Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Comparison of different n-body algorithms on various hardware platforms using SYCL

Tim Thüring

Course of Study:	Informatik
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Marcel Breyer, M.Sc.
Commenced:	October 5, 2022
Completed:	April 5, 2023

Abstract

The n-body problem has various applications in different fields of science such as astrophysics, where it describes the problem of calculating the movements of n different bodies which all interact with each other over time. There exist different algorithms that solve the n-body problem, for example, the naive approach and the Barnes-Hut algorithm. Since applications of the n-body problem can deal with large systems of bodies it is crucial to understand the runtime behavior of theses algorithms.

In this thesis the runtime behavior of the naive algorithm and the Barnes-Hut algorithm will be compared on different CPUs as well as on GPUs from different vendors. Both algorithms will be implemented using SYCL which is an abstraction layer that enables parallel programming for various types of devices. With SYCL there is no need to use different languages for parallel programming on CPUs and GPUs and the whole code can be written using standard C++.

The results show that one can achieve better performance with both algorithms on GPUs than on CPUs. A projection for the runtime of a simulation of approximately one earth year with a system of over 1.2 million bodies and a Δt of one hour shows that an NVIDIA A100 GPU could finish this simulation in under one day with the naive approach. A dual socket AMD EPYC 7543 would take almost twelve days for the same simulation. This shows that the naive algorithm maps extremely well to GPUs. CPUs can keep up better with the Barnes-Hut algorithm. Here the projection of the runtime showed that the NVIDIA A100 would finish the simulation of an earth year in just below an hour whereas the dual socket AMD EPYC 7543 could finish the simulation in 2.59 hours.

Kurzfassung

Das n-Körper-Problem hat viele Anwendungen in unterschiedlichen Forschungsgebieten, wie zum Beispiel Astrophysik, in denen es das Problem der Berechnung von Bewegungen unterschiedlicher Körper beschreibt, die alle miteinander interagieren. Es existieren verschiedene Algorithmen, die das n-Körperproblem lösen, zum Beispiel der naive Ansatz oder der Barnes-Hut Algorithmus. Da sich Anwendungen des n-Körper-Problems mit großen Systemen von Körpern beschäftigen können, ist es wichtig das Laufzeitverhalten dieser Algorithmen zu verstehen.

In dieser Arbeit wird das Laufzeitverhalten des naiven Algorithmus und des Barnes-Hut Algorithmus auf unterschiedlichen CPUs und GPUs von verschiedenen Herstellern verglichen. Beide Algorithmen werden unter der Verwendung von SYCL implementiert, was eine Abstraktionsschicht ist die es ermöglicht verschiedenste Hardware parallel zu programmieren. Mit SYCL besteht nicht die Notwendigkeit verschiedene Sprachen für das parallele Programmieren auf CPUs und GPUs zu verwenden und der gesamte Programmcode kann in Standard C++ geschrieben werden.

Die Ergebnisse zeigen, dass man bei beiden Algorithmen bessere Performanz auf GPUs erreichen kann als auf CPUs. Eine Hochrechnung der Laufzeit für eine Simulation von ungefähr einem Erd-Jahr mit einem System von 1,2 Millionen Körpern und einem Δt von einer Stunde zeigt, dass eine NVIDIA A100 GPU mit dem naiven Ansatz die Simulation in unter einem Tag beenden kann. Ein Dual-Socket AMD EPYC 7543 würde fast zwölf Tage für diese Simulation benötigen. Das zeigt, dass sich der naive Algorithmus enorm gut auf GPUs anwenden lässt. CPUs können beim Barnes-Hut Algorithmus besser mithalten. Hier zeigt die Hochrechnung, dass die NVIDIA A100 die Simulation eines Jahres in knapp unter einer Stunde schaffen würde, während der Dual-Socket AMD EPYC 7543 die Simulation in 2,59 Stunden schaffen könnte.

Contents

List of Figures

Acronyms

AU astronomical unit. 43

CPU central processing unit. 11

FPGA field programmable gate array. 19

GPU graphics processing unit. 11

HT hyper-threading. 33

TFLOPS tera floating point operations per second. 34

1 Introduction

The n-body problem occurs in different fields of science and describes the calculation of the movements of n different bodies in a system where all of these bodies interact with each other. One of the main applications of this problem can be found in astrophysics where the goal is to calculate the trajectories of different celestial objects like planets or asteroids which all interact with each other through gravitational forces. There are different approaches that solve the n-body problem. The naive approach considers all pairwise interactions between all bodies in the system. This leads to the most precise result, however, the runtime of the algorithm grows quadratically with the number of bodies in the system which makes the approach poorly suited for the simulation of very large systems. Therefore, other algorithms have been developed that apply approximations to the simulation in order to speed up the calculation. One such algorithm is the Barnes-Hut algorithm [BH86], which leverages an octree data structure to apply approximations and improve the performance of n-body simulations.

Since n-body simulations can be performed with large systems of bodies it is important to understand the runtime behavior of these algorithms. In this thesis the runtime behavior of two algorithms for the n-body problem, the naive algorithm and the Barnes-Hut algorithm, will be compared on different central processing units (CPUs) as well as on graphics processing units (GPUs) from NVIDIA and AMD. Parallel programming on such different hardware platforms usually involves the usage of different languages such as CUDA, HIP or OpenMP. Most of the works that implemented n-body algorithms for GPUs use CUDA. In this thesis both algorithms will be implemented with the usage of SYCL which is an abstraction layer that enables parallel programming on a variety of devices. The benefit of using SYCL is that one does not need to use different languages for parallel programming on a specific device. Instead, with SYCL one can write a whole application in standard C++ that is then capable of addressing a variety of devices.

The results of the comparison conducted during this thesis show that one can achieve better performance on GPUs than on CPUs for both algorithms. The differences between those two types of devices is especially significant for the naive algorithm where GPUs have a clear advantage over CPUs.

A projection for the runtime of a simulation of approximately one earth year with a system of over 1.2 million bodies and a Δt of one hour shows that when using the naive approach an NVIDIA A100 GPU could finish the simulation in under one day. A dual socket AMD EPYC 7543 would need almost twelve days in the same situation. However, CPUs can keep up better with GPUs when using the Barnes-Hut algorithm. Nevertheless, also with this algorithm GPUs are faster in the end. Here the projection of the runtime showed that with the Barnes-Hut algorithm the NVIDIA A100 would finish the simulation of an earth year in just below an hour whereas the dual socket AMD EPYC 7543 could finish the same simulation in 2.59 hours.

