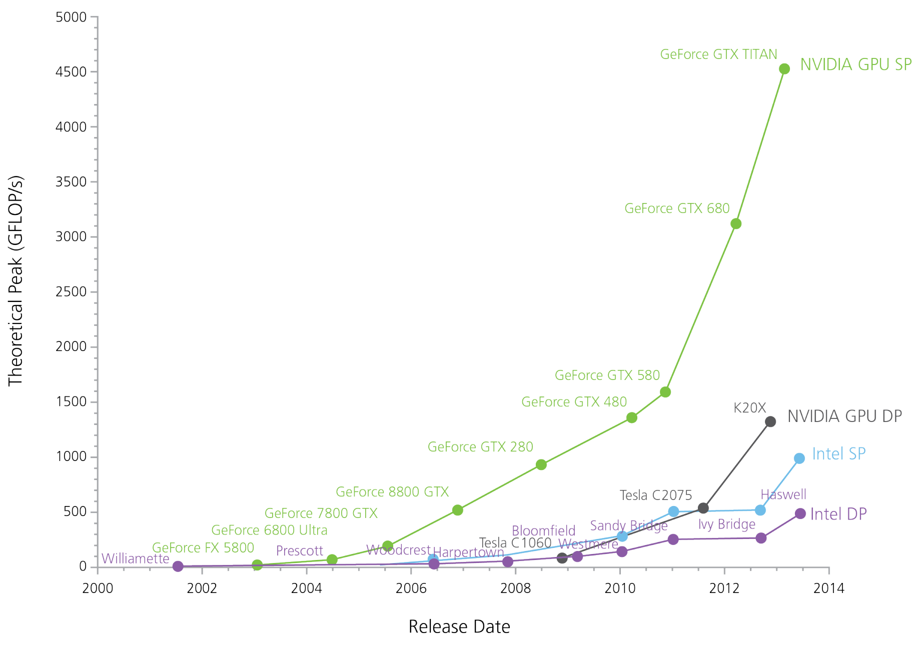


Fig. 1.1 Performance comparison of GPU's over time. [14]

1.2 Aim

This paper aims to evaluate different GPGPU frameworks in terms of performance, portability, code complexity and features with less focus on the performance evaluation. A suitable benchmarking algorithm will be implemented in the GPGPU frameworks CUDA, OpenCL and DirectX DirectCompute. The parallelized implementations will be evaluated against a sequential implementation of the same problem.

1.3 Research questions

- What is a suitable benchmarking algorithm?
- How does a parallel implementation of the algorithm compare to a sequential implementation?
- What framework-specific optimization's can be done for the implementation?

- How does the frameworks compare in terms of portability, code complexity and performance and features?

1.4 Delimitations

The selected algorithm will only be implemented in the discussed frameworks:

- CUDA
- OpenCL
- DirectCompute

Even though other optimization algorithms exists for the N-Body problem, only the Barnes-Hut algorithm will be implemented in this work, see section 2.5. The reason for this is because the thesis will not focus on evaluating the performance of the frameworks when running the selected algorithm, but on the comparison between frameworks in terms of portability, code complexity and features, which multiple optimization techniques implementations won't contribute to. Furthermore to give the implementation a fair comparison, the tree-structure used in Barnes-Hut will be constructed on the host.

1.5 Related work

This section describes previous research on the subject, both in terms of what previous work and research has been done on the N-Body problem and Barnes-Hut, as well as comparisons between the discussed frameworks.

1.5.1 Framework comparison

In 2016 a similar master thesis was performed by T. Sörman where the frameworks CUDA, OpenCL, DirectCompute and OpenGL Compute Shaders was compared as well as how the GPGPU implementations performed compared to an multithreaded implementation in OpenMP [39]. The thesis compares the different frameworks primarily in terms of performance, unlike this thesis which puts more focus on the portability, code complexity and features of the different frameworks.

The algorithm Sörman evaluated is a parallel implementation of the Fast Fourier Transform (FFT), and the result showed that the fastest framework, CUDA, was twice as fast as the slowest, OpenCL, and that the compute shader based frameworks OpenGL and DirectX, are competitive with both CUDA and OpernCL in terms of performance.

A lot of previous research has been abducted on the subject of comparing the performance between CUDA and OpenCL. K. Karimi et. al. performed an performance comparison between the two frameworks which showed similar results to Sörmans comparison [30]. Although the performance gap was more subtle in Karimi. et. al. work, the result still showed that CUDA was faster than OpenCL.

Another comprehensive study between the two frameworks was abducted by J. Fang et. al. [23]. The conducted study compares OpenCL and CUDA in terms of performance when a wide variety of benchmarking algorithms, such as graph traversal, reduction as well the N-Body problem and more, are executed in the two frameworks. Unlike the previously discussed comparisons, the result showed that under a fair comparison the performance difference between the two frameworks was very subtle although OpenCL beeing the fastest.

Another comparison study between CUDA, OpenCL and OpenGL Compute Shaders, as well as a CPU multicore implementation using OpenMP has been made by R. S. Oliveira et. al. [34]. The comparison was based of implementations of the Cardiac Monodomain Equations, and the results showed that the OpenGL approach was the fastest with a speedup of 446 compared to the parallel CPU implementation for a solution of a non-linear system of ordinary differential equations (ODEs). CUDA was the fastest for a numerical solution of parabolic partial diffential equations (PDEs) with a speedup of 8. OpenCL was the slowest for solving the PDEs and as fast as CUDA for solving ODEs.

1.5.2 N-Body with Barnes-Hut

The N-Body problem is a common problem, and a lot of previous work has been done regarding the algorithm. In the book GPU Gems 3, L. Nyland et. al. at NVIDIA corporation describes a parallel CUDA implementation of the N-Body used to generate an astrophysical simulation [33]. The implementation is done in CUDA, and describes various methods how the CUDA implementation can be optimized with the use of e.g. shared memory, loop-unrolling and coalesced memory accesses. The result of the performance can be seen in figure 1.2. The implementation described in this article does not use the Barnes-Hut algorithm to further speed up the simulation, but instead use what Nyland et. al. call *all-pairs*, meaning that all N-bodies are compared to all other, resulting in the base problem further described in section 2.4.3.

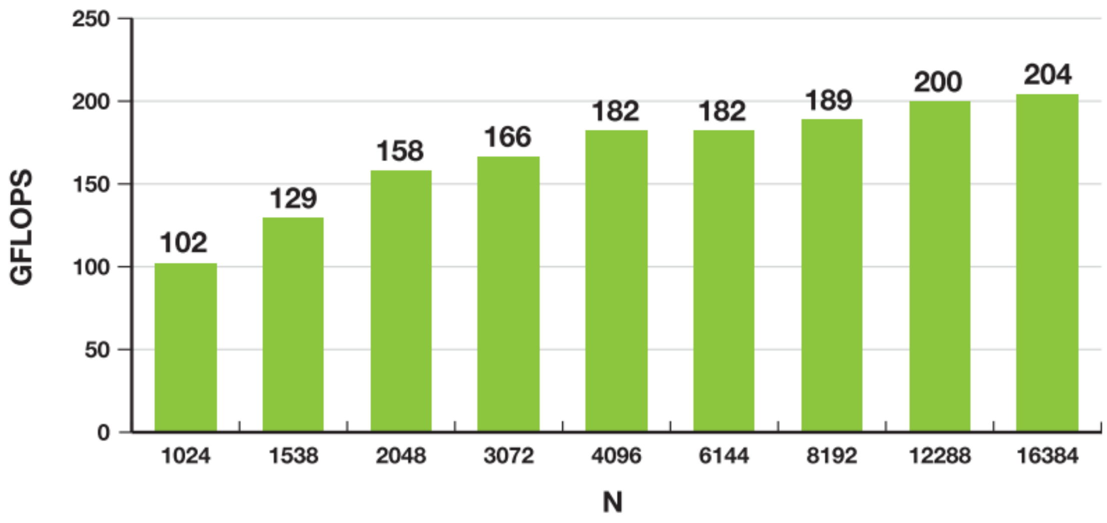


Fig. 1.2 N-Body performance increase as N grows. [33]

Another paper describing the N-body problem, and how it can be applied to astrophysical simulations is a paper by S. J. Aarseth [15]. Aarseth's work describes the historical development of the N-body problem, as well as a detailed description of the physical properties of the problem.

M. Burtscher et. al. made an implementation of the N-Body problem in CUDA with the Barnes-Hut optimization [18]. With focus on optimizing the implementation, the major part of the algorithm was performed on the GPU, resulting in minimal overhead of copying data back and forth between the host and device. M. Burtscher et. al. divided the implementation into six main steps, all performed on the GPU:

1. Compute bounding box around all bodies
2. Build hierarchical decomposition by inserting each body into octree
3. Summarize body information in each internal octree node
4. Approximately sort the bodies by spatial distance
5. Compute forces acting on each body with help of octree
6. Update body positions and velocities

The implementation was only done in CUDA and the paper focuses on how the implementation is optimized by e.g maximizing coalescing, minimize GPU/CPU data transfer, loop-unrolling etc. The resulting performance of the simulation when run in a sequential CPU implementation, all-pairs parallel implementation as well as a parallel Barnes-Hut implementation can be seen in figure 1.3.

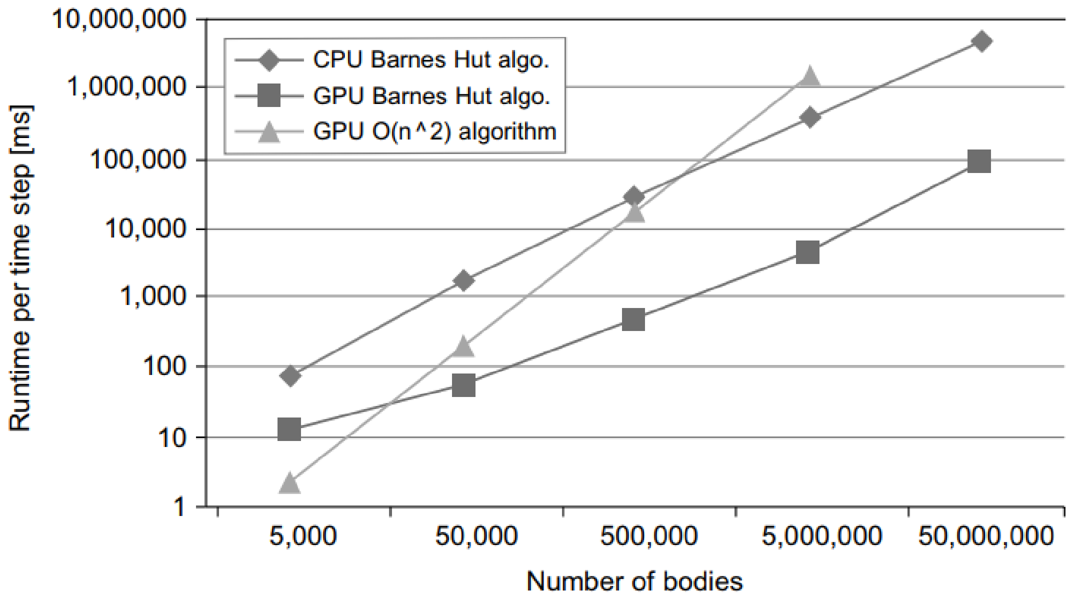


Fig. 1.3 Runtime per simulated step in M. Burtscher et. al. CUDA implementation of the N-Body problem. [18]

Chapter 2

Theory

This chapter describe background theory about parallelism, and why it has become a highly relevant topic in modern system architectures. It also describes the different frameworks and libraries evaluated in this work, as well as some typical parallelizable problems. Finally, benchmarking algorithms are presented and a motivation why each algorithm is suitable for this kind of evaluation.

2.1 Background

As mentioned in section 1.1, the performance inclination of single-cored CPU's has reached a limit. The reason for this decline is due to three walls.

The **ILP-wall** which states that there is not enough instruction level parallelism to keep the CPU busy. Some techniques do however exist such as Very Large Instruction Word (VLIW) and the Superscalar Architecture but they are limited by the hardware complexity.

The second wall, the **Memory wall** is reached because of the gap between the CPU speed and accesses to off-chip memory.

As mentioned in section 1.1 and visualized in figure 2.1, Moore's law is still valid, but the increased amount of on-chip micro transistors needs an increased amount of power, which leads to overheating problems and has been named the **Power wall**.

The solution to all of these problems are however the same. Although it is not possible to increase the single-thread performance, we can put more cores on a chip. Today, all major CPU manufacturers develop multi-core CPU's, and most devices used in our everyday life such as smartphones, laptops and desktop CPU’s have a multi-core architecture, and the number of cores on a chip seems to be increasing, see figure 2.1

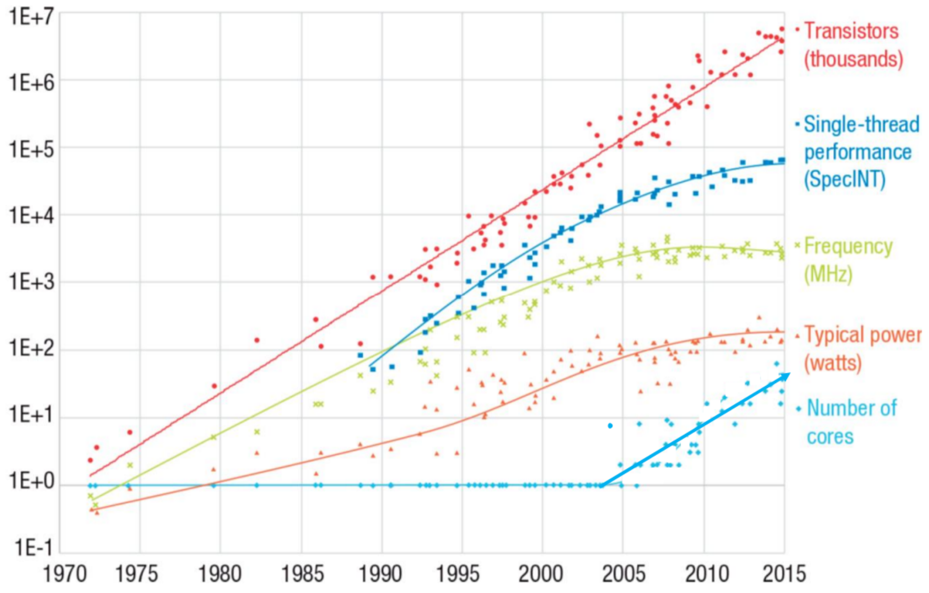


Fig. 2.1 Development of CPU’s. [31]

The idea of putting multiple cores on a single chip which may be run in parallel is not new technology, GPU’s has long been using this architecture and modern GPU chips contains hundreds of cores. This massive amount of parallelism and parallel computing power are designed to render graphics on screen and perform fast algebraic calculations commonly used in computer graphics such as matrix or vector operations, and is thus parallel in nature. But it can also be used for more general purpose computing, as quoted by Thompson et. al. "...Most of this time, however, this power is going unused because it is only being exploited by graphics applications" [40].

2.1.1 GPGPU History

With the release of programmable shaders and floating point precision GPU’s in 2001, the idea of performing general purpose computing on the GPU became popular. Early research on the subject implemented simple, well-parallelizable problems such as vector or matrix additions, and one of the first scientific GPGPU problems that outperformed the CPU was a implementation of LU factorization [22]. Another early research on the subject performed by Thompson et. al. from 2002 showed that a simple arithmetic operation applied to all elements of a vector of varying sized, outperformed the CPU once the problem size grew large enough, which is generally the case for GPGPU applications [40].

These early adaptations of GPGPU used the traditional rendering pipeline by performing computations in the fragment/pixel shaders of the application, using the two major application programming interfaces (API) OpenGL and DirectX. Although this approach adds some limitations, it is still widespread and are still in use today. Since then, both OpenGL and DirectX has released shaders specifically designed for GPGPU. These types of shaders are known as Compute Shaders (CS), and Microsoft released their CS support with the DirectCompute API, as a part of the DirectX collection of APIs.

The GPU manufacturer Nvidia realized the potential of GPGPU and developed the CUDA framework to make GPGPU programming easier by adding lots of features and simplifying the data transfer. Later on, OpenCL was released with the same purpose but with focus on the portability, with a lot of backing major companies such as Apple and IBM and it is today maintained by the Khronos group.

2.2 GPU Architecture

As previously discussed in section 1.1, whilst a CPU may have a few cores (8 cores on a modern desktop machine) that can be run in parallel, modern GPUs have hundreds of cores. Although not as powerful as a single CPU core, this extensive amount of cores allows for massively parallel computation to be made. Moreover, the GPU allows for a much higher theoretical bandwidth than a CPU as is illustrated in figure 2.2.

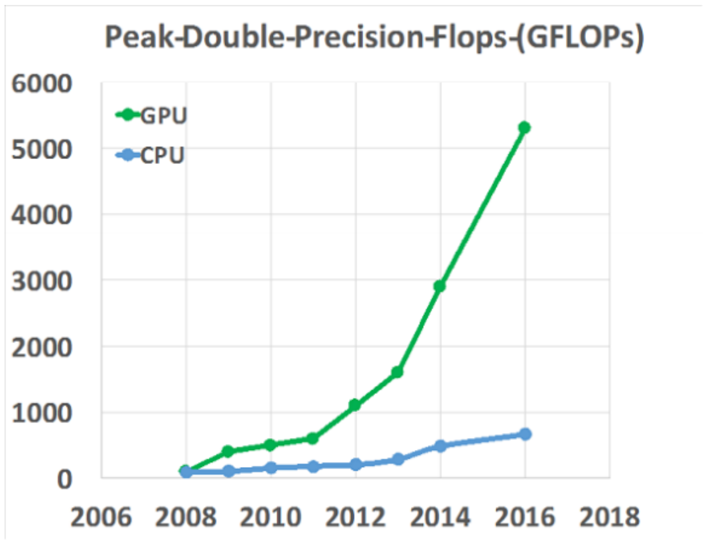


Fig. 2.2 Theoretical giga-floating point operations per second (GFLOPS) performance of CPUs versus GPUs. [36]

This is due to the efficient area use of the GPU architecture as visualized in figure 2.3, but in particular the SIMD architecture (Single Instruction, Multiple Data). SIMD simplifies the instruction handling since all cores receive the same instruction which is applied to multiple data in parallel, usually stored in list structures. The instruction that should be applied to each data-element is written as a separate piece of code, separated from the main program and run on the device (GPU, FPGA or other parallel hardware). Different frameworks use different languages in which this code is implemented, but the general GPGPU term used for this is a kernel. The most common way of executing a kernel is done in the following steps:

1. Allocate memory from the host (typically CPU) to the device (GPU or other parallel hardware).
2. Copy data from the host to the allocated memory on the device.
3. Launch the kernel, executing on the data that was just copied.
4. Copy back the result from the device to the host.