This thesis is structured as follows: In the next chapter important work related to this thesis will be presented. Chapter 3 will introduce all important fundamentals about n-body problems and SYCL. In chapter 4 the implementations of the naive algorithm and the Barnes-Hut algorithm conducted during this thesis will be described. The results of the comparison of both algorithms on different GPUs and CPUs will be presented in chapter 5. Finally, chapter 6 will conclude this thesis and discuss possible directions of future work.

2 Related work

This section will present scientific work related to the topic of this thesis. There exist a lot of implementations of n-body algorithms on CPUs as well as on GPUs. An implementation of the naive algorithm on GPUs was performed by Nyland et al. [NHP07]. The authors parallelized the naive approach on NVIDIA GPUs with the usage of CUDA. Arora et al. [ASV09] implemented the naive approach for several CPUs and GPUs and compared the performance on these different platforms. Their GPU implementation makes use of CUDA and uses the approach by Nyland et al. However, they only considered the naive algorithm. Furthermore, the hardware used for the comparison (NVIDIA Tesla C1060 and NVIDIA Tesla C870 GPUs as well as, for example, Intel Nehalem and AMD Barcelona CPUs) is not up to date anymore. Capuzzo-Dolcetta et al. [CS13] implemented the naive algorithm with OpenCL an compared their implementation on GPUs from different manufactures.

A CUDA implementation of the Barnes-Hut algorithm on NVIDIA GPUs was conducted by Burtscher et al. [BP11]. The octree creation phase of the Barnes-Hut algorithm was implemented using a parallel top-down approach that makes use of synchronization. The authors also compared their implementation of the Barnes-Hut algorithm with a CUDA implementation of the naive algorithm. A comparison of the performance of the Barnes-Hut algorithm on CPUs and GPUs was also performed by the authors. However, the CPU implementation of the Barnes-Hut algorithm used for this comparison was not parallelized.

There also exist other approaches for the octree creation like the one proposed by Warren et al. [WS93] which makes use of a hashed octree and leverages space filling curves for decomposition of all bodies in the space. It is also possible to construct the tree bottom-up. This technique has been, for example, applied by Alpay [Alp19b] to an algorithm that is similar to the Barnes-Hut algorithm with the usage of OpenCL. For the tree the SpatialCL library by Alpay [Alp19a] is used that also includes a demonstration of the Barnes-Hut algorithm.

Beyond the Barnes-Hut algorithm that has a theoretical complexity of $O(n\ log(n))$ there also exist other approaches that try to reduce the theoretical complexity of the naive algorithm. One example is the fast multipole method by Greengard et al. [Gre87] which reduces the theoretical complexity to $O(n)$. Yokota et al. [YB11] implemented both the fast multipole method and the Barnes-Hut algorithm with CUDA for execution on GPUs. They also compared both algorithms with the naive approach on GPUs and CPUs.

Overall, not many comparisons of the Barnes-Hut algorithm and the naive algorithm have been conducted on different modern hardware. Most comparisons so far have used different implementations of the algorithms with different programming languages for the various devices. The implementations of the naive algorithm and the Barnes-Hut algorithm presented in this work will only use SYCL code. This allows for using mostly the same code for the execution on CPUs and GPUs. Different frameworks than SYCL exist and also have been applied to n-body simulations like the Cabana library [SRJ+22] that uses Kokkos [TLA+22] for achieving performance portability.

However to my knowledge no comparisons of different algorithms with the usage of this library have been conducted. Furthermore, I am not aware of any similar work that makes use of SYCL for comparison between the naive algorithm and the Barnes-Hut algorithm on different hardware.

3 Foundations

In this chapter the basic concepts and tools which are relevant for this thesis will be introduced. Starting with physical background about n-body problems and the ideas behind two different algorithms for n-body simulations. Last, SYCL will be introduced as part of the technical background of this thesis.

3.1 The n-body problem

The n-body problem describes the calculation of pairwise interactions between n different bodies to obtain their movements over time. It has applications in various fields of science such as molecular dynamics [EH86], plasma physics [GSK+10], fluid dynamics [SW94] or astrophysics [BH86]. The variants of the n-body problem in those different fields mainly differ in the kind of force which underlies the interactions in the system.

This work considers the application of the n-body problem for astrophysics, where the main application is the simulation of celestial objects, e.g., the planets, moons and asteroids of our solar system. This variant of the n-body problem is also called gravitational n-body problem since the different bodies of the system interact with each other through gravitational forces. It is described in various literature, e.g., in the work by Meyer et al. [Ken17] on which the content of the following paragraphs is based.

In the gravitational case the interactions between the bodies arise through gravity. The individual bodies in the system are modeled as point masses where every body i is described by its mass m_i and a position $\vec{r}_i$. With this, the calculation of the gravitational force acting on body i can be derived from Newton's second law of motion and Newton's law of gravity. Together this gives the following equation for the total force $\vec{F}_i$ acting on body i generated through the gravity of all other bodies j:

$$\vec{F}_i = m_i \cdot \vec{a}_i = G \cdot m_i \cdot \sum_{j \neq i} \frac{m_j \cdot (\vec{r}_j - \vec{r}_i)}{\|\vec{r}_j - \vec{r}_i\|^3} \quad . \tag{3.1}$$

In equation 3.1, $G \approx 6.674 \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$ denotes the gravitational constant and $\vec{a}_i$ describes the acceleration of body i. Often the acceleration value $\vec{a}_i$ is sufficient, which simplifies equation 3.1 to equation 3.2:

$$\vec{a}_i = G \cdot \sum_{j \neq i} \frac{m_j \cdot (\vec{r}_j - \vec{r}_i)}{\|\vec{r}_j - \vec{r}_i\|^3} \quad . \tag{3.2}$$

In order to obtain the evolution of the system over time, i.e., the movements of the bodies in the system, one uses numerical integration over time. One way of getting the new positions $\vec{r}_{i+1}$ and velocities $\vec{v}_{i+1}$ of each body in the next time step $i + 1$, is the so called leapfrog integration method.

The method is described in various literature. This paragraph will be based on the work by Hairer et al. [HLW03]. The leapfrog integration method is sometimes also named Verlet integration and can be written and interpreted in a variety of ways. A possible interpretation of this scheme is that it is a composition of two symplectic Euler schemes. One variant of the leapfrog integration scheme can be expressed as shown below:

$$\vec{v}_{i+\frac{1}{2}} = \vec{v}_i + \vec{a}_i \cdot \frac{\Delta t}{2} \tag{3.3}$$

$$\vec{r}_{i+1} = \vec{r}_i + \vec{v}_{i+\frac{1}{2}} \cdot \Delta t \tag{3.4}$$

$$\vec{v}_{i+1} = \vec{v}_{i+\frac{1}{2}} + \vec{a}_{i+1} \cdot \frac{\Delta t}{2} \quad . \tag{3.5}$$

After the step described by equation 3.4 the new acceleration values $\vec{a}_{i+1}$ have to be determined. In the case of the gravitational n-body problem equation 3.2 could be used. However, as described by e.g. Nyland et al. [NHP07], for n-body simulations in practice this would not be applicable. Since the individual bodies are modeled as point masses and collisions are not taken into account, the forces between two bodies that get close to each other grow towards infinity. This can be circumvented by adding a softening factor ϵ^2 and rewriting equation 3.2 to equation 3.6 for the acceleration computation:

$$\vec{a}_i \approx G \cdot \sum_{j \neq i} \frac{m_j \cdot (\vec{r}_j - \vec{r}_i)}{\left(\|\vec{r}_j - \vec{r}_i\|^2 + \epsilon^2\right)^{\frac{3}{2}}} \quad . \tag{3.6}$$

Together, the acceleration computation and time integration form the basic building blocks of n-body simulations.

3.2 Algorithms for the N-Body Problem

The part dominating the runtime of n-body simulations is the acceleration computation which has to be done in every time step. Beyond the naive approach for the acceleration calculation, several other approaches have been developed to speed up the acceleration computations. These approaches include methods such as the Barnes-Hut algorithm [BH86], the fast multipole method [Gre87], or particle-mesh methods (see, e.g., Bertschinger et al. [BG91]). This work is concerned with two different algorithms for n-body simulations: the naive algorithm and the Barnes-Hut algorithm. In this section, these two algorithms for the acceleration computation of the n-body problem will be introduced.

3.2.1 The Naive Approach

The naive approach for the acceleration computation is described in various literature, e.g., by Nyland et al. [NHP07]. In order to obtain the movements of all bodies over time, one has to calculate the new acceleration vector of each body in every time step. This means that for a system of n bodies, n acceleration values have to be determined. Using the naive approach these values can be calculated by applying equation 3.6 for all n bodies i. Since each of the n acceleration computations requires to sum over the accelerations induced by the gravity of all the other $n - 1$ bodies j, naively computing the accelerations results into $O(n^2)$ complexity.

For large systems, this quadratic complexity can be problematic since the runtime of the naive algorithm can get quite long. Thus, other algorithms have been developed to speed up the calculation of the acceleration values. However, such algorithms often apply approximations and thus do not reach the accuracy of the naive approach. One such algorithm, the Barnes-Hut algorithm, will now be introduced in the next section.

3.2.2 The Barnes-Hut Algorithm

The Barnes-Hut algorithm was proposed by Josh Barnes and Piet Hut [BH86]. It is a tree based method that reduces the complexity of the acceleration computation to $O(n\,log(n))$ which makes it much more applicable for large systems compared to the naive approach.

The Barnes-Hut algorithm can compute the n different acceleration values $\vec{a}_i$ more efficiently, since it applies approximations to each of the n calculations. In contrast to the naive algorithm, which considers all interaction pairs for the calculation of one acceleration $\vec{a}_i$, the Barnes-Hut algorithm reduces the number of interaction calculations per acceleration value. The general idea of the approximation is that for the calculation of acceleration $\vec{a}_i$ of body i, the acceleration is only directly computed for the bodies j that are close to body i. Groups of bodies that are far away are grouped together and are estimated as one big virtual body. When the bodies are far away the loss of precision of this approximation compared to individually computing all accelerations is not large.

In order to determine when to apply this approximation, all bodies are stored in a tree data structure according to their position. In the three dimensional case, such a data structure would be an octree, in two dimensions, a quadtree. The implementation of the algorithm performed during this thesis always considers the three dimensional case. For easier visualization, all following graphics will be limited to the two dimensional case with quadtrees. However, all concepts described here can be easily extended to three dimensions.

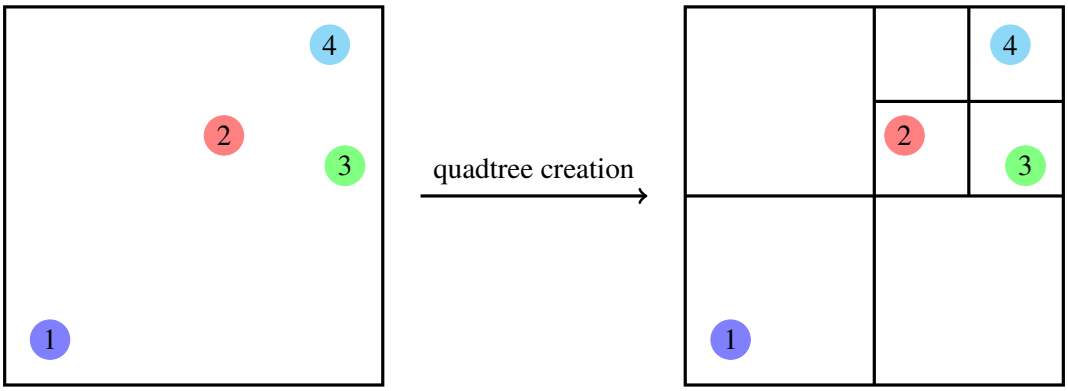


Figure 3.1: Example of a subdivision of a space containing four bodies numbered from 1 to 4 into a quadtree. In the situation shown on the left there is only one cell that contains all four bodies. This cell then gets recursively subdivided into smaller cells until there is only one body in each cell.

Figure 3.1 shows how a space containing 4 bodies numbered from 1 to 4. The space gets recursively subdivided into smaller cells, until there is only one body in each cell. In the corresponding quadtree data structure, each cell c_i corresponds to a node of the tree. The quadtree corresponding to figure 3.1 is visualized in figure 3.2.

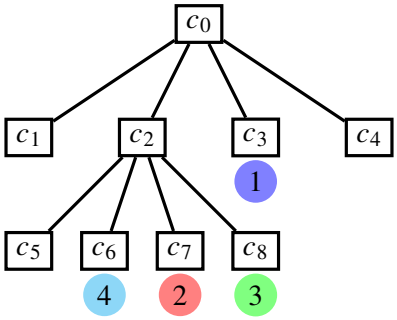


Figure 3.2: Example for a quadtree data structure for the quadree visualized in figure 3.1. Each node of the tree corresponds to one cell. Below the nodes that contain a body, the respective body is shown.

All nodes, have to store the sum of the masses of all bodies which are contained in any of the nodes further down the tree and the center of mass of all those bodies. The center of mass of several bodies k can be calculated using equation 3.7. With that, each non-leaf node of the tree also represents a virtual body located in the center of mass of all the bodies in the cell which is represented by this node, and with a mass of the sum of all masses of those bodies.