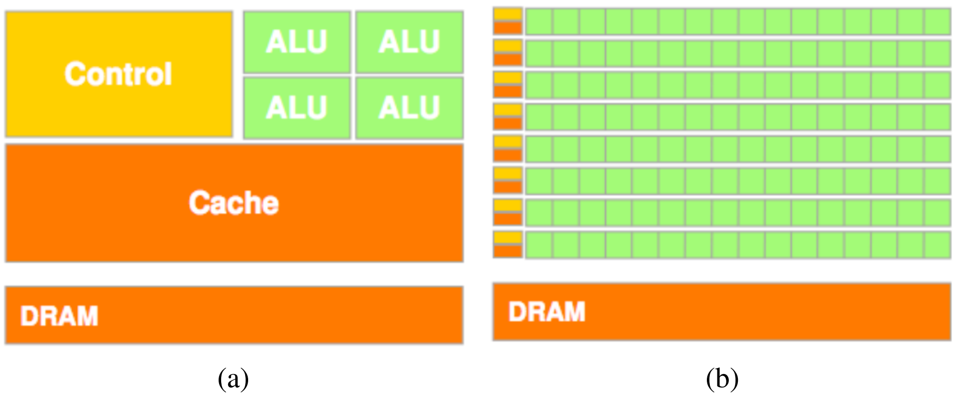


Fig. 2.3 Area use of CPU (a) and GPU (b)

2.3 Frameworks

This section describes the different frameworks that are a subject of comparison in this comparative study. Each section contains sample code of a very simple vector addition kernel for the respective frameworks. The popularity of the frameworks is based upon the chart in figure 2.4 from Google Trends.

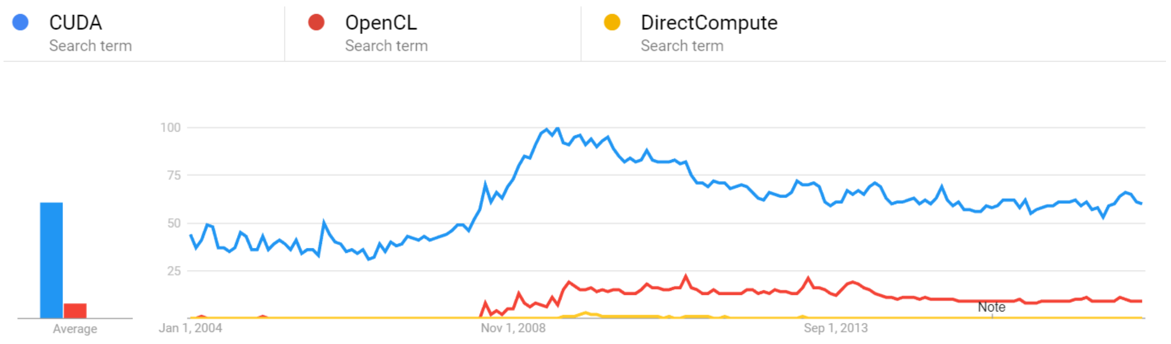


Fig. 2.4 Popularity (based on Google Trends) over time of CUDA, OpenCL, DirectCompute. The numbers represent search interest relative to the highest point on the chart for the given region and time.

2.3.1 CUDA

Released in 2007, CUDA developed by Nvidia was the first major GPGPU framework to be released. It aims to make the parallelization of a problem more manageable by providing an easy to work with API. One downside of CUDA is that is has a weak portability and can only be run on Nvidia GPU's. Despite this limitation it is a very popular framework among GPGPU developers, see figure 2.4 [1]. The installation procedure is very simple, all that is needed is a CUDA development toolkit which can be downloaded from Nvidia's webpage, and compatible hardware.

CUDA is an extension of the C/C++ language, allowing the developer to write both device and host code in a C/C++ like fashion. To run and compile CUDA, a custom compiler is used, NVCC. To define what parts of the code that should be run on the host and the device the keywords `__host__` and `__device__` is used, although the `__host__` keyword is rarely seen since it is specified per default. To specify that the next block of code is a kernel the keyword `__global__` is used. Each thread in a CUDA program is organized in a block, which in turn is organized in a grid, see figure 2.5. When launching a kernel, arguments specifying the grid and block-dimension must be supplied. There exists a few different types of memory in CUDA, these memory types are listed in table 2.1

A very simple CUDA kernel that performs a vector addition can be seen in Listing 2.1.

Memory	**Location**	**Cached**	**Access**	**Scope**
Register	On-chip	Cached	Access	Thread
Local	Off-chip	No	Read/write	Thread
Shared	On-chip	No	Read/write	Block
Global	Off-chip	N/A	Read/write	Global + host
Constant	Off-chip	Yes	Read	Global + host
Texture	Off-chip	Yes	Read	Global + host

Table 2.1 CUDA memory types.

```
__global__ void add(int *out, const int *in_a, const int *in_b)
{
        int idx = blockDim.x * blockIdx.x + threadIdx.x;
        if (idx < SIZE)
                out[idx] = in_a[idx] + in_b[idx];
}
```

Listing 2.1 CUDA vector addition kernel

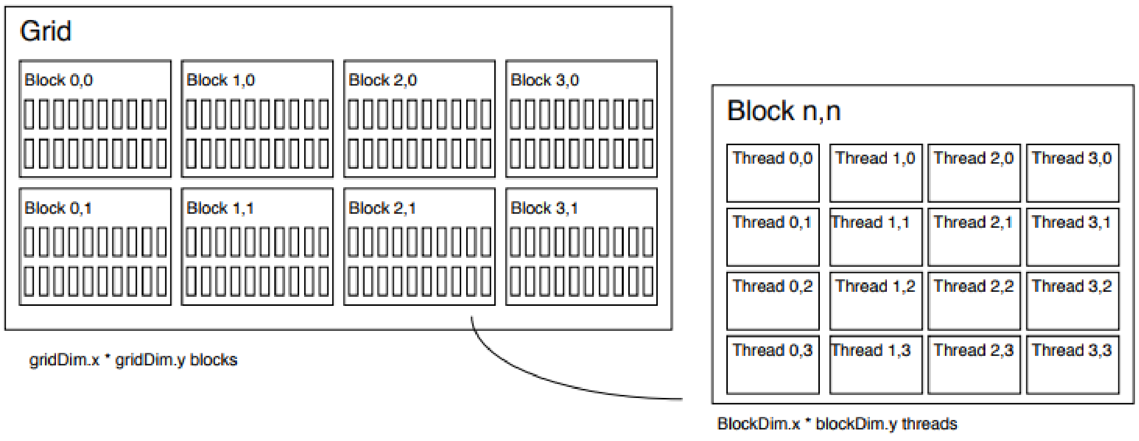


Fig. 2.5 Hierarchical CUDA model for grids, blocks and threads.

2.3.2 OpenCL

For a couple of years, CUDA was the only framework developed for the sole purpose of GPGPU and Nvidia had no competitors on the GPGPU front. That is until Apple took the initiative to develop a competitor, and backed by a lot of major companies, OpenCL was developed. OpenCL sought to offer a more portable and a wider array of supported parallel hardware, and OpenCL offers the ability to run parallel implementations on other devices than just GPU's such as FPGA's and ARM devices. OpenCL is an open standard, and implementations are available from Apple, AMD, Intel, Nvidia and more [7]. Because of this, the portability of OpenCL is good, and it can be run on most systems, provided a parallel hardware is present. Since there are multiple implementations of OpenCL, the setup procedure differs, but OpenCL is usually provided by the manufacturers drivers.

The syntax in OpenCL is quite similar to that of CUDA although some differences exist. Although the hierarchy model is very similar, OpenCL uses different terms for these, as well as for the memory types. These are listed in table 2.2.

OpenCL	**CUDA**
Compute Unit	Multiprocessor (SM)
Work item	Thread
Work group	Block
Local memory	Shared memory
Private memory	Registers

Table 2.2 CUDA vs OpenCL terms

Similar to CUDA, OpenCL uses keywords, the keyword that specifies a kernel is `__kernel`. Data that resides in the global and local memory are specified using the `__global` and `__local` keywords. Whilst CUDA automatically selects a target device of the available hardware on the system, OpenCL needs to know what parallel device to run on. Thus the setup procedure differs slightly from the procedure described in section 2.2. Before copying data and executing the kernel, a OpenCL application first have to do the following steps before doing the steps described in 2.2:

1. Get a list of platforms
2. Choose a platform
3. Get a list of devices
4. Choose a device
5. Create a context
6. Load and compile kernel code

A simple kernel that does the same thing as the CUDA kernel described in listing 2.1, that is perform a vector addition on two vectors, are given in listing 2.2.

```
__kernel void vectorAddition(__global read_only int* vector1,
                             __global read_only int* vector2,
                             __global write_only int* vector3)
{
        int indx = get_global_id(0);
        vector3[indx] = vector1[indx] + vector2[indx];
}
```

Listing 2.2 OpenCL vector addition kernel

2.3.3 DirectCompute

Initially released with the DirectX 11 API, DirectCompute is Microsoft's technology for GPGPU, and unlike CUDA or OpenCL which relies on launching kernels, DirectCompute runs a CS as a part of the graphics pipeline. Although released with the DirectX 11 API, DirectCompute runs on GPUs that use either DirectX 10 or 11 [9]. Since DirectCompute is a part of the DirectX API, no additional setup is required, but DirectCompute can only be run on Windows PCs that have a supported DirectX version.

Since DirectCompute is not a framework, but a API of the DirectX suite and uses the concept of CS to perform GPGPU calculations, it is quite different from CUDA or OpenCL. All of the computing in DirectCompute is done in a CS, which would be the equivalent to a kernel in CUDA or OpenCL. The setup process is quite similar to the one described in section 2.2, but uses the traditional graphics way of copying data between the host and the device using buffers, which are copied to the CS before the CS is run. The CS is written in the high-level shading language (HLSL) also developed by Microsoft, and a simple CS that performs a vector addition (using a structured buffer) can be seen in listing 2.3.

```
struct BufType
{
    int i;
    float f;
};

StructuredBuffer<BufType> Buffer0 : register(t0);
StructuredBuffer<BufType> Buffer1 : register(t1);
RWStructuredBuffer<BufType> BufferOut : register(u0);

[numthreads(1, 1, 1)]
void CSMain( uint3 DTid : SV_DispatchThreadID )
{
    BufferOut[DTid.x].i = Buffer0[DTid.x].i + Buffer1[DTid.x].i;
    BufferOut[DTid.x].f = Buffer0[DTid.x].f + Buffer1[DTid.x].f;
}
```

Listing 2.3 DirectCompute vector addition CS

2.4 Algorithm evaluation

In this section, algorithms that are suitable for a benchmarking application is briefly explained and discussed. The discussed algorithms are compared, and a motivation of the selected algorithm are given which is explained and discussed further in section 2.5.

2.4.1 Parallel QuickSort

As a popular sequential sorting algorithm, the QuickSort algorithm invented by C.A.R Hoare, is a recursive divide-and-conquer based sorting algorithm [27]. The algorithm has a time complexity of $O(n\ log\ n)$ in the best case, a worst case complexity of $O(n^2)$, and an average run time of $O(n\ log\ n)$. A pivot element $A[q]$ is selected from the array to be sorted. The array is then partitioned into two subarrays $A[p...q-1]$ such that all elements are less than $A[q]$, and $A[q+1...r]$ such that all elements are greater than or equal to $A[q]$. After this partitioning step, the pivot element are in its correct position. This procedure are then applied recursively to the subarrays until the entire array is sorted. Pseudocode for the algorithm can be seen in Algorithm 1. The comparison part of the algorithm is well suited for a parallel implementation, but due to the data dependency of the algorithm, parallelizing the partitioning stage is a more difficult task. Some implementations and papers describing how the algorithm can be parallelized do however exist [19][37][20].

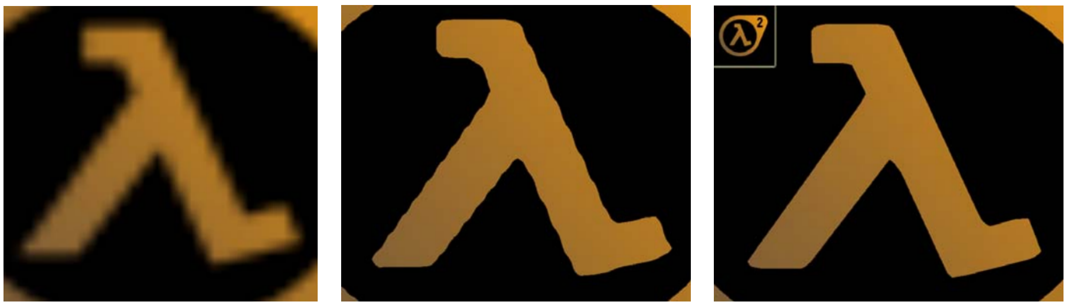
Algorithm 1 Quicksort pseudocode

```
1:  procedure QUICKSORT($A, lo, hi$)
2:      if $lo < hi$ then
3:          $p :=$ PARTITION($A, lo, hi$)
4:          $quicksort(A, lo, p-1)$
5:          $quicksort(A, p+1, hi)$
6:  procedure PARTITION($A, lo, hi$)
7:      $pivot := A[hi]$
8:      $i := lo - 1$
9:      for $j := lo$ to $hi - 1$ do
10:         if $A[j] < pivot$ then
11:             $i := i + 1$
12:             $swap(A[i], A[j])$
13:     if $A[hi] < A[i+1]$ then
14:         $swap(A[i+1], A[hi])$
```

A parallel implementation of the Quicksort algorithm would be an interesting algorithm to use for a benchmark application of this degree. The performance of an parallel implementation could be compared to the classical sequential QuickSort algorithm for varying problem sizes, although due of the triviality of the implementation this algorithm was discarded.

2.4.2 Distance transform

First presented by C.Green of Valve Softworks, a method which allows improved rendering of glyphs composed of curved and linear elements was proposed [24]. The technique works by generating a distance transform (DT) from a high-resolution texture by measuring the distance between a background texel to the closest foreground texel. The distance field is then stored into a channel of a lower-resolution texture, resulting in a texture with a arbitrary resolution, whilst occupying a small amount of video random access memory (VRAM). This low-resolution texture can then be rendered by using alpha-testing and alpha-thresholding features of modern GPU's with great results as illustrated in figure 2.6.



(a) Alpha-blended (b) Alpha tested (c) Green's technique

Fig. 2.6 Vector art encoded in a 64x64 texture using (a) simple bilinear filtering (b) alpha testing and (c) Green's distance field technique

S. Gustavson et. al. later proposed an improved version of Green's technique, using the Euclidean distance to generate a DT [26]. Whilst Green's description of the proposed algorithm was quite sparse, Gustavson et. al. technique described the general implementation more detailed. Although the general technique described by Green and Gustavson et. al. only describe the two dimensional case, the distance transform has also been extended to three dimensions [29].

One of the problems with the discussed distance transform is the ability to produce sharp features such as corners, and solutions to this problem has not been further investigated. Since the technique works on pixel/voxel level, the algorithm is well parallelizable, and an idea to further investigate this discussed problem was presented. This would however drift apart from the main idea of this research and was thus discarded.

2.4.3 N-Body

The final proposed algorithm is the N-Body problem. Although the base implementation of the algorithm is fairly trivial and embarrassingly parallel, the algorithm can be optimized by using the Barnes-Hut algorithm, reducing the time complexity from $O(n^2)$ to $O(n\, log\, n)$ [17]. The N-Body problem as well as the Barnes-Hut algorithm is further described in section 2.5.

2.4.4 Choice of algorithm

The choice of algorithm to be implemented and used to evaluate the different frameworks in this work was motivated to be complex enough so that a fair comparison between the frameworks could be made. If the algorithm was too trivial or embarrassingly parallel, the risk would be that framework specific features could be less utilized, and it would be difficult to compare the frameworks in this aspect. It would also raise the risk where the framework-part of the implementation is to small for a fair comparison to be made.

With this in mind, the algorithm had to be complex enough so that a fair comparison could be made, but not to complex so that the algorithm couldn't be implemented in the given amount of time. Another motivation of the choice of algorithm was that it would include more complex data structures, and not just lists like arrays or vectors.

The algorithm that best suits this motivation is the N-Body problem when optimized using the Barnes-Hut algorithm. Although the base case of the algorithm is embarrassingly parallel and fairly trivial, when optimized using the Barnes-Hut algorithm it gets more complex. The implementation must be able to handle the creation and traversal of trees which is not a very common implementation in GPGPU applications.

2.5 N-Body

The following section will describe the theory behind N-Body problem. A description of base problem is presented, followed by a description of the Barnes-Hut algorithm, and how it can be applied to the N-Body simulation to improve the performance.

2.5.1 Base problem

An N-body simulation is a numerical approximation of the behaviour of bodies in a system. A common implementation of the N-Body problem is a astrophysical simulation where each body represents a celestial body such as a star, galaxy or planet. Other astrophysical applications of the N-body problem can be used on a smaller scale to simulate a e.g. 3-body simulation of the earth, moon and sun. The simulation approximates how the celestial bodies behave over time when each body is affected by gravitational forces from all the others. It has also been used in physical cosmology, where N-Body simulations have been used to study the formation of e.g. galaxy filaments and galaxy halos from the influence of dark matter. [32]

The N-body problem has also been used in Plasma physics, where the bodies are ions or electrons, and in molecular dynamics where the bodies represent molecules or atoms (usually in fluid state). In fluid dynamics the vortex blob method for solving Navier-Stokes equations, and boundary value problems have been solved rapidly by using N-Body methods. [25]

Another application where the N-Body simulation are known to be used is protein folding, where N-body simulations are used to calculate electrostatic and van der Waals forces. It is also used in the computer graphics field, where it is used for turbulent fluid flow simulation and global illumination computation. [33].

The simulation made in this work is a astrophysical simulation of a cluster of stars, where each star is affected by gravitational forces from all others. As mentioned earlier this is one of the most common applications of N-Body problem and many papers and implementations regarding this kind of simulation has been made earlier [15][18][33].

General formulation

Consider n point masses m_i where $i \in [1,2,...,n]$. Each point mass has a position vector p_i in two or three dimensional space $\mathbb{R}^3$. Newton's second law states that mass times acceleration $m_i \frac{d^2 p_i}{dt^2}$ is equal to the sum of all of the forces applied on the mass. In a astrophysical simulation, the only force applied to a body is the gravitational force, and Newtons law of gravity says that the gravitational force applied to a mass m_i by a mass m_j is given by the equation

$$F_{ij} = \frac{G m_i m_j (p_j - p_i)}{||p_j - p_i||^3} \tag{2.1}$$

where G is the gravitational constant and $||p_i - p_j||$ is the magnitude of the distance between the masses. Summing over all masses, the total gravitational force F_i applied to mass m_i results in the N-body equations of motion:

$$F_i = m_i \frac{d^2 p_i}{dt^2} = \sum_{j=1, j \neq i}^{n} \frac{G m_i m_j (p_j - p_i)}{||p_j - p_i||^3} \tag{2.2}$$

Equation 2.2 has to be applied to each point mass i in each timestep of the simulation, and thus have to be compared to all other $n-1$ point masses in the system resulting in a time complexity of $O(n^2)$. Pseudo-code of this *all-pairs* n-body calculation using equation 2.2 can be seen in algorithm 2. By analyzing equation 2.2 and the pseudocode given in Algorithm 2, we can conclude that there are two parts of the algorithm that can be parallelized. Using $p = n$ processors, the outer for-loop in the main procedure can be parallelized, resulting in each particles body-body interaction is calculated by a single process. Once the particles velocities have been updated, the position updating is embarrassingly parallel using a suitable integration scheme, e.g Runge-Kutta or Euler integration.

Algorithm 2 All pars N-body pseudocode

```
procedure BodyForce($p_i, p_j$)
    $F_i := 0$
    $Gm_im_j := G * p_i.m * p_j.m$
    $dPos := p_i - p_j$
    $distance := dist(dPos)$
    $magn3 := abs(dist)^3$
    $F_i := Gm_im_j * dPos/magn3$
    return $F_i$
procedure MAIN
                                                        ▷ Update velocities
    for $i := 0$ to $n$ do
        $p_i := particles[i]$
        $F_i := 0,0,0$
        for $j := 0, j \neq i$ to $n$ do
            $p_j := particles[j]$
            $F_i := F_i +$ BodyForce($p_i, p_j$)
        od
        $p[i].v = p[i].v + dt * F_i$
    od
                                                        ▷ Update positions
    for $i := 0$ to $n$ do
        $p_i := particles[i]$
        $p_i.pos = p_i.pos + p_i.v * dt$
    od
```

Although this all-pairs implementation is straightforward and could be implemented effortlessly, with the time complexity $O(n^2)$ it is not very performance efficient and scales badly as the size of the problem grows. Various methods to improve the performance of the algorithm has been investigated by using hierarchical methods such as fast multipole method (FMM), the Barnes-Hut algorithm and a radiosity method. [38][17]

Both Barnes-Hut, further discussed in section 2.5.2, and the FMM uses a recursive decomposition of the computational domain into a tree structure. The FMM is very similar to the Barnes-Hut method. The main difference between FMM and Barnes-Hut is that while the Barnes-Hut only computes particle-particle, or particle-cell interactions, the FMM also computes interactions between internal cells directly. Due to the FMM method beeing more complex, only the Barnes-Hut algorithm is implemented in this work [38]. The radiosity implementation is mostly used in computer graphics when computing global illumination and is not further discussed here.

2.5.2 Barnes-Hut algorithm

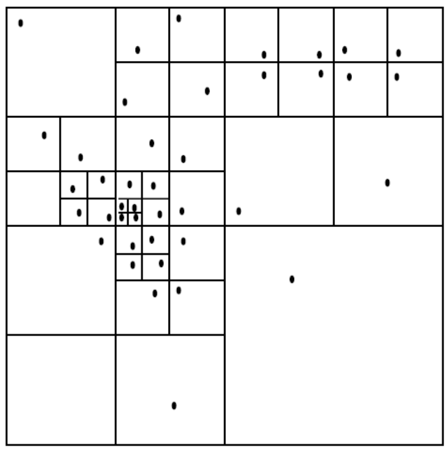
Invented by J. Barnes and P. Hut, the Barnes-Hut algorithm is a hierarchical tree based force calculation algorithm with the time complexity $O(n\ log\ n)$ [17]. The algorithm is described in three dimensional space, but is trivially adapted to two dimensional space as needed.

The algorithm uses a hierarchical tree-structured subdivision of space into cubic cells, each divided into eight subcells whenever more than one particle is found to occupy the same cell. The root of the tree does thus represent the entire spatial space the particles reside in. When calculating the force applied to a single particle, only particles that are close enough under a certain condition, will be accurately force calculated. Particles that are far away from the particle will have a small impact on the resulting force, and can thus be approximated. In Barnes-Hut this is done by calculating each cells center of mass after the tree has been constructed. The tree is then traversed for each particle, if the cell's center of mass is far enough away from the particle, the entire subtree of that cell is approximated by a single "particle" at the cell's center of mass. If however the cell's center of mass is not far enough away from the particle the cell's subtree must be traversed [38].

The tree is built by recursively adding particles into the initially empty root cell, subdividing into eight children when necessary. The resulting tree's internal nodes are space cells, and the leaves of the tree are individual particles. A two dimensional spatial representation as well as the resulting tree can be seen in figure 2.7. Each node in this tree structure has four children and is thus called an quadtree. In three dimensional space, each node will thus have eight children, and this kind of tree-structure is called a octree. The tree is then reconstructed at every timestep to avoid ambiguity and tangling. For a timestep t, the N-Body simulation procedure can be summarized in the following steps, each which can be run in parallel [18]:

1. Compute bounding box around all bodies
2. Build hierarchical decomposition by inserting each body into the octree
3. Summarize body information in each internal octree node

4. Approximately sort the bodies by spatial distance
5. Compute forces acting on each body with help of the octree
6. Update body positions and velocities


(a) Spatial domain

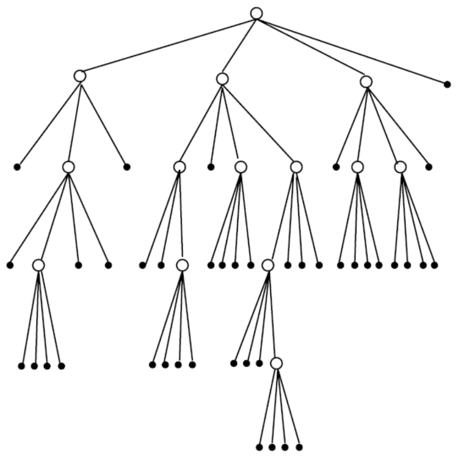

(b) Quadtree representation

Fig. 2.7 Barnes-Hut recursive subdivision, visualized in 2D for simplicity

Chapter 3

Method

3.1 Implementation

A common application where GPGPU is used is when computing calculation heavy simulations such the N-Body problem described in this thesis. Other common visualizations where GPGPU can be applied is to visualize fractals such as the Julia or Mandelbrot set, named after the french mathematician Gaston Julia and Benoit Mandelbrot. GPGPU has also been used in medicine for accelerated medical image reconstruction [16], as well as accelerating the Marching Cubes algorithm [28].

This section describes the implementation of the visualization and the parallel N-Body algorithm in all discussed frameworks, as well as how the measurements were performed and what framework specific features was used. All implementations was implemented in C/C++. The visualization was implemented in the cross-platform API OpenGL on a PC.

3.2 Visualization

Although not necessary for the evaluation, a visualization of the system was implemented. This was the first step in the implementation process, and the visualization was made using OpenGL. The purpose of the visualization is to make it easier to test and debug the application, which is very difficult without a proper way of visualizing the calculated positions. The N-Body visualization is very similar to a particle system, where each body is represented by a particle. To be able to visualize a very large amount of bodies, the visualization performance is critical, and there are a few ways of implementing a particle system in OpenGL.

The first, and perhaps the most intuitive way is to render a sphere in all positions, by calling `glDrawArrays` N times e.g. in a for-loop. This is very inefficient in two regards; a sphere requires a lot of vertices to appear smooth, and drawing a large amount of spheres requires a large amount of vertex data. The second reason this is inefficient is because all SM's (Streaming Multiprocessor) on the GPU will be dedicated to drawing a single polygon, resulting in a huge amount of performance loss. However, since the particles are so small, they don't have to be rendered with a high resolution. A commonly used trick when rendering particle systems is to represent each particle as a quad with a semi-transparent texture with a circle. Each particle will thus only consist of four vertices, which is far less than if each particle was represented by a sphere. The quad is then rotated so that the quad always faces the camera, giving the illusion that the particle is actually a sphere (or whatever shape the texture represents). This technique is known as *billboarding*.

The solution to the second problem is a bit more complex, and a few solutions exists. One way is to generate a single vertex buffer object (VBO) with all the particles in them. This is a easy and effective solution that works on all platforms.

The second way is to use geometry shaders to render a particle in each position. The downside to using geometry shader is that geometry shaders is only supported in systems with OpenGL version 3.2+, and is thus not very portable.

The third way is to use instancing, meaning that a single mesh is used, but many instances of the mesh. This solution has a nice balance between performance and availability and was thus chosen in this implementation. To achieve this, two main VBO's are used: one VBO containing the positions of the quad, i.e. four vertex coordinates, and a second VBO of size n containing the positions of the instances of the quad, where n is the number of instances. The quad is then rendered using `glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, 4, n)` [11], where the first parameter states that the object should be rendered as a triangle strip, i.e. a series of connected triangles, sharing vertices. The second parameter specifies the starting index in the enabled arrays. The third parameter specifies the number of indices to be rendered, and the fourth the number of instances. The quad positions are passed to the vertex shader as an attribute, and the shader then translates the quad into its correct position. The result of the implementation can be seen in figure 3.1, where $n = 3.5 \times 10^6$, running at a stable 60fps on a Nvidia GTX970 GPU. Each quad is rendered in a random (x, y, z) position in a given bounding box.

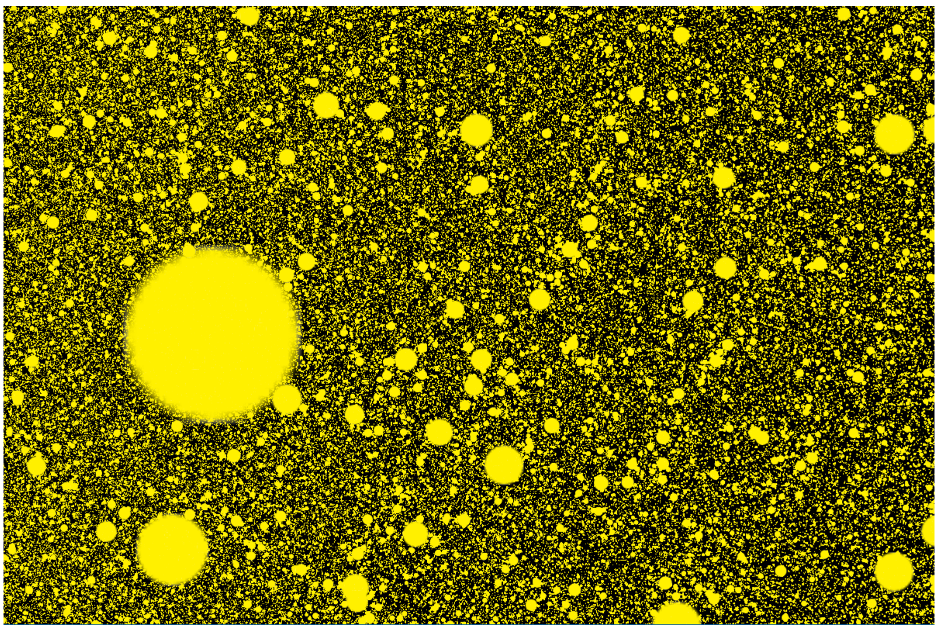


Fig. 3.1 Instanced quad rendering, $3.5*10^6 instances$

3.3 Sequential

This section will describe how a sequential implementation of the all-pairs N-body algorithm was implemented, followed by an optimized sequential implementation using the Barnes-Hut method. [17]

3.3.1 All-pairs implementation

With the particle system visualization implementation described in section 3.2 as a template, a sequential all-pairs N-Body simulation was implemented. A particle is represented by a data structure which contains the position, velocity and mass of the particle. These particles are then instantiated and given a random position. All particles are given the same amount of mass, which makes it easier to analyze the simulation. To make the visualization more visually interesting, the particles are placed in a galaxy like structure using polar coordinates using a random φ angle, and a random radius r such that $x = r\ cos\ \varphi$, $y = r\ sin\ \varphi$. If the particles index is even it is translated with a constant length in the positive x-direction, whilst if the index is odd, it is translated in the negative x-direction resulting in two separated galaxy shapes.

To further mimic the behavior of a galaxy, each particle is given a speed according to:

$$v = (c - p) \times z \tag{3.1}$$

where v is the velocity of the particle, c is the center of the galaxy, p is the position of the particle and z is the z vector $(0,0,1)$. This results in that each particle will spin around the center of the galaxy, simulating how stars in a real galaxy behaves.

Once the system has been initialized, the integration is done in two separate steps. First, the net force F applied on each particle is calculated as it is affected by gravity from all others according to equation 2.2. In a sequential all-pairs implementation, this is typically done by using a nested for-loop resulting in a time complexity of $O(n^2)$. The next step is to update the position of the particle which is done using a first order Euler integration according to:

$$p' = p + F * \Delta t \tag{3.2}$$

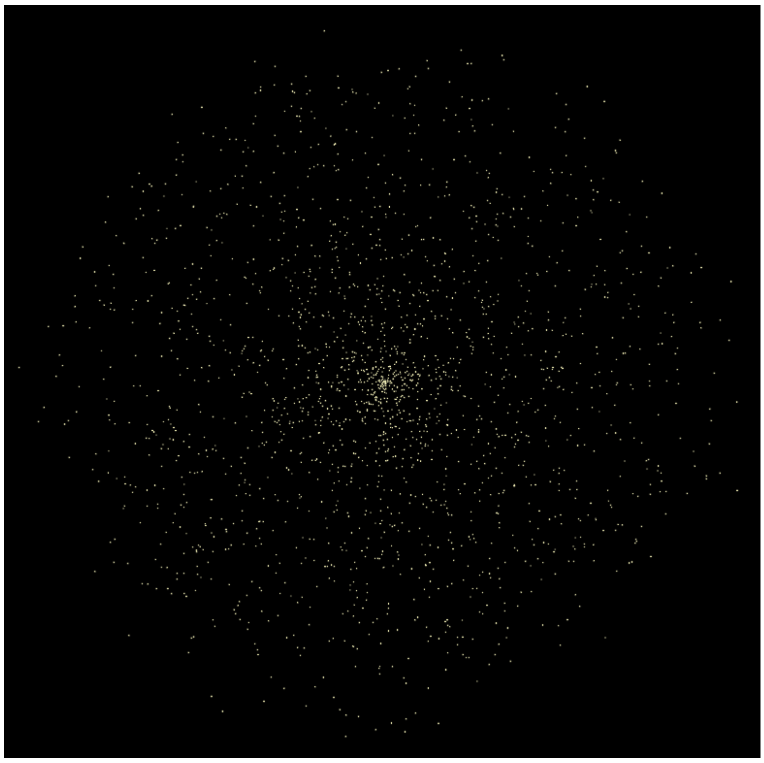
where p' is the new particle position, p is the old position, F is the net force applied to the particle and Δt is the delta time of the simulation. Since this is a first-order method, the error grows quickly with a local error proportional to the square of the step size, and a global error proportional to the step size. The step size of the Euler method used in this simulation does thus depend on the delta time which may vary on different systems. To prevent this a constant $\Delta t = c$ is used, with a small enough constant c so that the integration is accurate enough. This simulation does not aim to be a physical accurate simulation so this integration method works well in this implementation.

3.3.2 Octree construction

The next step in the implementation was to start working on the Barnes-Hut algorithm by creating an octree from the particles in the particle system. This procedure follows the theory described in section 2.5.2 and illustrated in figure 2.7. The octree is represented as a data structure containing the spatial bounds of the cell, the spatial position of the node's center of mass, as well as the total mass contained in the node. It also contains it's eight children, which are of the same kind of data structure, thus resulting in a recursive data structure. The first step to building the tree is to find the bounds of the entire particle system, which is done in a simple reduction algorithm. With this information, the root cell of the octree can then be constructed with the given bounds. The next step is to insert the positions of the particles which is done recursively. The position of the particle to be inserted is passed as an parameter, and the tree is then recursively traversed to find the respective node to insert the particle in. If this node is found to be already occupied with another particle, the node is then subdivided into eight new nodes, and the two particles are then moved into their respective new nodes.