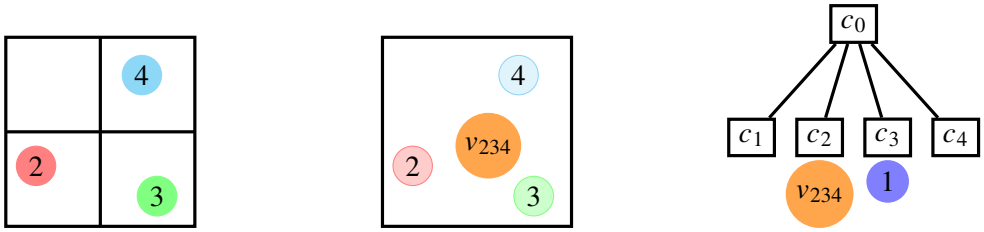
$$\vec{C} = \frac{\sum_k m_k \vec{r}_k}{\sum_k m_k} \tag{3.7}$$

For example, cell c_2 in figure 3.2 got subdivided into more cells since it contains a total of 3 bodies: body 2, 3 and 4. Thus, as shown in figure 3.3, the node corresponding to cell c_2 represents the virtual body v_{234} located at the center of mass of the bodies 2, 3 and 4 and with a mass corresponding to the sum of the individual masses of the three bodies.

During the acceleration computation, for each body i, the tree is traversed top to bottom. For each node, one has to decide if an approximation should be used, i.e., if only one acceleration induced by the virtual body represented by the current node should be used. In such a case, all nodes and bodies in the subtree of this node will not be considered anymore for the acceleration computation. This can drastically reduce the number of acceleration computations. In the example above this would correspond to the following situation: assume that the goal is the computation of acceleration $\vec{a}_1$. When the algorithm arrives at the node representing cell c_2 it has to make the decision if the virtual body v_{234} should be used or if an approximation is not feasible here and one should continue to step down deeper into the tree, computing more precise accelerations with bodies 2, 3 and 4 individually.

The decision, if the virtual body v represented through the current node under consideration in the tree traversal should be used as an approximation for all other bodies contained in the cell c represented by this node is based on the following criterion:

$$\frac{edgeLength(c)}{||\vec{r}_v - \vec{r}_i||} < \theta \quad . \tag{3.8}$$



(a) Cell c_2

(b) Virtual body v_{234} represented by cell c_2

(c) Quadtree with the virtual body v_{234} instead of the subtree containing the individual bodies 2, 3 and 4

Figure 3.3: Example of how nodes of a quadtree represent virtual bodies. Figure 3.3a shows cell c_2, divided into four quadrants and containing the bodies with ID 2, 3 and 4. Figure 3.3b shows cell c_2 with the virtual body v_{234} located in the center of mass of all bodies contained in cell c_2. Figure 3.3c shows the quadtree data structure where the virtual body v_{234} represents the whole subtree containing the bodies 2, 3 and 4.

Here, $\vec{r}_i$ is the position vector of the body i whose acceleration $\vec{a}_i$ should be computed, whereas $\vec{r}_v$ corresponds to the position of the virtual body, which is the center of mass of all bodies in the cell. θ is a constant that defines the accuracy of the algorithm. For higher values of θ, approximations with virtual bodies will be applied more often, which leads to better performance but lower accuracy. Lower θ-values give higher accuracy, but it comes at the cost of more computations and thus longer run-times. For $\theta = 0$ no approximations are applied and the algorithm would correspond to the naive approach again.

The tree creation step of the Barnes-Hut algorithm requires $O(n\ log(n))$ calculation steps. Furthermore, the time complexity of the computation for the calculation of one acceleration $\vec{a}_i$ is $O(log(n))$. Since in total n acceleration values have to be computed per time step, the total complexity of the acceleration computation step is $O(n\ log(n))$. With that, the total complexity of the Barnes-Hut algorithm reaches $O(n\ log(n))$, which is a significant improvement over the naive approach.

3.3 SYCL for parallel programming on various hardware

SYCL 2020 [Khr] is a standard that acts as an abstraction layer which enables parallel programming for heterogeneous systems and is developed by the Khronos Group[1]. SYCL code can be written using only standard ISO C++. The advantage of an application written with SYCL code is that it can be executed in parallel on a variety of devices, such as CPUs and GPUs from different manufacturers or even field programmable gate arrays (FPGAs).

In this thesis, SYCL will be used to compare the runtime behavior of different n-body algorithms on different GPUs and CPUs. For creating a SYCL application and running it on different devices one needs an implementation of SYCL. There are a variety of SYCL implementations, each of which include support for different backends, that are then used for executing code in parallel on

[1] https://www.khronos.org (visited on 04/03/2023)

the selected devices. For example, when executing SYCL code on an NVIDIA GPU, the CUDA backend could be used in the background, whereas code executing on a CPU could be parallelized using OpenMP as a backend.

This work uses two different SYCL implementations: Open SYCL[2] [AH20] and DPC++[3]. Open SYCL offers support for NVIDIA and AMD GPUs with the CUDA and the ROCm backend respectively. Furthermore Intel GPUs are supported with Level Zero. Open SYCL also offers broad CPU support through the OpenMP backend.

DPC++ supports Intel CPUs, GPUs and FPGAs with the SPIR-V backend and the OpenCL backend. Intel GPUs are also supported with Level Zero. DPC++ also offers support for NVIDIA GPUs through CUDA and AMD GPUs through HIP. However, the AMD HIP backend is currently in experimental state.

There also exist other SYCL implementations like, for example, Codeplay ComputeCpp[4]. However, these implementations will not be considered in this work.

The SYCL application written in this work utilizes the CUDA backends of Open SYCL and DPC++ for execution on NVIDIA GPUs and the ROCm backend of Open SYCL for AMD GPU support. For parallel execution on CPUs the OpenMP backend of Open SYCL is used.

In the next sections a short introduction into programming with SYCL will be given which explains the high level concepts of how to parallelize code with SYCL and execute it on different devices.

3.3.1 Programming with SYCL

The content of this section follows the explanations of the book Data Parallel C++ by Reinders et al. [RAB+21].

On the highest level, a SYCL application can be divided into host and device code. Device code is most of the time the computational intensive part of the program which should get accelerated by parallel execution on a device. Such a device could be a GPU but also a CPU. Host code on the other hand is all the other code of the application, besides the device code. This code gets executed on the CPU of the system.

Listing 3.1 shows a simple example of SYCL code. The code initializes a vector with some data on the CPU. After that each entry of the vector is squared. This operation will happen in parallel on a device of the system. The following paragraph will describe the example step by step.

In line 1, a queue is created. A queue is associated with one device and used to submit code which should be executed on the device. In this case no device is specified, so the SYCL implementation will select a device. However, here it would be possible to specify that, for example, this queue should be associated with a GPU.

[2] https://github.com/OpenSYCL/OpenSYCL (visited on 04/03/2023)
[3] https://github.com/intel/llvm (visited on 04/03/2023)
[4] https://developer.codeplay.com/products/computecpp/ce/home/ (visited on 04/03/2023)

Listing 3.1 Example of how code gets submitted to a device for parallel execution when using SYCL

```
1  sycl::queue queue;
2
3  std::vector<int> storage = {2, 9, 10, 7, 4, 3, 1, 5};
4  sycl::buffer<int> data_buffer(storage.data(), storage.size());
5
6  int range = storage.size();
7
8  queue.submit([&](sycl::handler &h) {
9
10         sycl::accessor<int> data_accessor(data_buffer, h);
11
12         h.parallel_for(sycl::range<1>(range), [=](auto &i) {
13                 data_accessor[i] = data_accessor[i] * data_accessor[i];
14         });
15 }).wait();
```

The code in line 4 creates a buffer for variables of type int. A buffer is an abstraction layer for data in SYCL. This data can be accessed by the device using an accessor. In the example an accessor is created in line 10.

In line 8 a kernel containing device code is submitted to the device with which the previously created queue is associated. The only device code which gets executed in parallel is line 13. Here the previously created accessor is used to modify the data stored in the buffer. By default, device code submitted to a queue gets executed asynchronous to the host code. The code in line 10 however, gets executed directly and not asynchronously. In this example the call to `wait()` in line 15 causes to the host device to wait until the device executing the kernel is finished with the computation.

One instance of the kernel which executes line 13 in parallel to all other instances is called work-item. The amount of parallelism or the number of work-items which are used is controlled in line 12. In this example it corresponds to the size of the buffer. Each of the work-items of the kernel is identified by an ID, here, in this example the ID is stored in the variable i.

In the previous example, besides the amount of work-items no further specifications about the parallel execution are made. However, SYCL provides a more powerful option to express parallelism: nd-range kernels. For this work, the most important property of nd-range kernels is the ability to group different work-items together into work-groups. E.g.: one can divide a total of 1024 work-items into 16 work-groups with 64 work-items each.

The advantage of this approach is that there exist several work-group specific functions that can be used to optimize the application. For example, each work-group has its own local memory shared between all work-items of that work-group. Depending on the device which is used for running the kernel, this type of memory can be much faster than the global memory that can be accessed by all work-items. Furthermore there are several build in synchronization options between work-items that belong to one single work-group like barriers or memory fences.

4 Implementation

In this work, two algorithms for the n-body problem, the naive approach and the Barnes-Hut algorithm, have been implemented and parallelized for execution on different hardware using SYCL. All computationally demanding parts of the algorithms have been parallelized with SYCL for execution on CPUs and GPUs, including the leapfrog integration. In this chapter, the approaches of how the algorithms are implemented for parallel execution will be described. Furthermore, it will be explained how the algorithms are optimized for better performance.

4.1 Implementation of the naive algorithm

The implementation of the naive algorithm with SYCL conducted in this thesis is based on the CUDA implementation of this algorithm by Nyland et al. [NHP07].

As described in section 3.2.1, calculating all the n accelerations of a system with n bodies requires $O(n^2)$ acceleration computations. Each of those $O(n^2)$ accelerations can be computed independently of each other which makes this approach embarrassingly parallel. After that all accelerations corresponding to one body can be summed up. This means that in theory one could have a maximum of $O(n^2)$ calculations executing in parallel. The disadvantage of this method is that it also requires quadratic amount of memory, since all acceleration values have to be stored until the final n accelerations can be calculated. Thus, Nyland et al. chose a different approach where only the n acceleration computations for each body happen independent of each other in parallel and the $n-1$ acceleration values which emerge through the interaction of one body with all other bodies are calculated sequentially. This approach still results in more than enough parallelism to fully utilize modern GPU and CPU hardware.

The implementation of Nyland et al. was performed using CUDA and thus only targets NVIDIA GPUs. The authors describe in detail how they optimized their implementation for execution on GPUs. Since the high degree of parallelism available in the naive algorithm is especially well suited for GPUs, the implementation of the naive algorithm done in this work is based on this GPU implementation by Nyland et al. Their optimization approaches for GPUs programmed with CUDA are also realized in SYCL code. The implementation of the algorithm with SYCL enables execution on both GPUs and CPUs. The impact of the optimization on the runtime on CPUs will be analyzed more closely in chapter 5.3.1.

4.1.1 Optimization for GPUs

One optimization approach of the naive algorithm described by Nyland et al. aims to speed up the acceleration computation by using the shared memory of GPUs. In SYCL this corresponds to using local memory which is accessible to all work-items of one work-group. In order to achieve

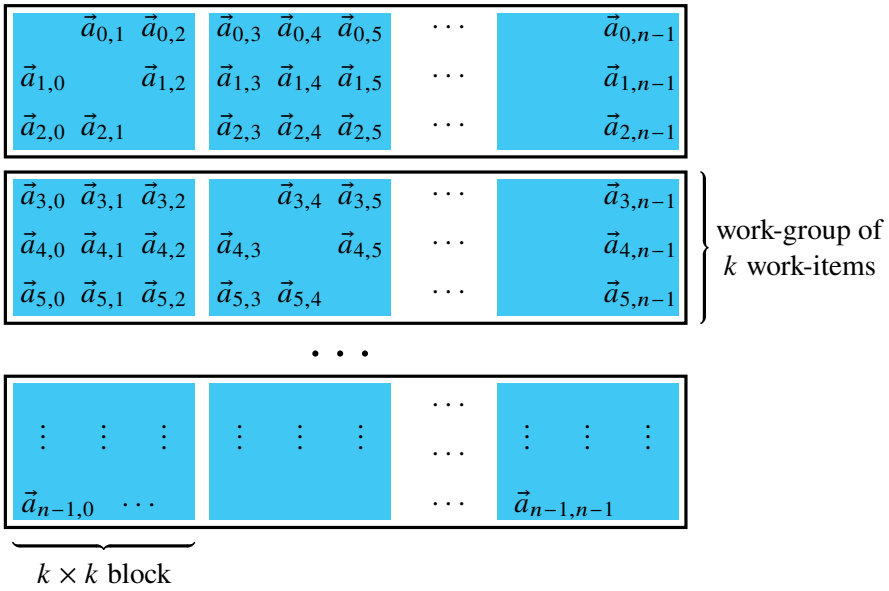


Figure 4.1: Grouping of the acceleration computations into work-groups of size k=3. The computation of the acceleration values in each work-group happens in blocks of size $k \times k$ (highlighted in blue) in order to make use of the work-groups local memory. Modified from Nyland et al. [NHP07] Figure 31-4.

this optimization with SYCL, the first step is to group the n acceleration value computations into work-groups of a fixed size k. Figure 4.1 shows an example of how the individual calculations for all acceleration values $\vec{a}_{i,j}$ are grouped into work-groups for $k = 3$. Here i denotes the index of the body whose acceleration $\vec{a}_i$ is calculated by the i-th work-item. $\vec{a}_{i,j}$ corresponds to an individual accelerations induced by one of the other $n - 1$ bodies j.