A problem with the insertion is to know what octant the particle should be inserted to. Since there are eight octant's this can be represented as a single byte. By comparing the position to be inserted with the middle of the cells x,y,z dimensions, this problem is reduced to a single binary expression by following a set of fixed rules. Pseudocode for this procedure can be seen in algorithm 3, the procedure `insert` is called for the root of the tree once for each particle. The resulting octree applied to a system with 2048 particles is visualized in figure 3.2.

Once the tree has been constructed, the next step is to calculate each nodes center of mass (COM) and the total mass contained inside the spatial domain that the node represents. This is again done by utilizing recursion. Each nodes COM is the average of it's children's COM and its total mass is the sum of it's children's total mass. The algorithm starts by calculating the root's COM and total mass, which recursively steps through the tree, calculating the COM of each nodes as it traverses the tree. The termination condition of the recursion is when a leaf node, or a empty cell is reached.


(a) Particle system

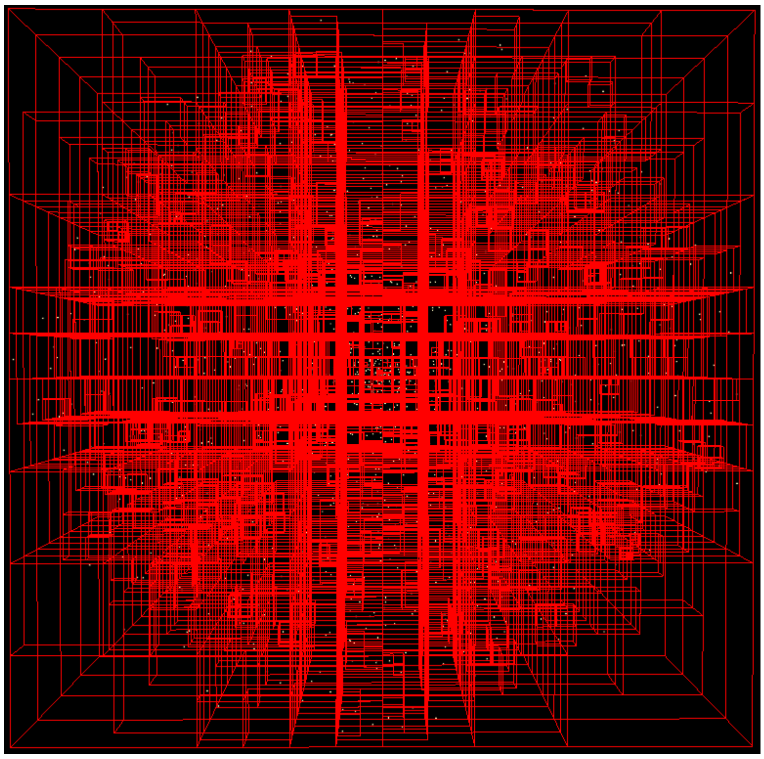

(b) (a) with rendered octree bounds

Fig. 3.2 Particle system subdivided into an octree

Algorithm 3 Building octree pseudocode

```
1: procedure INSERT($x, y, z, data$)
2:     if $node.isEmpty$ then                                          ▷ Turn into leaf
3:         $node.posx := x$
4:         $node.posy := y$
5:         $node.posz := z$
6:         $node.data := data$
7:     else                                                  ▷ Node already contains data
8:         if $node.isChild$ then                 ▷ Subdivide and move into appropriate child
9:             INSERTSUB($node.posx, node.posy, node.posz, node.data$)
10:            $node.data = NULL$
11:        INSERTSUB(x,y,z, data)
12: procedure INSERTSUB($x, y, z, data$)
13:     $sub := 0$
14:     if $x > midX$ then                    ▷ Children 0,2,4,8 have positive x-coordinates
15:         $sub+ = 1$
16:         $newMinX := midX$
17:         $newMaxX := maxX$
18:     else
19:         $newMinX = minX$
20:         $newMaxX = midX$
21:     if $y > mid_y$ then                   ▷ Children 0,1,3,4 have positive y-coordinates
22:         $sub+ = 2$
23:         $newMinY := midY$
24:         $newMaxY := maxY$
25:     else
26:         $newMinY := minY;$
27:         $newMaxY := midY;$
28:     if $z > midZ$ then                    ▷ Children 0,1,2,3 have positive z-coordinates
29:         $sub+ = 4$
30:         $newMinZ := midZ$
31:         $n_m ax_z := maxZ$
32:     else
33:         $newMinZ := minZ$
34:         $newMaxZ := midZ$
35:     if !children[sub] then          ▷ $sub$ will now contain the octant index, $sub \in 0, 8$
36:         $children[sub] := newOctreeNode($
                $newMinX, newMinY, newMinZ,$
                $newMaxX, newMaxY, newMaxZ)$
        $children[sub] - > insert(x, y, z, usr_d ata)$
```

3.3.3 Barnes-Hut force calculation

Once the tree has been built and the COM of each node has been calculated, the force calculation can begin. Besides the tree, an container containing information about all particles in the systems is used to simplify the data management. Each particle in this container is similar to that of the cells in the tree described in section 3.3.2. The particle data structure contains information about the particles mass, position and velocity. This makes the force calculation simpler since the tree can be traversed once for each particle when calculating its net-force, which is a procedure that can easily be parallelized.

The force calculation for a particle p starts by traversing the tree from the root and then recursively traverses the tree. For each node it traverses, the euclidean distance from the particles position to the COM of the current node is calculated according to:

$$d(q,p) = d(p,q) = \sqrt{(p_x - q_z)^2 + (p_y - q_y)^2 + (p_z - q_z)^2} \tag{3.3}$$

where q and p is the position of the particle and the node. If this distance is equal to zero, it means that the current node is the same node as the particle, and the traversal continues. If the COM of the node is far away enough, the entire sub-tree of the node can be approximated as a single point mass and the force can be calculated using the COM and the mass of the node. If the node is to close to the particle, the sub-tree has to be "opened" and the traversal continues through the sub-tree, recursively calculating the net-force as the tree is traversed.

One of the problems with the force calculation is what the distance that decides if a node is far away enough so that the sub-tree can be approximated should be. Barnes. et. al. does not mention a general strategy how this distance condition is specified [17] . Burtscher et. al. uses a constant cutoff distance, which specifies when a node is far away enough [18]. This implementation uses a more general approach to this problem. The average width of the nodes bounds is calculated according to equation 3.4, where the max and min variables represent the bounds of the current node. The calculated average width is then used to calculate the ratio between the width and the distance to the node. If this width-to-distance ratio is smaller than a constant c, the node is considered to be far away enough to be evaluated as a single point.

$$\begin{aligned} W_{avg} = &((max_x - min_x) + \\ &(max_y - min_y) + \\ &(max_z - min_z))/3 \end{aligned} \tag{3.4}$$

When the net-force for all particles have been calculated, the positions of the nodes is updated. It is important that this step is done after all of the forces have been calculated to get a correct simulation. When the forces are calculated, the velocity property of each particle is updated. The position update is then done by applying equation 3.2 for each particle. With the position update, the simulation step is concluded. The tree is deallocated before beeing rebuilt in the next simulation step.

3.4 CUDA

Once a sequential implementation was in place the work on porting the implementation to CUDA was started. The octree data structure was built using pointers, pointing to its child sub-trees. This creates two problems when copying the tree structure to the GPU. The first problem is that the size of the tree is unknown and changes very frequently as the tree is rebuilt every simulation step, resulting in an uncertain amount of data to be transferred to the device. Although the size of the tree can be calculated, either when the tree is beeing built, or by a simple depth- or breadth-first traversal of the tree once the tree has been constructed, this solution does however only solve the first problem.

The second problem is that pointers in the octree point to a memory location in the heap memory, which when transferred to the device must be updated to point at the correct position on the device, which would be a performance heavy task. Moreover, complicated structures involving a lot amount of pointers works poorly performance-wise in CUDA, since all the pointer chasing might drastically decrease the total memory bandwidth.

The solution to both problems is to flatten the tree into an array with a fixed size $n > N$, where N is the number of bodies. The tree will have N leaves, so the size of the array has to have a fixed size which is bigger than the amount of bodies. This was done in two steps. The first step was to do an iterative breadth-first traversal of the tree, starting from the root, and inserting the traversed nodes into the array, as well as assigning the index to each node. This results in that the array will be sorted top-down, left-right from the tree with the root at index one, visualized on a binary tree in figure 3.3. Now that the tree has been flattened into a single list, the child pointers of each node can be replaced with the indices of the child nodes which lies in the same list as the node, making the requirement of pointers in the CUDA kernel redundant and avoiding the the pointer chasing issue.

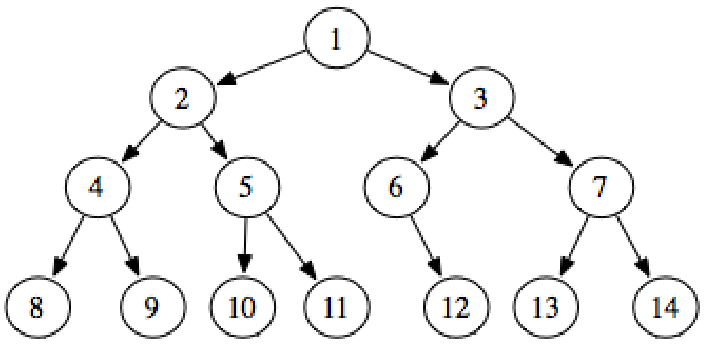


Fig. 3.3 Order of the flattened tree when a breadth-first flattening algorithm is applied to a binary tree

Now that the octree has been flattened into a tree, it can effortlessly be copied to the device since both the size of the flattened tree is known, and the pointers to each nodes children has been abstracted away and been replaced with indices in the flattened tree. The flattened octree list can simply be passed to the kernel as an argument.

The remaining part of the algorithm is now reduced to two well parallelizable problems. The first is to calculate the net force of each particle, and the second to update the positions of each particle. The sequential force calculation handles one particle at a time, and traverses the tree once for each particle. This problem is well parallelizable by letting each thread handle one particle. The position update was parallelized in a similar fashion, each thread updates the position of a particle in parallel by using the Euler integration method described in equation 3.2.

The sequential algorithm which was implemented before the CUDA implementation started utilizes recursion to traverse the tree when calculating forces. All Nvidia GPU's of compute capability 2.0 and higher support a stack and can thus utilize recursive functions in CUDA, making the tree traversal in the kernel very similar to the sequential tree traversal. However since recursive calls can't be inlined, and since the amount of overhead spawned when allocating memory for the recursive calls increases, they might significantly decrease the performance due to the spawned overhead when using recursion. Thus an iterative tree-traversal was implemented as well in a breadth-first fashion.

One way of doing an iterative breadth-first traversal is by using a queue. All children of the active node is pushed onto the queue, and in each iteration the first element of the queue is selected, iteratively pushing each nodes children onto the queue. CUDA does not support the use of the `std::queue` in kernel code, so a manual FIFO (First In, First Out) queue data-structure was implemented.

Since CUDA supports C++ in kernels the queue could be implemented as a datastructure, containing expected member functions such as `push()`, `pop()` etc. To avoid pointer chasing, the queue is implemented using an indexed array as well as integers indexing the front and the back of the queue. The code for the queue can be seen in listing 3.1.

```
struct MyQueue {
        static const int MAX_SIZE = 10000;
        int f = -1, r = -1;
        int A[MAX_SIZE];

        bool empty() {
                return (f == -1 && r == -1);
        }
        bool isFull() {
                return (r + 1) % MAX_SIZE == f ? true : false;
        }
        void push(int x) {
                if (isFull()) {
                        return;
                }
                if (empty()){
                        f = r = 0;
                }
                else {
                        r = (r + 1) % MAX_SIZE;
                }
                A[r] = x;
        }
        void pop()
        {
                if (empty()) { return; }
                else if (f == r) {
                        r = f = -1;
                }
                else {
                        f = (f + 1) % MAX_SIZE;
                }
        }
        int front(){
                if (empty()) {
                        return -1;
                }
                return A[f];
        }
};
```

Listing 3.1 Device queue used for iterative traversal of the tree

3.5 OpenCL

Once the CUDA implementation was done, the work started on the OpenCL implementation. To be able to perform a fair comparison between the selected frameworks, the same host program that was used for the CUDA implementation was used for the OpenCL implementation. This works well since the main program is implemented in a way so that the device part of the code is well abstracted away from the main application. The host is responsible for building the octree, flattening the octree into an array as well as the graphical simulation, the device part of the code is responsible for calculating forces and updating the positions of the bodies. The device part of the application is separated into its own class object, thus abstracting it away from the main application.

The OpenCL distribution used in this implementation is Nvidia's OpenCL implementation, version 1.2, which is the newest version developed by Nvidia and adds some restrictions further discussed in section 4.2. Although OpenCL are in many aspects very similar to CUDA, some major differences exist. Since OpenCL does not only target GPU devices, but a wider range of parallel hardware (see section 2.3.2), the implementation needs to specify what device that the parallel kernel should run on. This step is automated by CUDA since it selects the default GPU device available on the system. Since this implementation was performed at a system containing only one GPU, this implementation selects the first available GPU residing in the system. Moreover, unlike CUDA where the device code is processed at compile time using Nvidia's NVCC compiler, OpenCL has to compile the kernel code during runtime. Although the kernel code can be written inline as a string, the common practice is to separate the device and host code which was done in this implementation. The kernels are written in separate `.cl` files which is read by the host application, and compiled during run time.

Similar to the CUDA implementation, the parts of the N-Body simulation that has been parallelized is the force calculation as well as the position updating. The flattened tree that is to be copied to the device is a data structure containing a list of class objects, `OctreeNode`. To get this data to the device in CUDA is to just do a simple copy of the array to the device memory since CUDA kernels are based on C++ and does thus support classes. Nvidia's OpenCL version 1.2 which is used in this implementation uses the language OpenCL C for device code and does thus not support this feature. Newer versions of OpenCL (v2.0, released in 2013) does support OpenCL C++ in kernels which is based on C++11 and allows for the creation of classes, templates, operator overloading, function overloading etc [10]. OpenCL C++ does however not support certain C++ features such as exceptions, memory allocation, recursion and function pointers. Although Nvidia is a major backer to OpenCL, Nvidia's latest version of OpenCL is version 1.2, and does thus not support C++ features, so the device code has to be written in OpenCL C. This means that the process of copying the flattened tree to the device is more cumbersome than in the CUDA implementation. Although OpenCL C does not support classes, it supports data structures to be copied between the host and the device. Thus the flattened tree was restructured to contain data structure representations of the `OctreeNode` class. This step is done in the host algorithm that flattens the tree. Now that the tree has been flattened and each node is represented as a data structure instead of a class object, the flattened tree can be copied to the device.

As mentioned in section 3.4, all Nvidia GPU's of compute capability 2.0 support recursive calls in kernels, and although the GPU used for the OpenCL implementation is the same as the CUDA implementation and does thus hardware-wise support recursive calls, OpenCL does not. The approach to the first implementation is thus different to the CUDA implementation described in section 3.4. Since the recursive traversal of the tree is more intuitive and simpler, it was the first force calculation algorithm that was implemented in CUDA, but since OpenCL does not support recursion inside kernels, the iterative version was implemented straight away. To perform a breadth-first tree traversal, the most common iterative way is to used a queue, where non-visited nodes are enqueued. As mentioned in section 3.4, CUDA does not support the `std` library, which is also the case in OpenCL, and an own implementation of a queue was made. In CUDA this was done using a C++ `struct`, containing initialized member variables and member functions expected from a queue such as `push()`, `pop()`, `front()` and `empty()`. However since OpenCL C is based on C99, where data structures can't contain member functions, and structure variables can't be given a default value, the code responsible for pushing, popping etc. from the queue had to be inlined in the kernel.

3.6 DirectCompute

Compared to CUDA and OpenCL, DirectX Compute Shaders, or DirectCompute, is very different. Whilst CUDA and OpenCL are frameworks developed with the purpose of GPGPU, DirectCompute is more similar to traditional graphics programming and resembles traditional GPGPU using fragment-/pixel-shaders. DirectX developed by Microsoft, for Microsoft systems, is widely available on Microsoft systems, and DirectX 11 is included in systems running Windows 7 or newer.

Released as a part of the Direct3D 11, Microsoft's Compute Shader (CS) is an alternative method of performing general purpose computing on the GPU [5]. Similar to a CUDA or OpenCL kernel, the CS is similar to a vertex- or pixel-shader but with the purpose of doing more general computing, and is written in High Level Shading Language (HLSL), developed by Microsoft.

Compared to CUDA and OpenCL, the initial setup procedure in a Direct Compute application is more complex and a few steps has to be done before the application is ready to perform the computations.

The first step is to create a device context and to create a target device on which the computation will be performed. The device is created by calling the method `D3D11CreateDevice`, with arguments defining what type of device that should be created. Since DirectCompute is only supported on systems with Direct3D 10 or 11, this must be taken into account when creating the device. To specify this, a `D3D_FEATURE_LEVEL` variable is passed as an argument to the device creation method which specifies which feature levels the device will use. Once the method has been called, the resulting feature level of the device can be determined. This information is important because if the resulting feature level is too low, lower than 10.0, CS is not supported. This method also allows the option to select what hardware the device is referring to, in this implementation the default graphics card is selected.

The second step is to compile the actual HLSL compute shaders. The DirectX provides a method for doing this depending on the DirectX version is running on the system. One thing to keep in mind when creating the CS is what CS shader profile to use. Although there's not much feature-wise difference between CS 4.0 and 5.0, if the system is running Direct3D (D3D) feature level of 11 we generally want to use CS 5.0 as it allows for better performance on 11-class hardware [9]. The HLSL CS source file can then be compiled by using either `D3DCompileFromFile` or `D3DX11CompileFromFile`, depending on the feature level of the hardware. If the source file is successfully compiled, a CS instance object can then be created from the compiled shader code.

The process of performing the computation on the device is similar to the process in CUDA and OpenCL. First memory has to be allocated on the device. DirectCompute uses buffers to achieve this. These buffers have a wide variety of options which are set by using a `D3D11_BUFFER_DESC` data structure instance, passed as an argument when creating the buffer. This description may contain information such as if it is a constant buffer, if it is a raw or structured buffer, the size of an element in the buffer as well as the total size of the buffer. This data structure along with the data the buffer will contain is then be passed to `ID3D11Device::CreateBuffer` which creates an buffer instance object.

Before the CS can be launched, view interfaces has to be generated from the buffers which can then be passed and accessed by the CS. Two types of view interfaces was used in this implementation: `ID3D11ShaderResourceView` and `ID3D11UnorderedAccessView`, where the

Variable name	Description
SV_GroupIndex	Flattened 1D index of a thread within a thread group
SV_DispatchThreadID	Global thread index, sum of SV_GroupID * numthreads and GroupThreadID
SV_GroupThreadID	3D Indices for a thread within a group
SV_GroupID	Indices for which thread group a compute shader is executing in

Table 3.1 HLSL index variables.

main difference is the CS has read-only access to a a shader resource view (SRV), whilst the CS has read-and-write access to a unordered access view (UAV) at the cost of performance. UAVs does thus act as outputs from the compute shader and is the only type of view interface that can be used as an output buffer. Constant buffers can be passed straight to the shader without requiring the use of SRVs or UAVs.

Now all information needed to launch the CS is received. Although it requires a lot of code to set up buffers and view interfaces, the process of launching a CS is trivial. We first have to bind the shader to make it active, this is done by using the method `CSSetShader` with the corresponding CS instance as an argument. We can then pass the view interfaces and constant buffers to the CS in a similar fashion by calling `CSSetUnorderedAccessViews`, `CSSetShaderResources` or `CSSetConstantBuffers`. The CS is then launched by calling `Dispatch (X,Y,Z)`, where `X,Y,Z` specifies the number of thread groups to be launched in each dimension. The number of thread groups (blocks in CUDA, work groups in OpenCL) is thus specified from the host when dispatching the CS, the number of threads per group is specified on the device in the CS by using the `numthreads(X,Y,Z)` attribute, which specifies how many threads should be dispatched in each thread group. E.g a `Dispatch(8,1,1)` with a `numthreads (512,1,1)` would dispatch a compute shader with a total of 8 thread groups, each group containing 512 threads, thus the total number of threads would be 4096. The thread and group index can then be retrieved with the keywords listed in table 3.1. The result from the CS can then be retrieved by using the method `Map`, and copy the data back into its original container with a `memcpy`.

Since all data that is to be passed to the CS has to be bound to a buffer, there is no way of directly passing single data elements such as integers or floating points. The common practice of achieving this is to create a data structure containing all single element variables, and then generate a single element constant buffer with the data structure which is passed to

Buffertype	Description
cbuffer	Constant buffer
RWBuffer	Raw buffer
RWStructuredBuffer	Raw structured buffer

Table 3.2 HLSL buffer types.

Register type	Description
b	Constant buffer
t	Texture and texture buffer
c	Buffer offset
s	Sampler
u	Unordered Access View

Table 3.3 HLSL register types.

the CS. Since it is a constant buffer, this buffer can be passed to the CS without the need of a SRV or UAV.

The UAVs, SRVs and constant buffers are unlike OpenCL and CUDA not recieved as arguments to the kernel function, but as "global" variables inside the shader code. The buffer is recieved according to `bufferType<T> bufferName : register(Type#)` in the HLSL code, where the `#` specifies in what register slot the buffer is assigned, which is specified when setting the buffer from the host, and the `Type` is a single character describing the register type. Although many types of HLSL buffers exists, the ones used in this implementation are listed in table 3.2. The register types used in this implementation are listed in table 3.3.

Similar to the CUDA and OpenCL implementation, there are three datastructures that has to be passed to the CS in order to calculate the forces and update the positions:

- The positions of the particles.
- A container with more information about each particle such as it's velocity and mass.
- The flattened octree.

Both the position container and the particle container has to be updated in the CS, and are thus passed to the shader as UAVs. The flattened octree container however is unmodified in the shader and can be passed as a SRV to the shader for optimization purposes. To minimize the amount of overhead generated by copying data between the host and the device, the shader views are copied once before the force calculation CS is dispatched. There is no need to retrieve the result from this dispatch since the required data for the position update CS is already on the device. Once both dispatches has been finished, the data can be copied back into it's original containers.

Two types of CS was used in the implementation, one CS responsible for calculating the forces applied on each body in the system, and a second CS responsible for updating the positions. Since the positions have to be updated after the force calculation has been finished for each simulation step, two CS dispatches was made to be able to synchronize between the thread groups.

The first CS, the force calculation CS, was ported from the OpenCL force calculation kernel and is very similar. The CS responsible for updating the positions of the bodies is very trivial. Since the force calculation has been done before this shader is dispatched, all information needed to calculate the new position of the body is obtained. The position is updated in the same way as in CUDA and OpenCL by using Euler Integration. Since the positions have to be retrieved and read by the host to later be passed to the visualization, the position vector is converted into a raw buffer, and passed to the shader as an UAV.

The force calculation CS is more complex than the position update CS. Since HLSL does not support recursive calls, an iterative implementation using a queue was implemented. Although shader model 5.0 used in this implementation support C++ like data structures and classes with member functions and variables, some limitations exists. HLSL classes/data structure cannot contain member variables with initial values. This can however easily be avoided by e.g using a initialize member function. However when using a data structure method inside a loop, the loop is forced to be unrolled. When performing the iterative tree traversal using a queue, a while-loop is running until the queue is empty. The problem is that the while loop cannot be unrolled, and thus it's not a viable solution to use a data structure representing a queue. The solution to this was similar to how the same problem was solved in the OpenCL implementation as described in section 3.5, by in-lining the push/pop operations inside the CS. The actual force calculation is very similar to the force calculation described in section 3.4 and 3.5.

Chapter 4

Results

This section presents the results obtained by this evaluative study. Performance results for the various frameworks is presented, obtained by measuring the execution time of the implementations. Features of the different frameworks and the portability of the frameworks are presented. Finally, a simple full vector addition implementation in the various frameworks, including both the host and the device code, are presented and evaluated in terms of cyclomatic code complexity for the various frameworks.

4.1 Performance

To measure the performance of all frameworks when performing the N-Body simulation, the execution time was measured when simulating a fixed amount of timesteps for a dynamic range of bodies. The `std::chrono` library was used the measure the execution time in milliseconds. To minimize the performance degradation generated by the visualization, the visualization was disabled when the tests were run. The tests were performed in Microsoft Visual Studio 2012 by running the various implementations without the debugger since the debugger drastically degrades the performance. The code was compiled by Microsoft's Visual C++ 11.0. The specifications for the system that the tests were performed on are listed in table 4.1.

General	
Operating system	Windows 7 Professional x64
CPU	Intel Core i7-3770 @ 3.40GHz (8 CPUs)
GPU	NVIDIA GeForce GTX 1050
RAM	16 GB
GPU	
Manufacturer	NVIDIA
Model	GeForce GTX 1050
GPU Architecture	Pascal
Compute Capability	6.1
Global Memory	2048mb
Memory Speed	7 Gbps
Memory Bandwidth	12 GB/sec
No. Multiprocessors	5
CUDA Cores	640
Warp size	32
Frameworks	
CUDA	v9.1
OpenCL	v1.2 (NVidia)
DirectX 11	11
CS Shader Profile	CS 5.0

Table 4.1 System specifications on which the tests were performed.

The total execution time of the frameworks are presented in figure 4.1. The total execution time includes both the part of the algorithm that is executed on the host and the device when simulating N-Body systems in the range $N \in [1024, 20480]$ and was measured by simulating 100 timesteps for each N without using any visualization and calculating the average execution time for one timestep.

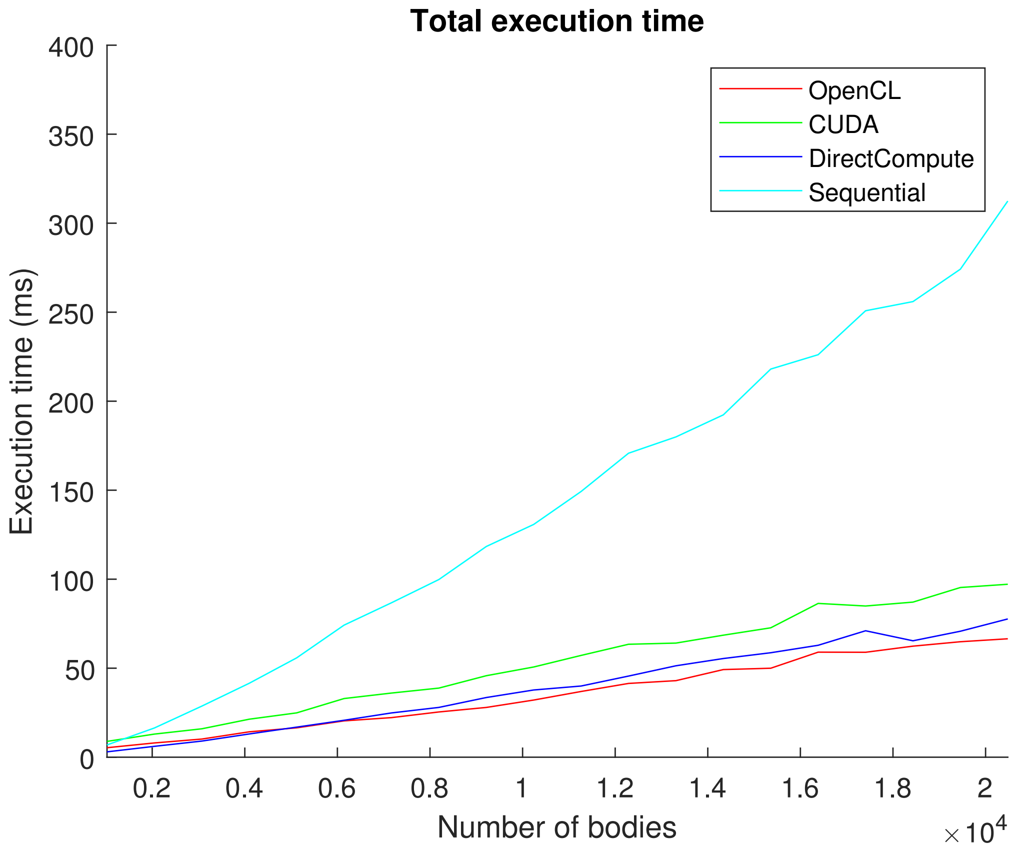


Fig. 4.1 Total execution time

The part of the algorithm that is executed on the device, i.e. the force calculation and position update was measured and is presented in figure 4.2.

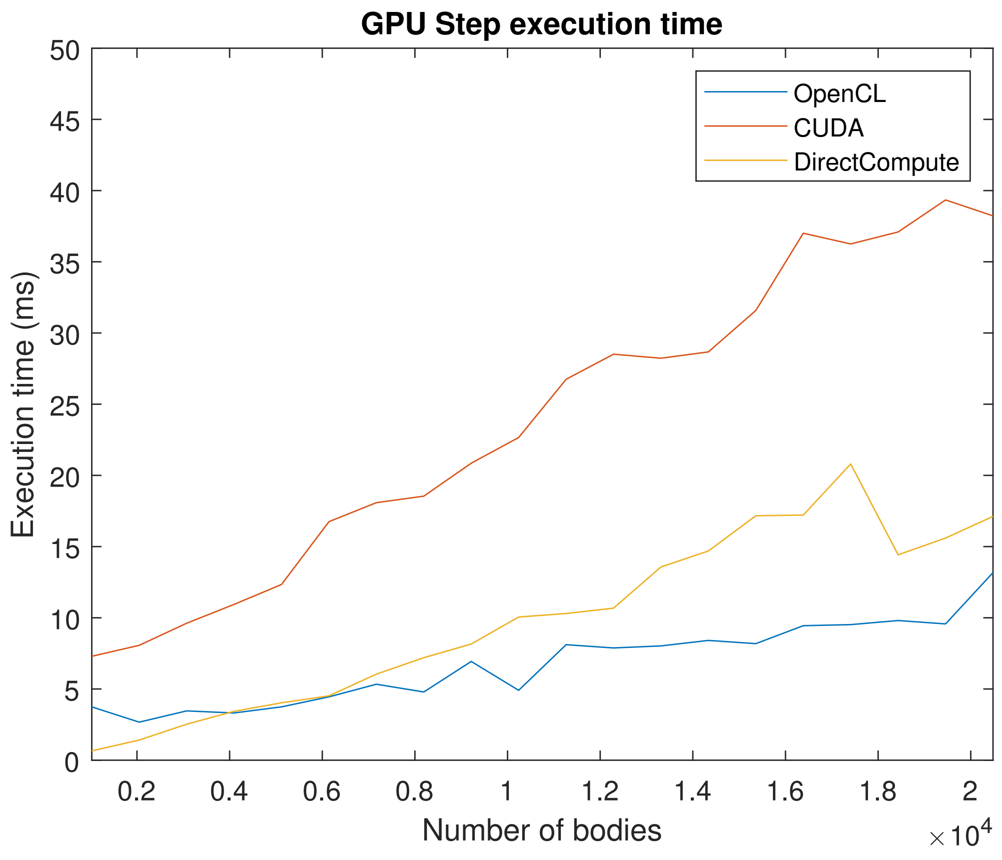


Fig. 4.2 Execution time of the GPGPU step

To be able to compare what parts of the algorithm that was the most time consuming, the total execution time was broken apart and separated into 4 major parts:

- **Build tree** - The time it takes for the CPU to for each timestep build the tree from the N-body system.
- **Calc. tree COM** - The time it takes for the CPU to traverse the tree and calculate each cells COM, total mass and the number of bodies in the subtree.
- **Flatten tree** - The time it takes for the CPU to flatten the pointer based tree into an container.
- **Step** - The time it takes to calculate the forces and update the positions. Executed on the GPU (apart from the sequential implementation).

The measured execution times for these parts are presented in figure 4.3, 4.4, 4.5 and 4.6. The sequential implementation has no need to flatten the tree every timestep as discussed in section 3.3.2, why this is extradited from the graph.