The computations of the acceleration values in each work-group happen in blocks. For a work-group consisting of k work-items, the $k \cdot (n - 1)$ acceleration values are calculated in blocks of size $k \times k$. This is necessary for the usage of the local memory. Before each block, each work-item loads the values of one of the k bodies into the local memory. After that all work items that belong to this work-group are synchronized through a barrier. Now, all values needed for the acceleration computation (mass and position) of the k bodies of the current block are accessible to all work-items of the work-group in local memory. Each work-item has to only load the values of one body from the slower global memory into the faster local memory. After the synchronization, each work-item has fast access to the position and mass values of all k bodies of the block and can now compute the k accelerations induced by these bodies to the body associated with the respective work-item.

For example, in the first block of the second work-group in figure 4.1, the work-item with ID 3 would load all values of body 0 into the local memory, making it also accessible with fast access for work-items 4 and 5. Work-item 4 and 5 will do the same for the bodies with ID 1 and 2 respectively. After each work-item has finished the calculation of the acceleration values, the k work-items synchronize before the next position and mass values for the next k bodies are loaded into the local memory.

In the variant described above, the size k of a work-group would have to divide the total amount of bodies, since in SYCL all work-groups must have the same size. Since the number of bodies can be arbitrary, it would limit the possible values for k. However, the size k of the blocks is not irrelevant. For example for GPUs it is best to choose k as a multiple of a power of 2. In order to allow arbitrary values for k, a padding is added to the global range of all work-groups so that it is divisible by k. With this change there exist a few work-items in the last work-group that are not associated with any acceleration computation. It only requires small changes to handle this case and ensure that only work-items associated with an acceleration value $\vec{a}_i$ perform the calculations. However, the work-items that do not perform any calculations of acceleration values are still needed to load values into the local memory.

4.2 Implementation of the Barnes-Hut algorithm

The Barnes-Hut algorithm [BH86] consists of two major parts: the octree creation and the computation of the acceleration values using the octree. The biggest challenge when implementing this algorithm is the parallelization of the octree creation, especially on GPU hardware. An implementation of the Barnes-Hut algorithm on GPUs with a classical octree was conducted by Burtscher et al. [BP11] using CUDA. The SYCL implementation of the Barnes-Hut algorithm on CPUs and GPUs performed in this thesis will be based on the implementation of Burtscher et al.

In their CUDA implementation of the Barnes-Hut algorithm, Burtscher et al. divided the Barnes-Hut algorithm further into several steps. First, the bounding box of all bodies used for the simulation is calculated. After that the octree data structure gets build up, followed by the computation of the center of mass for each node. To minimize thread-divergence, Burtscher et al. introduced a new step before the acceleration computation: an in-order sorting of all bodies in the octree. These concepts are realized in this thesis with the usage of SYCL. The following sections will describe in more detail how the Barnes-Hut algorithm was implemented using SYCL for execution on CPUs and GPUs.

4.2.1 Parallel octree creation

Burtscher et al. use a parallel top-down approach for creating the octree data structure. At the beginning, the octree consists of only one node, the root node, which represents a cell corresponding to the bounding box of all bodies. In the SYCL implementation of this work, every work-item gets assigned a subset of all bodies that it has to insert into the tree. One work-item can insert bodies in parallel to other work-items. When a work-item inserts a body into the octree, it traverses the tree top to bottom starting with the root node. Stepping further down into the tree does not require any synchronization, since no modifications are made. If a leaf node is reached, the insertion point for the current body is found in the tree. Since modifications of the node will happen in the next step, it is necessary to lock the node and prevent other work-items from working on it while the current work-item tries to insert the body. If the leaf node is empty, i.e., does not already contain a different body, the body can be inserted and the work-item can continue with the next body assigned to it. If the leaf node already contains a body, the node has to be split, i.e., it has to become an inner node and eight new child nodes have to be created.

When making such changes, like splitting a node and creating eight new nodes, or inserting a body into a node, one needs to be aware of the following. After releasing the lock that locks the node, it is not guaranteed that other work-items will see the changes made to the node. Thus, a memory fence is needed to ensure that the view of all work-items on the global memory is consistent after modifying the tree.

Memory fences are supported by SYCL, however, different SYCL implementations vary a lot in their implementation and feature support of this function. DPC++ already supports the `atomic_fence` function introduced with SYCL 2020 whereas currently this function is not yet supported by Open SYCL. Open SYCL only supports the `mem_fence` function of the older SYCL 1.2.1 specification [Khr20]. In this specification however, there exist only memory consistency guarantees for work-items in one work-group when using a `mem_fence`. But no consistency guarantees are made for work-items in different work-groups (see [Khr20] chapter 3.5.2.2.). The implementation of `atomic_fence` in DPC++ did not guarantee consistency across different work-items that belong to different work-groups during testing with the CUDA backend. However, these issues would require more research which is beyond the scope of this thesis. Furthermore in the current revision 6 of the SYCL 2020 standard, the SYCL memory model is not completely formalized yet (see [Khr22] chapter 3.8.3.2.).

Due to this current state and the goal that the implementation of the Barnes-Hut algorithm should be compatible with several backends and SYCL implementations, it can not be assumed that there are any consistency guarantees across work-groups when using memory fences with SYCL. Thus, this limits the approach described above to one work-group in SYCL. Due to device specific constraints this also limits the amount of parallelism that is available since work-groups are not allowed to be arbitrarily large. For example for most GPUs tested during this work the limit is 1024 work-items per work-group. Limiting the whole octree creation to one work-group would thus result into a non-optimal usage of the resource of a device, especially for highly parallel GPU hardware. Thus a new approach had to be taken which makes a few changes to the algorithm in order to achieve more parallelism with SYCL. This approach will be explained in the following section.

Leveraging subtrees for more parallelism

The general idea of the approach that will be described in this section is that it builds the octree with the same method as previously, but in two steps. First, the top of the tree gets build up to a certain level that can be specified using a parameter. Figure 4.2 visualizes this step with an example for a maximum level of two. Here, ten bodies with IDs from 1 to 10 are inserted into the upper part of the tree. In this example the tree corresponds to a quadtree for easier visualization. However the concept can be easily extended to an octree. The body with ID 4 got already inserted into a node on level one, thus no subtree originates here. The other bodies have not reached their final position on a level smaller or equal than two. This means that the node on level two into which they would have been inserted becomes the root node of their subtree. For example, in figure 4.2 the bodies with ID 1, 2 and 3 will belong to the same subtree.

Figure 4.3 shows how the subtrees which were determined in the first step are build in a second step. The benefit is that each of these subtrees can now be build independently of the other subtrees in separate work-groups. More importantly, if changes are made to one of the subtrees, it is only important that all work-items of the work-group associated with this subtree see these changes.

Work-items belonging to other work-groups will never access nodes of this subtree, thus it is not important that their view on this part of the memory is consistent. This means that with this approach memory fences which only have effects on one work-group are fully sufficient.

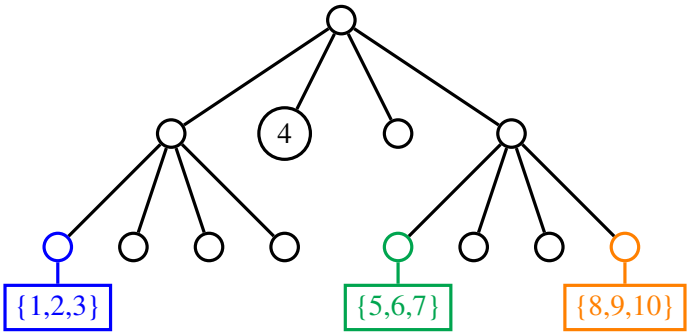


Figure 4.2: Building the tree to a maximum level of two to determine the subtrees for each body. All leaf nodes in this example that contain a set of more than one body (represented through their ID) will become subtree root nodes. Body 4 got already inserted at level one. Thus body 4 has reached its final position and will not be part of any subtree

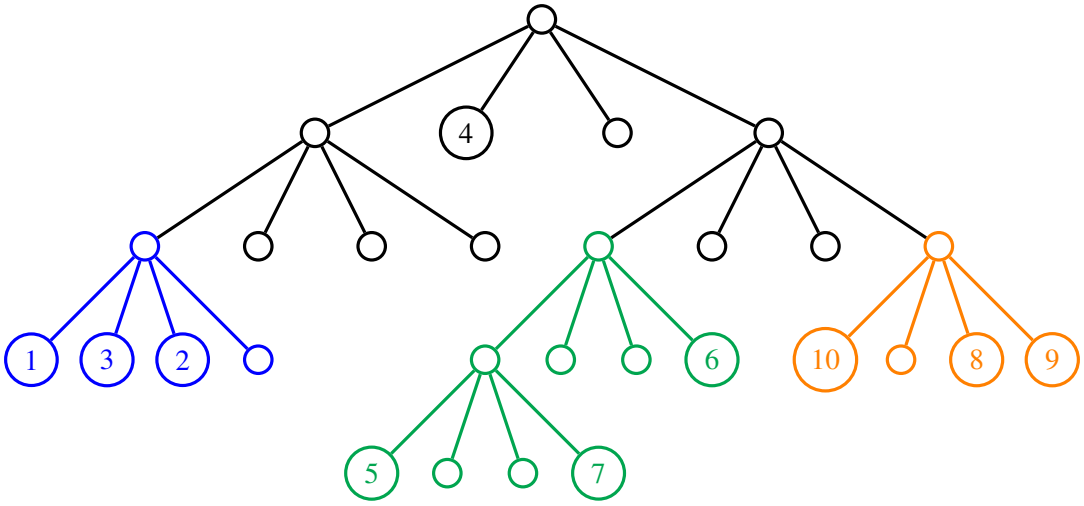


Figure 4.3: Creating all subtrees independently from each other after all nodes that belong to a subtree have been determined

With one work-group per subtree, one has much more work-groups and parallelism available in the second step. However, one is still limited to one work-group for the creation of the top of the octree and has additional overhead for distributing the bodies to the work-groups associated with the subtree that will contain the body. The performance differences of this approach with subtrees and the approach which is limited to one work-group will be further analyzed in chapter 5.3.2.

4.2.2 Calculating the center of mass

After the octree data structure is build, one can now calculate the center of mass and the sum of all masses for each node. Burtscher et al. extracted this part of the calculation into a separate step. It is theoretically possible to calculate the center of mass on the fly during the octree creation. However this requires extensive use of atomic operations and during the implementation process it turned out that this drastically reduces performance. Thus, in the SYCL implementation of the Barnes-Hut algorithm conducted during this thesis the computation of the center of mass is also performed in a separate step.

The general idea of Burtscher et al. is to traverse the octree bottom-up in parallel and compute the center of mass and sum of masses for each node. For the bottom-up traversal the authors make use of ordering guarantees for the order in which the nodes are stored. In the SYCL implementation of the algorithm in this work all nodes of the tree are stored in one buffer. The next insertion index is determined atomically during the octree creation. With this method it is guaranteed that child nodes will have a higher index as their parents. The data structure does not store pointers to the parent node of a child node explicitly. But because of this ordering guarantee, the octree can be traversed bottom up by starting with the last node in the buffer that stores all nodes and iterating through it until the root node at index 0 is reached.

In the algorithm each work-item gets assigned a certain subset of the nodes and starts with the node that has the highest index. If the node is a leaf node the center of mass can be calculated directly. If it is an inner node of the octree, it has to be checked if all child nodes of this node are already processed. If this is the case, the center of mass and sum of masses can be calculated by combining the respective values of the child nodes, if not, the work-item continues with the next node and revisits the node in a subsequent iteration over all nodes assigned to the work-item. This happens as long as not all nodes are processed yet. Burtscher et al. use the sum of masses value as a flag to indicate if a node has already been processed. This enables the algorithm to work without atomic operations. However, since it is not guaranteed that if a work-item sees the center of mass value, it will also see the center of gravity value of the node, a memory fence is needed again.

For the same reasons as with the octree creation, the usage of memory fences limits this kernel to one work-group for the SYCL implementation. Using an approach with subtrees again is much harder for this kernel, since with the current SYCL implementation all nodes are stored in one buffer, independent of what subtree they belong to. This makes it hard to filter out nodes belonging to one specific subtree without overhead. Separating the storage space, so that all nodes of one subtree are stored right next to each other is also not trivial, since estimating how many nodes will be created for one subtree and thus how much storage is needed is not simple. Even a subtree containing only two bodies can become really deep if the bodies are close to each other. This scenario is not unrealistic in astrophysics since such a case could correspond to a typical planet-moon system. Thus, in the SYCL implementation of this thesis most of the computations for the center of mass are limited to one work-group.