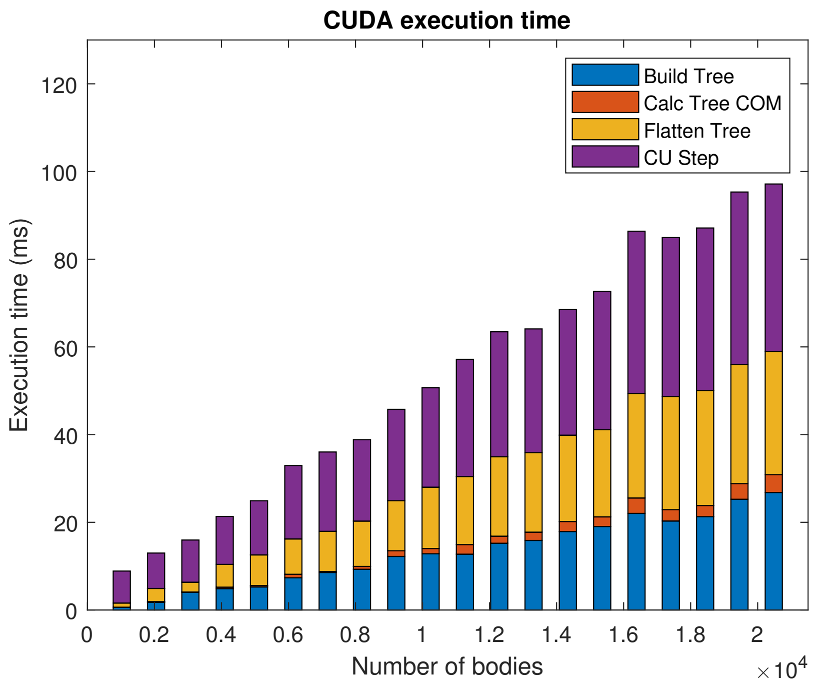


Fig. 4.3 CUDA execution time

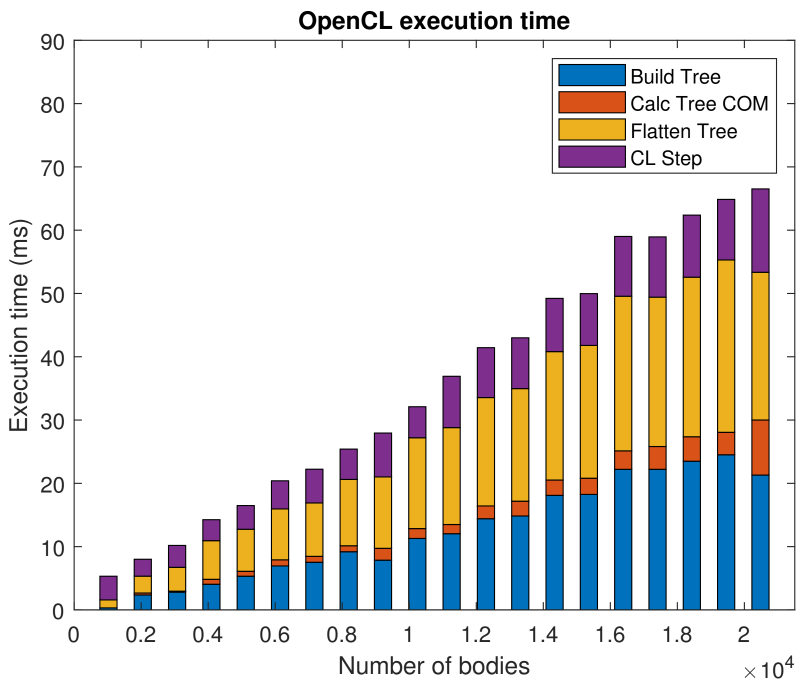


Fig. 4.4 OpenCL execution time

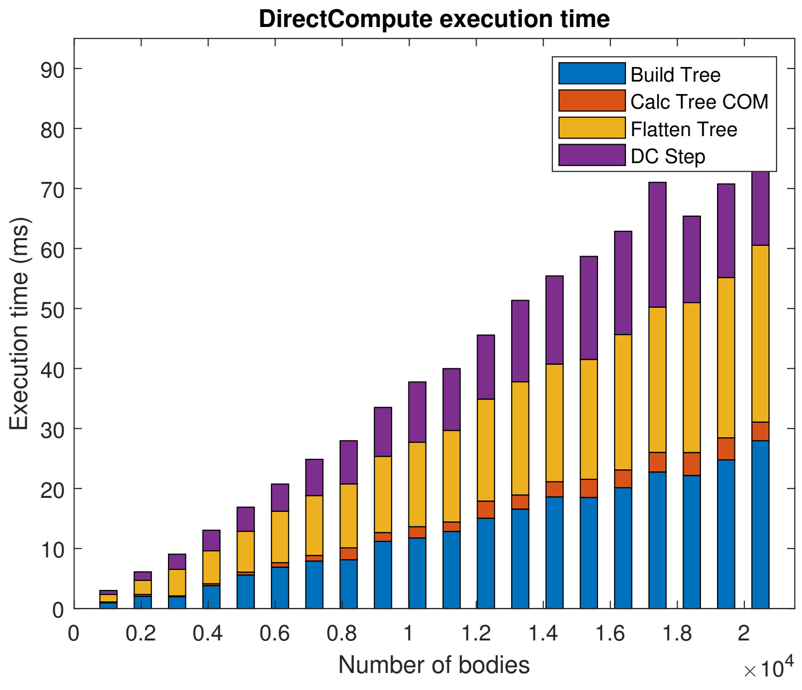


Fig. 4.5 DirectCompute execution time

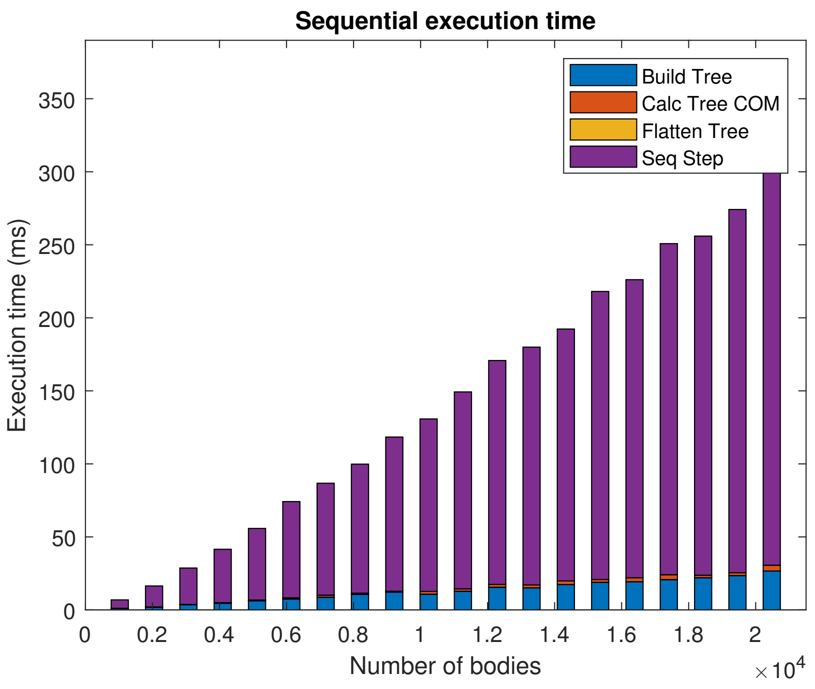


Fig. 4.6 Sequential execution time

To be able to compare the performance difference when using buffers containing data structures and buffers containing class objects in CUDA, the execution time for both these cases was measured, the result is presented in figure 4.7. Furthermore to be able to compare the performance for CUDA when using recursion in the force calculation as discussed in section 3.4, the execution time for this was measured and is presented in figure 4.8. The default stack size in CUDA is only 1024 bytes and the recursive force calculation gets deep. The stack size was thus increased to 2048 bytes when testing the recursive implementation to prevent stack overflow from occurring.

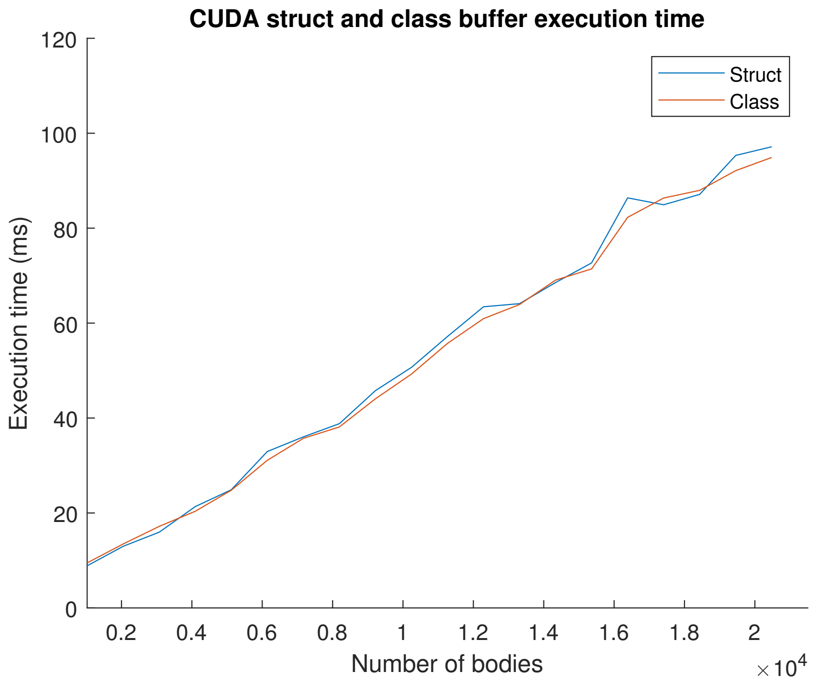


Fig. 4.7 Execution time, CUDA Struct vs Class object buffers

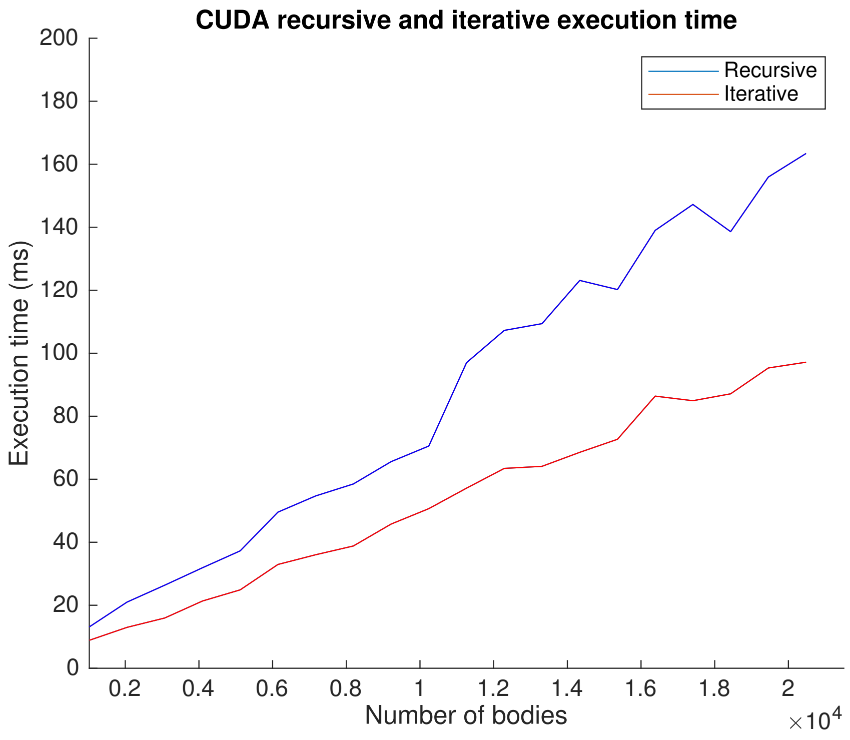


Fig. 4.8 Execution time, CUDA recursive force calculation vs iterative force calculation

4.2 Features

The features of the various frameworks given the specifications that are listed in table 4.1 are listed in table 4.2 [2][3][4].

The table lists the feature specifications for OpenCL 1.2, which is the newest Nvidia version and was used in this implementation. OpenCL 2.0 brought major differences which makes it a strong candidate feature-wise compared to CUDA, and OpenCL 2.0 supports the OpenCL C++ kernel language, which thus makes it possible to create and utilize classes in the kernel. OpenCL 2.0 also adds the ability to launch kernels from within kernels, i.e Dynamic parallelism.

Furthermore, the table lists specifications for DirectCompute when run on Direct3D 11.x hardware and thus using the CS 5.0 model. Older hardware (Direct3D 10.x) only supports the CS 4.0 model which adds some further restrictions. The main differences is that while CS 5.0 supports a max number of 1024 threads per group, CS 4.0 only supports 768. CS 4.0 does not support 3D grid dimensions, and have no atomic operations, scatter operations and does not support double precision. Finally, while CS 5.0 supports eight UAVs which can be bound to a shader, CS 4.0 only allows for one UAV.

Features			
	CUDA	**OpenCL (1.2)**	**DirectCompute (D3D11)**
Kernel Language	Cuda C/C++	OpenCL C	HLSL
Kernel classes	Yes	No	No
Kernel recursion	Yes	No	No
Dynamic parallelism	Yes	No	No
Class object buffers	Yes	No	No
Structured buffers	Yes	Yes	Yes
Warp size	32	32	32
Atomic Operations	Yes	Yes	Yes
64 bit precision	Yes	Yes	Yes
3D Grid Dimensions	Yes	Yes	Yes
No. Threads/group	1024	1024	1024
Thread local mem.	512 KB	-	-
Shared mem./group	48 KB	42 KB	16 KB
Constant mem. size	64 KB	64 KB	64 KB
Group synchronization	Yes	Yes	Yes
Gather operations	Yes	Yes	Yes
Scatter operations	Yes	Yes	Yes

Table 4.2 Framework features 4.1.

4.3 Portability

This section will describe the portability of the various frameworks. Limitations for each of the evaluated frameworks and the portability based on market share statistics for operating systems and GPU vendors are discussed.

OpenCL was developed with portability in mind, and is able to be utilized on systems with compatible hardware. Since OpenCL is able to perform parallel computations on a wide variety of devices, the portability of the framework is high and works on most systems.

CUDA only available on Nvidia GPUs, newer than the 8800 series, codenamed G80 (2006). Cuda is obtained by downloading and installing the Nvidia CUDA Toolkit and is available on all major operating systems.

DirectCompute which is a part of the DirectX suite, only works on Microsoft Windows systems with Direct3D 10.0 and up. DirectX 11 is included in systems running Windows 7 or newer which as of February 2018 is 96% of all windows systems, see fig 4.9. [13]

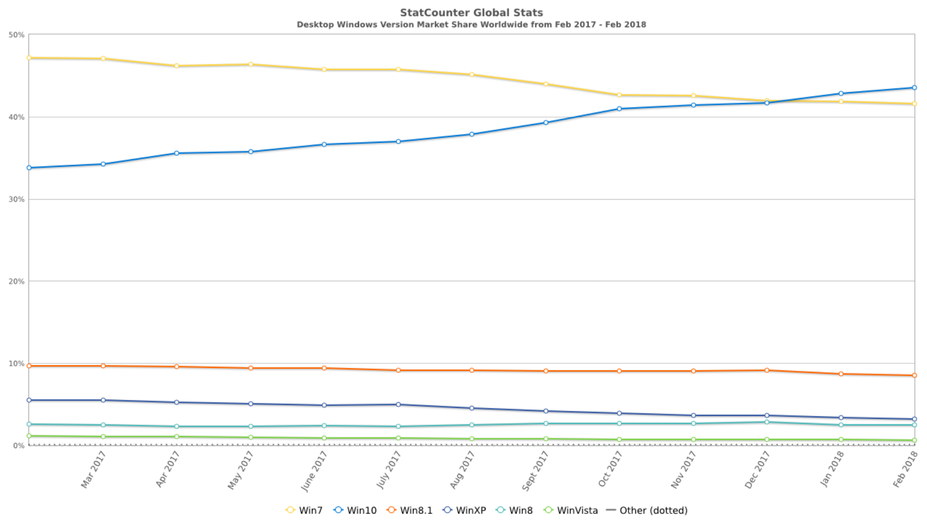
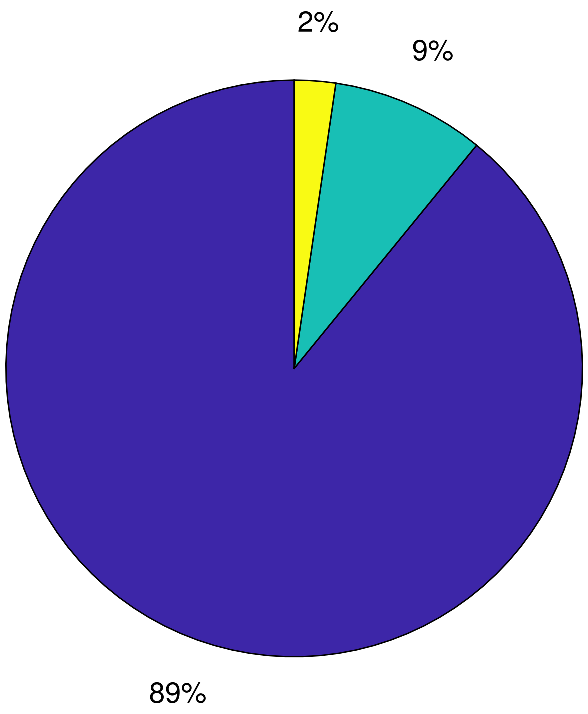


Fig. 4.9 Desktop Windows Version Market Share Worldwide

The market share of the most popular desktop and laptop system operating systems Windows, OSX and Linux, are presented in figure 4.10 (Q1 2018) [8]. The market share of the GPU vendors AMD and Nvidia are listed in figure 4.11 (Q4 2017) [6].



🟦	Windows
🟩	Mac OS
🟨	Linux

Fig. 4.10 Operating system market share (Q1 2018) [8]

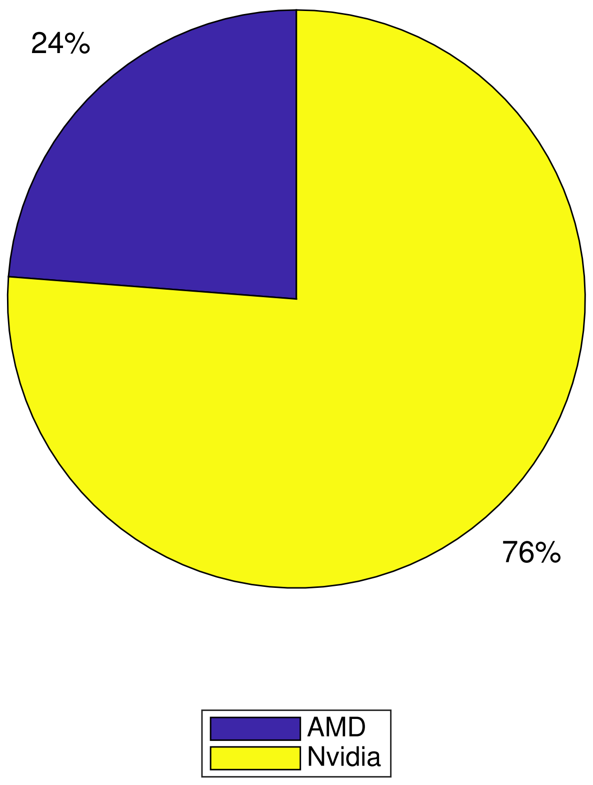


Fig. 4.11 AMD and Nvidia market share (Q4 2017) [6]

4.4 Complexity

Appendix A, B and C lists full code examples which performs a simple vector addition according to $a + b = c = (a_1 + b_1, a_2 + b_2, ..., a_n + b_n)$ in CUDA, OpenCL and DirectCompute accordingly. To measure the complexity of the frameworks, these implementations was tested using the freeware software SourceMonitor [12] which measures the cyclomatic complexity, lines of code (not counting comments or empty lines), and the control flow graph depth. This gives a more intuitive insight of the complexity of the discussed frameworks than if the N-body implementations was to be measured. The results of the measurements are presented in table 4.3.

CUDA Vector Addition	
Lines	45
Statements	38
Max Complexity	4
Avg. Complexity	0.97
Max Depth	3
Avg. Depth	0.97
OpenCL Vector Addition	
Lines	68
Statements	42
Max Complexity	4
Avg. Complexity	2.5
Max Depth	3
Avg. Depth	0.95
DirectCompute Vector Addition	
Lines	241
Statements	144
Max Complexity	9
Avg. Complexity	4.14
Max Depth	5
Avg. Depth	1.31

Table 4.3 Metrics for the vector addition examples listed in Appendix A, B and C

Chapter 5

Discussion

This chapter discusses the results obtained from this study, as well as an objective and a subjective conclusion of the discussed frameworks.

5.1 Performance

Although not in focus in this study, the performance is always an interesting factor when comparing frameworks, and although a lot of comparison studies have been made between CUDA and OpenCL, DirectCompute is usually left out of the picture. The outcome from these kind of studies tend to show that CUDA usually outperforms the other frameworks, although studies show that the performance can be very similar under a fair comparison as previously discussed in section 1.5.1.

Figure 4.1 shows the total execution time for the various frameworks, as well as the execution time for the sequential implementation. As expected, the parallel GPGPU implementations outperform the sequential implementation considerably. For small N-body systems ($0.2 * 10^4$), the performance of the sequential compared to the parallel implementations are very similar but as the size of the problem grows the parallel implementations clearly outperforms the sequential implementation and for very large problem sizes ($2.0 * 10^4$) the speedup between the sequential and the fastest parallel implementation OpenCL is 4.69.

Surprisingly, CUDA which is usually the fastest framework as discussed in section 1.5.1, is the slowest of the evaluated frameworks in this implementation which is more clearly presented in figure 4.2. There could be many reasons why this is the case. Due to the nature of how the tree is structured and traversed, it is difficult to achieve coalesced memory accesses in the kernel which may prove that CUDA handles non coalesced memory accesses more poorly than the other frameworks.

Furthermore, we can see that the although more unfamiliar framework DirectCompute is a strong competitor to both CUDA and OpenCL. For small problem sizes, DirectCompute outperforms both CUDA and OpenCL, and has a similar performance compared to OpenCL for bigger problem sizes.

Visualized in figure 4.3-4.6, we can see that the parts of the algorithm that is performed on the device, i.e. the force calculation and position update scales better than the tree construction and the tree flattening which are the main bottlenecks of the performance. Although more complex, both of these steps can be performed in parallel on the device as described by M. Burtscher and K. Pingali [18]. By moving these steps, along with the COM calculation, the overhead spawned by coping the data back and forth between the host and the device every timestep would be eliminated and may further increase the performance of the application.

Figure 4.6 shows the execution time of the sequential implementation, and shows that the force calculation and the position update scales very poorly when run sequentially and is thus the parts of the algorithm that are the most suited for beeing parallelized, which was done in this implementation.

Out of the evaluated frameworks, CUDA is the only one able to copy class objects in a buffer to the kernel. This eliminates the need to convert the class object based octree into a data structure based octree. Although this data conversion is a fast operation which is included in the Calc. Tree COM step, it is still interesting to compare how the use of class objects buffers affect the performance. Figure 4.7 shows the execution time for CUDA when using structured buffers and class object buffers. The execution time of these are very similar and we can conclude that the usage of class object buffers in this case does not hurt the performance, while at the same time keeps the code less complex.

Another feature only available in CUDA is the ability to use recursion in kernels. This can be utilized in the force calculation when traversing the tree as discussed in section 3.4. The execution time when using a recursive tree traversal was measured and compared to the iterative tree traversal used in the other implementations. The result of this comparison is presented in figure 4.8. As expected, the recursive implementation is more time consuming. The most likely cause of this is due to the amount of overhead spawned when allocating stack memory.

5.2 Features

To be able to compare the features of the frameworks, the documentation of the frameworks was studied and summarized in table 4.2. The table summarizes some of the most important features of the respective framework and although a lot of the features are shared, CUDA has the advantage feature wise. Features such as kernel classes, recursion, dynamic parallelism and class object buffers are only supported in CUDA. These features are however very handy and can simplify and abstract the code significantly.

Out of the features listed in table 4.2 only available in CUDA, kernel classes, kernel recursion and class object buffers was used. This made the readability and understandably of the code much better since it abstracted away a lot of the complexity.

Since neither OpenCL 1.2 or DirectCompute supports classes or C++ like data structures inside the kernel/CS code, the queue had to be inlined as discussed in section 3.5 and 3.6 which decreases the readability of the application. For applications using larger more complex kernels the ability to be able to write object oriented code may be an important feature, and if the entirety of this implementation would be performed on the device, this feature would be of most importance, making this a strong feature for CUDA to have.

Recursion was utilized when calculating the forces which also simplified the complexity and readability. Although recursion simplified the tree traversal, it came at the cost of performance as discussed in section 5.1. Since the overhead spawned by allocating stack memory for the recursive calls is difficult to avoid, recursion often hurt the performance and whether it should be used is a question about the performance-readability trade-off.

Although not used in this implementation, dynamic parallelism is a feature only available in CUDA (available in OpenCL 2.0, see 4.2) which could be an alternative to recursive function calls. Since it was not used in this implementation it is difficult to tell how dynamic parallelism would affect the performance, although in a paper by J. DiMarco and M. Taufer where dynamic parallelism was used on different clustering algorithms, speedups up to 3.03 times was observed and dynamic parallelism scaled better for larger problem sizes than when not using dynamic parallelism [21]. Another paper by J. Wang and S. Yalamanchili showed that using dynamic parallelism could achieve 1.13x-2.73x potential speedup but the overhead caused by the kernel launching could negate the performance benefit. [41]

5.3 Portability

To be able to compare the portability of the different frameworks and get a better understanding of the extent on which a framework may run on, market share statistics for the most popular operating systems and the two major GPU vendors for desktop systems Nvidia and AMD are presented in figure 4.10 and 4.11.

OpenCL

As discussed in section 4.3, OpenCL is designed to be portable and can be run on almost any device. Since almost all systems have either a GPU or a multicore CPU, both which OpenCL supports, most systems are able to utilize the framework. When speaking in terms of utilizing OpenCL for GPGPU however, the portability depends on whether the GPU hardware developer supplies an OpenCL implementation. However both AMD and Nvidia which are the two major GPU vendors today have OpenCL implementations, making OpenCL the most portable of the evaluated frameworks.

CUDA

CUDA can only be utilized on systems with a Nvidia GPU with CUDA support. Although this restriction, Nvidia supplies the CUDA toolkit for the three most popular desktop operating systems Windows, macOS and Linux, making it OS independent. Figure 4.11 also shows that as of Q4 2017 Nvidia controls 76% of the market share making the portability of CUDA strong.

DirectCompute

DirectCompute, as discussed in section 4.3 is only available on Windows systems with Direct3D 10.0 or higher. As discussed in section 3.6 and shown in figure 4.9, based on the Windows market share, 96% of the windows systems does have support for DirectX 11. As visualized in figure 4.10, Windows control 89% of the market share, making it not as portable as CUDA nor OpenCL but a strong alternative on the Windows platform since its shipped with the OS and no further installation is required.

5.4 Complexity

As discussed in section 4.4, to be able to get a good understanding of the complexity of the various frameworks, a vector addition application was implemented in all frameworks. The code for these are given in Appendix A, B and C. The result of the measurements are summarized in table 4.3.

Out of the evaluated frameworks, CUDA is the least complex with a max complexity of 4, and an average complexity of 0.97 which is also reflected in the attached code example in Appendix A. With only 45 lines of code, a working CUDA vector addition can be implemented. Furthermore, since CUDA uses its own NVCC compiler, the device code can be included in the same file as the CUDA host code, which may increase the readability of the implementation. The actual kernel launch is very similar to a normal function call, making it more intuitive.

With a max complexity of 4, and a average complexity of 2.5, the OpenCL vector addition implementation is the second least complex. Since the developer has to specify the target device, the additional step of finding a compatible device and creating a context has to be implemented, and thus the size of the OpenCL implementation grows to 68 lines. Although the feature of selecting a device is possible in CUDA as well, it is not necessary since the framework per default selects the default GPU. The process of copying data to the device is done by using buffer objects and filling the buffers with the relevant data, whilst in CUDA the data can be copied directly to the device. The usage of buffers adds another step to the process, further decreasing the intuitiveness of the application. The actual copying of the data and the kernel launch is done by using a command queue. For a developer without a good knowledge in GPGPU, this may further decrease the intuitiveness of the application.

The most complex of the vector addition examples is the DirectCompute implementation with a max complexity of 9 and an average complexity of 4.41. This is also reflected in the size of the application with 241 lines, compared to only 45 in CUDA and 68 in OpenCL. One of the reason why this is the case is because Direct3D 10 and 11 uses different functions for doing the same thing, why these cases has to be tested. Similar to OpenCL, DirectCompute also requires the developer to specify the target compute device and create the context, making the implementation a bit more cumbersome than CUDA. Also similar to OpenCL, DirectCompute handles the data transfer between the host and the device by using buffers, although a second step is required before the buffer can be copied to the device. The buffers have to be converted into access view, readable by the compute shader, making the intuitiveness worse. Although the process of dispatching the CS is trivial, the process of retrieving the data to the host is a bit more complex and requires the usage of a debug buffer, as well as mapping the resulting data to a mapped sub-resource, making the implementation more complex.

From a subjective point of view, CUDA is the most intuitive of the various frameworks. It's API is well developed and forces the developer to write well structured and readable code. Out of the evaluated frameworks CUDA is the one that mostly resembles sequential programming which most developers are used to. Although OpenCL is similar to CUDA in many ways, it more resembles graphical programming with the use of buffers. DirectCompute is the most complex of the frameworks, but DirectX or similar graphics programming experience may facilitate the implementation and understanding. Furthermore, as illustrated in figure 2.4, CUDA and OpenCL is the most popular GPGPU frameworks and is well documented by the GPGPU community, making it easy to find sample applications and getting help from the community which may also be an important factor. Since DirectCompute is relatively unknown, this is more difficult and sample applications using DirectCompute is difficult to find. The only examples found in this work was examples implemented by Microsoft.

5.5 Encountered problems

This section describes problems that arose during the implementation, and how they were solved. Different approaches to how these problems could be solved is also discussed.

Copy the Barnes-Hut Octree to the device

One of the encountered problems that arose during the implementation phase was how to copy the Octree, built on the host, to the device in order to calculate the net forces applied on the bodies. The problem is that all of the evaluated frameworks requires information about how large an element that is to be copied to the device is. The pointer based octree is a very dynamic data structure and since it is rebuilt each step, might vary greatly in size. A second problem with the pointer based octree is that the pointers point to memory location on the host, which thus has to be updated after the octree has been copied to the device.

One way in which this could be solved would be to traverse the octree using either a recursive or iterative tree traversal algorithm, calculating the size as it traverses the octree. However, this solution would still make the octree reliant on a pointer based structure which would result in a lot of pointer chasing on the device.

The second and more common solution is to flatten the tree into a simpler data structure. Similarly the tree has to be traversed and each node of the octree has to be copied to the simpler data structure such as an array. This was the choosen solution, which also eliminated the need for pointers since each node in the tree can be represented as an index in the array.

Iteratively traverse the octree

Since CUDA is the only framework which allows recursion in the kernel code, a way to traverse the octree in an iterative manner had to be explored. This is however a common problem which is usually solved by using either a stack or a queue.

Since there exists no queue or stack functionality in any of the frameworks kernel code, a simple queue was implemented. In CUDA, which supports C++ like features, the implementation was made as a structure with member functions and default variable values. Since neither DirectCompute or OpenCL 1.2 which was used in this implementation supports these C++ features, the queue code had to be inlined, resulting in a more complex code structure.

Copying class objects to the kernel

Out of the frameworks that was evaluated in this study, CUDA is the only framework that supports class objects in the buffer which is passed to the kernel. Both DirectCompute and OpenCL (1.2) does however support buffers containing data structures. The solution was thus to convert the class object based octree into data structure, which could be done whilst flattening the tree to avoid unnecessary performance degradation.

Chapter 6

Conclusion

This work shows that the N-body problem, along with the Barnes-Hut optimization algorithm is adaptable to be performed in parallel on a GPU. Tests show that when the problem grows large, the parallel implementation scales better than a sequential implementation, although the implementation could be significantly optimized by moving the entire implementation to be performed on the device. The most computational demanding parts of the algorithm is the force calculation and position updating as shown in figure 4.6, why these steps are the main targets for parallelization.

Tests showed that the fastest framework OpenCL, outperformed the sequential implementation with a speedup of 4.69 for large problem sizes. CUDA which usually is the fastest framework as discussed in section 1.5, was the slowest in this implementation. Reasons why this is the case are discussed in section 5.1.

Furthermore, the N-body problem using the Barnes-Hut algorithm investigates the use of complex data structures. Tree representations and method of tree traversals on the GPU have been described in chapter 3 using iterative or recursive solutions where results show that iterative solutions show a better performance.

One of the investigation areas of the frameworks was the features of the discussed frameworks. To compare these, the documentation of the frameworks was studied and was summarized in table 4.2. Features such as kernel recursion, dynamic parallelism and class object buffers are only available in CUDA, although some features are available in newer versions of OpenCL as mentioned in section 4.2. These features can be considered as strong and which only are available in CUDA, making it the strongest framework in terms of features.

Another important aspect when choosing a framework is the simplicity and intuitiveness of the framework. To estimate this, simple vector addition applications was implemented which are listed in Appendix A, B and C. Measurements was done on these implementations where the lines of code and the cyclomatic complexity was measured and the results of these measurements are presented in table 4.3 and further discussed in section 4.4. With 45 lines of code, a simple CUDA vector addition was implemented, OpenCL required 68 lines of code and DirectCompute requires 241. DirectCompute also had the highest cyclomatic complexity of 9, which is also reflected in the large amount of code needed to get a simple DirectCompute application running.

To evaluate the portability of the various framework, conclusions are based on market share statistics of the most popular operating systems and GPU developers are used as a base to get a better understanding about the portability. The results show that OpenCL is the most portable of the discussed frameworks as it is able to run on the most platforms. CUDA and DirectCompute are OS and device restricted, and the portability of these frameworks are discussed and evaluated using the market share statistics as a base in section 5.3.

The development of the application started by making sequential implementation. With this as a base, the first parallel implementation was made in CUDA, which was later ported to OpenCL and DirectCompute. The reason why CUDA was the selected framework for the initial parallel implementation was because CUDA has the most intuitive API and most features. The parallel CUDA implementation was then ported to OpenCL and DirectCompute, but since CUDA specific features was used, parts of the implementation had to be refactored. A better approach to the implementation would be to implement the most complex framework, i.e DirectCompute first, and then port the DirectCompute implementation to the other frameworks.

Furthermore, the tree architecture in its current state is complex. Since this is one of the core implementations of the applications, a re-implementation of the tree would result in a large amount of refactoring, thus the tree implementation should have been simplified at the start of the development process.

6.1 Future work

A topic for future work could be to move the entirety of the algorithm to be performed on the device as the main bottleneck of the current implementation is the tree construction and tree flattening steps as discussed in section 5.1. Although the implementation would be more complex, this would be an interesting topic and the performance comparison between the application in it's current stage could be compared to an implementation where the entire algorithm is performed on the device. The performance is expected to drastically increase since the time consuming step of flattening the tree would be redundant.

Furthermore, dynamic parallelism is a feature that might suit this type of problem. It would be interesting to see how a implementation using dynamic parallelism would compare to the implementation in it's current stage.

References

[1] About cuda. https://developer.nvidia.com/about-cuda. Accessed: 2018-01-25.

[2] Cuda programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2018-03-27.

[3] Directcompute programming guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/DirectCompute/doc/DirectCompute_Programming_Guide.pdf, . Accessed: 2018-03-27.

[4] Directcompute optimizations and best practices. http://on-demand.gputechconf.com/gtc/2010/presentations/S12312-DirectCompute-Pre-Conference-Tutorial.pdf, . Accessed: 2018-03-27.

[5] Direct3d 11 features. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476342(v=vs.85).aspx. Accessed: 2018-03-13.

[6] Nvidia and amd discrete gpu market share report for q3 2017. https://wccftech.com/nvidia-amd-discrete-gpu-market-share-report-q3-2017/. Accessed: 2018-04-03.

[7] Opencl overview. https://www.khronos.org/opencl/. Accessed: 2018-01-25.

[8] Net marketshare. https://netmarketshare.com. Accessed: 2018-04-03.

[9] Directcompute. https://developer.nvidia.com/directcompute. Accessed: 2018-01-25.

[10] Intro to opencl c++. https://www.khronos.org/assets/uploads/developers/resources/Intro-to-OpenCL-C++-Whitepaper-May15.pdf, . Accessed: 2018-02-26.

[11] Opengl documentation. https://www.khronos.org/registry/OpenGL-Refpages/, . Accessed: 2018-02-02.

[12] Sourcemonitor version 3.5. http://www.campwoodsw.com/sourcemonitor.html. Accessed: 2018-03-28.

[13] Desktop windows version market share worldwide. http://gs.statcounter.com/windows-version-market-share/desktop/worldwide/. Accessed: 2018-03-13.

[14] Cpu vs gpu performance. http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html. Accessed: 2018-04-03.

[15] Sverre J Aarseth and Sverre Johannes Aarseth. *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003.

[16] Virasin Archirapatkave, Hendra Sumilo, Simon Chong Wee See, and Tiranee Achalakul. Gpgpu acceleration algorithm for medical image reconstruction. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 41–46. IEEE, 2011.

[17] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446, 1986.

[18] Martin Burtscher and Keshav Pingali. An efficient cuda implementation of the tree-based barnes hut n-body algorithm. In *GPU computing Gems Emerald edition*, pages 75–92. Elsevier, 2011.

[19] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA)*, 14:4, 2009.

[20] Shifu Chen, Jing Qin, Yongming Xie, Junping Zhao, and Pheng-Ann Heng. A fast and flexible sorting algorithm with cuda. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 281–290. Springer, 2009.

[21] Jeffrey DiMarco and Michela Taufer. Performance impact of dynamic parallelism on different clustering algorithms. In *Modeling and Simulation for Defense Systems and Applications VIII*, volume 8752, page 87520E. International Society for Optics and Photonics, 2013.

[22] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[23] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[24] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, pages 9–18. ACM, 2007.

[25] Leslie Greengard. *The rapid evaluation of potential fields in particle systems*. MIT press, 1988.

[26] Stefan Gustavson and Robin Strand. Anti-aliased euclidean distance transform. *Pattern Recognition Letters*, 32(2):252–257, 2011.

[27] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[28] Gunnar Johansson and Hamish Carr. Accelerating marching cubes with graphics hardware. In *CASCON*, volume 6, page 39. Citeseer, 2006.

[29] Mark W Jones, J Andreas Baerentzen, and Milos Sramek. 3d distance fields: A survey of techniques and applications. *IEEE Transactions on visualization and Computer Graphics*, 12(4):581–599, 2006.

[30] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[31] R. Stanley Williams Kirk M. Bresniker, Sharad Singhal. *Adapting to Thrive in a New Economy of Memory Abundance*. 2015.

[32] Julio F Navarro. The structure of cold dark matter halos. In *Symposium-international astronomical union*, volume 171, pages 255–258. Cambridge University Press, 1996.

[33] Lars Nyland, Mark Harris, Jan Prins, et al. Fast n-body simulation with cuda. *GPU gems*, 3(1):677–696, 2007.

[34] Rafael Sachetto Oliveira, Bernardo Martins Rocha, Ronan Mendonça Amorim, Fernando Otaviano Campos, Wagner Meira, Elson Magalhães Toledo, and Rodrigo Weber dos Santos. Comparing cuda, opencl and opengl implementations of the cardiac monodomain equations. In *International Conference on Parallel Processing and Applied Mathematics*, pages 111–120. Springer, 2011.

[35] I Present. Cramming more components onto integrated circuits. *Readings in computer architecture*, 56, 2000.

[36] Ingemar Ragnemalm. Lecture notes in information coding / computer graphics, isy, lith. http://computer-graphics.se/multicore. Accessed: 2018-01-30.

[37] Peter Sanders and Thomas Hansch. Efficient massively parallel quicksort. In *International Symposium on Solving Irregularly Structured Problems in Parallel*, pages 13–24. Springer, 1997.

[38] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2): 118–141, 1995.

[39] Torbjörn Sörman. Comparison of technologies for general-purpose computing on graphics processing units. Master's thesis, Linköping University, 2016.

[40] Chris J Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 306–317. IEEE, 2002.

[41] Jin Wang and Sudhakar Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 51–60. IEEE, 2014.

Appendix A

CUDA Vector Addition

```
#pragma once
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <iostream>

const unsigned int SIZE = 1024;

// addition kernel
__global__ void add(const int *in_a, const int *in_b, int *out)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < SIZE)
        out[idx] = in_a[idx] + in_b[idx];
}

int main(){

    // Host pointers for io data
    int *a = new int[SIZE];
    int *b = new int[SIZE];
    int *c = new int[SIZE];

    // Device pointers
    int *d_a, *d_b, *d_c;

    for (int i = 0; i < SIZE; i++){
        a[i] = i;
        b[i] = 2*i;
    }

    // Allocate memory on the device
    const unsigned int size = SIZE * sizeof(int);
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);

    // Copy the input data to the device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    dim3 dimGrid(1);
    dim3 dimBlock(SIZE);

    // Launch kernel with the given grid and block dimensions
    add <<<dimGrid, dimBlock >>> (d_a, d_b, d_c);

    // Retrieve the result
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Assert that the result is correct
    for (int i = 0; i < SIZE; i++){
        if (a[i] + b[i] != c[i])
            return 1;
    }
    std::cout << "Sucess!" << std::endl;

    // Clean up
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    delete[] a; delete[] b; delete[] c;

    return 0;
}
```

Appendix B

OpenCL Vector Addition

```
#include <CL/cl.hpp>
#include <iostream>
#include <vector>

const int N_elements = 1024;

// Kernel source, is usually stored in a seperate .cl file
std::string src =
"__kernel void vectorAddition("
"    __global read_only int* vector1, \n"
"    __global read_only int* vector2, \n"
"    __global write_only int* vector3)\n"
"{\n"
"    int indx = get_global_id(0);\n"
"    vector3[indx] = vector1[indx] + vector2[indx];\n"
"}\n";

int main()
{

    // Get list of available platforms
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    // Get list of devices
    std::vector<cl::Device> devices;
    platforms[0].getDevices(CL_DEVICE_TYPE_ALL, &devices);

    // Create a context from the devices
    cl::Context context(devices);

    // Compile kernel
    cl::Program program(context, cl::Program::Sources(1, std::make_pair(
        src.c_str(), src.length() + 1)));
    cl_int err = program.build(devices, "-cl-std=CL1.2");

    // Input data
    int* vector1 = new int[N_elements];
    int* vector2 = new int[N_elements];

    for (size_t i = 0; i < N_elements; i++){
        vector1[i] = i;
        vector2[i] = 2 * i;
    }

    // Create buffers from input data
    cl::Buffer vec1Buff(context, CL_MEM_READ_ONLY, sizeof(int) *
        N_elements, vector1);
    cl::Buffer vec2Buff(context, CL_MEM_READ_ONLY, sizeof(int) *
        N_elements, vector2);

    int* vector3 = new int[N_elements];
    cl::Buffer vec3Buff(context, CL_MEM_WRITE_ONLY, sizeof(int) *
        N_elements, vector3);

    // Pass arguments to the vector addition kerel
    cl::Kernel kernel(program, "vectorAddition", &err);
    kernel.setArg(0, vec1Buff);
    kernel.setArg(1, vec2Buff);
    kernel.setArg(2, vec3Buff);

    // Create command queue and copy data to the device
    cl::CommandQueue queue(context, devices[0]);
    queue.enqueueWriteBuffer(vec1Buff, CL_TRUE, 0, sizeof(int) *
        N_elements, vector1);
    queue.enqueueWriteBuffer(vec2Buff, CL_TRUE, 0, sizeof(int) *
        N_elements, vector2);

    // Launch kernel
    queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(
        N_elements), cl::NDRange(1024));

    // Read back result
    queue.enqueueReadBuffer(vec3Buff, CL_TRUE, 0, sizeof(int) *
        N_elements, vector3);

    // Assert that the result is correct
    for (size_t i = 0; i < N_elements; i++){
        if (vector1[i] + vector2[i] != vector3[i])
            return 1;
    }

    std::cout << "Sucess!" << std::endl;

    delete[] vector1;
    delete[] vector2;
    delete[] vector3;

    return 0;
}
```

Appendix C

DirectCompute Vector Addition

C.1 Vector Addition Compute Shader (.hlsl)

```
// Vector addition compute shader
RWBuffer<int>       Buffer0     : register(u0);
RWBuffer<int>       Buffer1     : register(u1);
RWBuffer<int>       BufferOut   : register(u2);

[numthreads(1, 1, 1)]
void CSMain(uint3 DTid : SV_DispatchThreadID)
{
    BufferOut[DTid.x] = Buffer0[DTid.x] + Buffer1[DTid.x];
}
```

C.2 Vector Addition main (.cpp)

```

#include <stdio.h>
#include <d3d11.h>
#include <d3dcompiler.h>
#include <iostream>

#include "dxerr.h"


#pragma comment(lib,"d3d11.lib")
```

```
#pragma comment(lib,"d3dcompiler.lib")

#ifndef SAFE_RELEASE
#define SAFE_RELEASE(p)      { if (p) { (p)->Release(); (p)=nullptr; } }
#endif
#ifndef CHECK_ERR
#define CHECK_ERR(x)    { hr = (x); if( FAILED(hr) ) { DXTraceW( __FILEW__, __LINE__, hr, L#x, true ); exit(1); } }
#endif


#define NUM_ELEMENTS     1024


HRESULT CreateComputeDevice(ID3D11Device** deviceOut, ID3D11DeviceContext** contextOut, bool bForceRef){
    *deviceOut = nullptr;
    *contextOut = nullptr;

    HRESULT hr = S_OK;

    // We will only call methods of Direct3D 11 interfaces from a single thread.
    UINT flags = D3D11_CREATE_DEVICE_SINGLETHREADED;
    D3D_FEATURE_LEVEL featureLevelOut;
    static const D3D_FEATURE_LEVEL flvl[] = { D3D_FEATURE_LEVEL_11_0, D3D_FEATURE_LEVEL_10_1, D3D_FEATURE_LEVEL_10_0 };

    bool bNeedRefDevice = false;
    if (!bForceRef)
    {
        hr = D3D11CreateDevice(nullptr,      // Use default graphics card
                D3D_DRIVER_TYPE_HARDWARE,    // Try to create a hardware accelerated device
                nullptr,                     // Do not use external software rasterizer module
                flags,                       // Device creation flags
                flvl,
                sizeof(flvl) / sizeof(D3D_FEATURE_LEVEL),
                D3D11_SDK_VERSION,           // SDK version
                deviceOut,                   // Device out
                &featureLevelOut,            // Actual feature level created
                contextOut);                 // Context out

        if (SUCCEEDED(hr))
        {
            // A hardware accelerated device has been created, so check for Compute Shader support

            // If we have a device >= D3D_FEATURE_LEVEL_11_0 created, full CS5.0 support is guaranteed, no need for further checks
            if (featureLevelOut < D3D_FEATURE_LEVEL_11_0)
            {
                // Otherwise, we need further check whether this device support CS4.x (Compute on 10)
                D3D11_FEATURE_DATA_D3D10_X_HARDWARE_OPTIONS hwopts;
                (*deviceOut)->CheckFeatureSupport(D3D11_FEATURE_D3D10_X_HARDWARE_OPTIONS, &hwopts, sizeof(hwopts));
                if (!hwopts.ComputeShaders_Plus_RawAndStructuredBuffers_Via_Shader_4_x)
                {
                    bNeedRefDevice = true;
                    printf("No hardware Compute Shader capable device found, trying to create ref device.\n");
                }
            }
        }
    }

    if (bForceRef || FAILED(hr) || bNeedRefDevice)
    {
        // Either because of failure on creating a hardware device or hardware lacking CS capability, we create a ref device here
        SAFE_RELEASE(*deviceOut);
        SAFE_RELEASE(*contextOut);

        hr = D3D11CreateDevice(nullptr,                 // Use default graphics card
            D3D_DRIVER_TYPE_REFERENCE,                  // Try to create a hardware accelerated device
            nullptr,                                    // Do not use external software rasterizer module
            flags,                                      // Device creation flags
            flvl,
            sizeof(flvl) / sizeof(D3D_FEATURE_LEVEL),
            D3D11_SDK_VERSION,                    // SDK version
            deviceOut,                            // Device out
            &featureLevelOut,                     // Actual feature
                level created
            contextOut);                          // Context out
        if (FAILED(hr))
        {
            printf("Reference rasterizer device create failure\n");
            return hr;
        }
    }

    return hr;
}

HRESULT CreateComputeShader(LPCWSTR pSrcFile, LPCSTR pFunctionName,
    ID3D11Device* pDevice, ID3D11ComputeShader** ppShaderOut)
{
    HRESULT hr = S_OK;

    DWORD dwShaderFlags = D3DCOMPILE_ENABLE_STRICTNESS;

    const D3D_SHADER_MACRO defines[] =
    {
        "USE_STRUCTURED_BUFFERS", "1",
        nullptr, nullptr
    };


    // We generally prefer to use the higher CS shader profile when
        possible as CS 5.0 is better performance on 11-class hardware
    LPCSTR pProfile = (pDevice->GetFeatureLevel() >=
        D3D_FEATURE_LEVEL_11_0) ? "cs_5_0" : "cs_4_0";


    ID3DBlob* pErrorBlob = nullptr;
    ID3DBlob* pBlob = nullptr;

#if D3D_COMPILER_VERSION >= 46
    hr = D3DCompileFromFile(pSrcFile, defines,
        D3D_COMPILE_STANDARD_FILE_INCLUDE, pFunctionName, pProfile,
        dwShaderFlags, 0, &pBlob, &pErrorBlob);
```

```
#else
    hr = D3DX11CompileFromFile(pSrcFile, defines, nullptr, pFunctionName, pProfile, dwShaderFlags, 0, nullptr, &pBlob, &pErrorBlob, nullptr);
#endif

    if (FAILED(hr))
    {
        if (pErrorBlob){
            std::cout << (char*)pErrorBlob->GetBufferPointer() << std::endl;
        }


        SAFE_RELEASE(pErrorBlob);
        SAFE_RELEASE(pBlob);

        return hr;
    }



    hr = pDevice->CreateComputeShader(pBlob->GetBufferPointer(), pBlob->GetBufferSize(), nullptr, ppShaderOut);

    SAFE_RELEASE(pErrorBlob);
    SAFE_RELEASE(pBlob);


    return hr;
}

HRESULT CreateRawBuffer(ID3D11Device* pDevice, UINT uElementSize, UINT uCount, void* pInitData, ID3D11Buffer** ppBufOut){
    *ppBufOut = nullptr;

    D3D11_BUFFER_DESC desc;
    ZeroMemory(&desc, sizeof(desc));

    desc.BindFlags = D3D11_BIND_UNORDERED_ACCESS | D3D11_BIND_SHADER_RESOURCE;
    desc.ByteWidth = uElementSize * uCount;
    desc.MiscFlags = D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS;
    desc.StructureByteStride = uElementSize;

    if (pInitData)
    {
        D3D11_SUBRESOURCE_DATA InitData;
        InitData.pSysMem = pInitData;
        return pDevice->CreateBuffer(&desc, &InitData, ppBufOut);
    }
    else
        return pDevice->CreateBuffer(&desc, nullptr, ppBufOut);
}

HRESULT CreateBufferUAV(ID3D11Device* pDevice, ID3D11Buffer* pBuffer,
    ID3D11UnorderedAccessView** ppUAVOut)
{
    D3D11_BUFFER_DESC descBuf;
    ZeroMemory(&descBuf, sizeof(descBuf));
    pBuffer->GetDesc(&descBuf);

    D3D11_UNORDERED_ACCESS_VIEW_DESC desc;
    ZeroMemory(&desc, sizeof(desc));
    desc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
    desc.Buffer.FirstElement = 0;

    if (descBuf.MiscFlags & D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS)
    {
        // This is a Raw Buffer

        desc.Format = DXGI_FORMAT_R32_TYPELESS; // Format must be
            DXGI_FORMAT_R32_TYPELESS, when creating Raw Unordered Access
            View
        desc.Buffer.Flags = D3D11_BUFFER_UAV_FLAG_RAW;
        desc.Buffer.NumElements = descBuf.ByteWidth / 4;
    }
    else
        if (descBuf.MiscFlags & D3D11_RESOURCE_MISC_BUFFER_STRUCTURED)
        {
        // This is a Structured Buffer

        desc.Format = DXGI_FORMAT_UNKNOWN;      // Format must be must
            be DXGI_FORMAT_UNKNOWN, when creating a View of a Structured
            Buffer
        desc.Buffer.NumElements = descBuf.ByteWidth / descBuf.
            StructureByteStride;
        }
        else
        {
            return E_INVALIDARG;
        }

    return pDevice->CreateUnorderedAccessView(pBuffer, &desc, ppUAVOut);
}

ID3D11Buffer* CreateAndCopyToDebugBuf(ID3D11Device* pDevice,
   ID3D11DeviceContext* pd3dImmediateContext, ID3D11Buffer* pBuffer)
{
    ID3D11Buffer* debugbuf = nullptr;

    D3D11_BUFFER_DESC desc;
    ZeroMemory(&desc, sizeof(desc));
    pBuffer->GetDesc(&desc);
    desc.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
    desc.Usage = D3D11_USAGE_STAGING;
    desc.BindFlags = 0;
    desc.MiscFlags = 0;



    if (SUCCEEDED(pDevice->CreateBuffer(&desc, nullptr, &debugbuf)))
    {
        pd3dImmediateContext->CopyResource(debugbuf, pBuffer);
    }


    return debugbuf;
}

int main()
{
    ID3D11Device* device = nullptr;
    ID3D11DeviceContext* context = nullptr;
    ID3D11ComputeShader* CS = nullptr;

    if (FAILED(CreateComputeDevice(&device, &context, false))) exit(1);
    if (FAILED(CreateComputeShader(L"../VectorAddCS.hlsl", "CSMain",
        device, &CS))) exit(1);

    HRESULT hr = S_OK;

    // Buffers
    ID3D11Buffer* b_inA = nullptr;
    ID3D11Buffer* b_inB = nullptr;
    ID3D11Buffer* b_out = nullptr;

    // Access views
    ID3D11UnorderedAccessView*  b_inA_UAV  = nullptr;
    ID3D11UnorderedAccessView*  b_inB_UAV  = nullptr;
    ID3D11UnorderedAccessView*  b_out_UAV = nullptr;

    // Setup data
    int *i_inA = new int[NUM_ELEMENTS];
    int *i_inB = new int[NUM_ELEMENTS];
    int *i_out = new int[NUM_ELEMENTS];

    for (int i = 0; i < NUM_ELEMENTS; i++){
        i_inA[i] = i;
        i_inB[i] = 2 * i;
    }

    // Create buffers
    CHECK_ERR(CreateRawBuffer(device, sizeof(int), NUM_ELEMENTS, &i_inA
        [0], &b_inA));
    CHECK_ERR(CreateRawBuffer(device, sizeof(int), NUM_ELEMENTS, &i_inB
        [0], &b_inB));
    CHECK_ERR(CreateRawBuffer(device, sizeof(int), NUM_ELEMENTS, nullptr
        ,  &b_out));


    // Create access views
    CHECK_ERR(CreateBufferUAV(device, b_inA, &b_inA_UAV));
    CHECK_ERR(CreateBufferUAV(device, b_inB, &b_inB_UAV));
    CHECK_ERR(CreateBufferUAV(device, b_out, &b_out_UAV));

    // Launch CS
    {
        context->CSSetShader(CS, nullptr, 0);
        ID3D11UnorderedAccessView* aRViews[3] = { b_inA_UAV, b_inB_UAV,
            b_out_UAV };
        context->CSSetUnorderedAccessViews(0, 3, aRViews, nullptr);
        context->Dispatch(NUM_ELEMENTS, 1, 1);

        // Unmap resources
        ID3D11UnorderedAccessView* aRViewsNullptr[3] = { nullptr,
            nullptr, nullptr };
        context->CSSetUnorderedAccessViews(0, 3, aRViewsNullptr, nullptr
            );
    }

    // Retrieve results
    {
        // Retrieve positions
        ID3D11Buffer* resDebugbuf = CreateAndCopyToDebugBuf(device,
            context, b_out);
        D3D11_MAPPED_SUBRESOURCE mappedRes;

        context->Map(resDebugbuf, 0, D3D11_MAP_READ, 0, &mappedRes);
        memcpy(i_out, (int*)mappedRes.pData, NUM_ELEMENTS*sizeof(int));
        context->Unmap(resDebugbuf, 0);

    }

    for (size_t i = 0; i < NUM_ELEMENTS; i++) {
        if (i_inA[i] + i_inB[i] != i_out[i]) return 1;
    }


    std::cout << "Sucess!" << std::endl;

    return 0;
}
```