The assignment of nodes to work-items is a central part of this kernel and important for its runtime behavior. Since for nodes with higher indices it is more likely that they can be processed directly, these nodes should be processed first by the work-items. Figure 4.4 shows an example of two methods for assigning nodes two work-items. In this example 14 nodes get assigned to two work-items. Method 1 is based on the method described by Burtscher et al. where work-item i starts from the back with the node that has the highest index and processes every i-th node of the array.

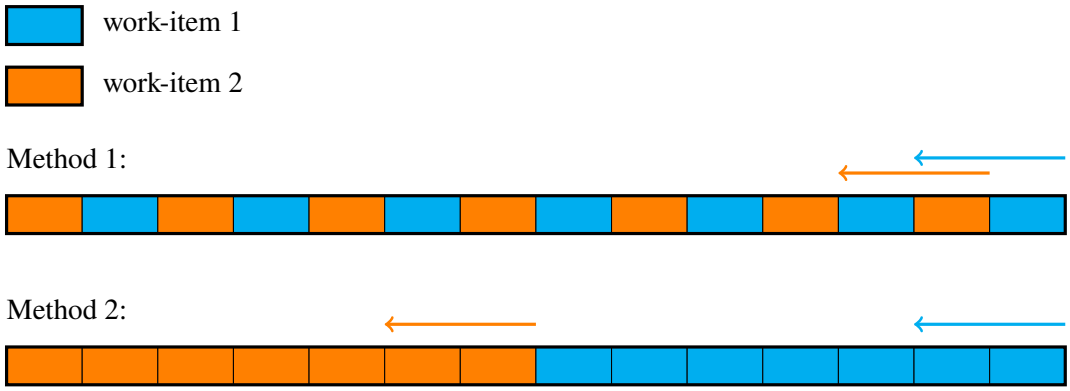


Figure 4.4: Example for the two methods of assigning nodes to two work items. Each cell in the arrays represents a node, the color denotes the work-item to which a node gets assigned. The arrows indicate the processing direction of each work-item. Method 2 performs worse since a lot of the computations of work-item 2 will depend on computations that work-item 1 has to do.
Based on [BP11] p. 82, figure 6.10.

Method 2 shows a different mapping between work-items and nodes. Here all nodes in the array are split into blocks and each block gets assigned to one work-item. Each work-item will again start processing with node that has the highest index. However, since in the SYCL implementation of this work child nodes always have higher indices as their parent node and the processing of the parent node depends on the processing of the child node, it is likely that many nodes assigned to work-item 2 in the example depend on nodes assigned to work-item 1, forcing work-item 2 to wait more often.

Because of this it is better to choose the first method for assigning nodes to work-items. However, method 1 does not work with the Open SYCL OpenMP backend on CPUs, since the algorithm does not terminate. On GPUs this methods works reliable with various backends. Since the performance gain of the first method over the second method on GPUs is not negligible, two separate kernels for GPUs and CPUs were implemented with SYCL. It is not fully clear why the method does not work with the Open SYCL OpenMP backend on CPUs and would require further research. One reason could be that method one of this algorithm heavily relies on each work-item making progress eventually. However, this property is not guaranteed by the SYCL standard and therefore depends on the implementation (see [Khr22] chapter 3.8.3.4.).

Optimizations for the center of mass computation

As described by Burtscher et al. the center of mass and sum of masses values of leaf nodes can be directly calculated since the computation of those values does not depend on any other node. This property is even more important in the SYCL implementation of this algorithm performed during this thesis. Since there are no dependencies for the computation of these values, a memory fence is not needed. Thus, in the implementation with SYCL, the calculations for all leaf nodes are extracted into a separate kernel. In this kernel a maximum amount of parallelism can be used since one is not

limited to one work-group anymore. Furthermore, it reduces the amount of work for the second kernel which is limited to only one work-group. Additionally, since the center of mass is already calculated for each leaf node, their parent nodes can directly be processed in the second kernel.

Since each work item often has to iterate over all nodes assigned to it several times it is better to cache nodes that still have to be processed. This optimization was also performed by Burtscher et al. In the implementation done during this thesis, all nodes that could not be processed during the first phase are stored in a separate buffer. For all subsequent iterations only theses nodes are considered. Like in the implementation of this kernel by Burtscher et al., the number of bodies that are contained in the cell represented through a node gets stored for each node. This is important for the sorting of the bodies which will be described in the next section.

4.2.3 Sorting the bodies with the tree

This part is not needed for the correctness of the algorithm. However, it is important for improving the performance of the acceleration computation on GPUs. Burtscher et al. use this step to minimize thread divergence on GPUs. The order of the bodies after the sorting step will correspond to the in-order tree-traversal of the octree that has been created.

In the SYCL implementation of this work, for each body the tree is traversed top to bottom until the leaf node of this body is reached. During each step, when a work-item steps down one level in the tree it has to determine how many nodes have to be stored before this node. This can be done using the value of how many bodies belong to a cell represented by a node which was calculated during the center of mass computation. The algorithm increments a counter with this value for all nodes whose bodies will be stored before the bodies of the current node, each time it steps down in the tree. The insertion index of the current body in the buffer of the bodies sorted according to an in-order tree-traversal is, thereby, determined when the leaf node containing the body is reached.

4.2.4 Computing the acceleration for each body

For each acceleration value $\vec{a}_i$ the octree has to be traversed top to bottom. However, it is not necessary to traverse the tree up to the leaf level every time. In the SYCL implementation of the Barnes-Hut algorithm this kernel has no constraints for the maximum amount of parallelism. Thus, one work-item computes exactly one acceleration value $\vec{a}_i$. In order to have more control over the mapping from work-groups to the corresponding primitives of the device specific backend, the work-group size can be explicitly specified via a parameter. To allow arbitrary work-group sizes a padding is added in a similar way as described in section 4.1.1.

Each work-item traverses the tree top to bottom with the usage of a stack. For each node two cases have to be considered. The first case is that the center of mass value of the node can be used as an approximation for the computation of $\vec{a}_i$. In this case, all nodes belonging to the subtree of this node do not have to be considered anymore. In the other case, if the approximation can not be made, all child nodes of the node have to be pushed on a stack, indicating that they still have to be processed later.

In order to minimize thread divergence it is desired that threads in one warp always execute code belonging to the same branch, i.e., consider the same of the two cases described above. This means that the decision whether an approximation can be used or not should be the same for the threads in one warp. It is likely that this is the case if the acceleration values the threads are computing belong to bodies that are close to each other spatially. This is why Burtscher et al. introduced the in-order sorting step of all bodies, since bodies which are close to each other in the in-order sorting of the octree are also likely to be close to each other spatially.

With SYCL, however, there is no direct control of mapping specific calculations to warps, so one has to rely on the SYCL implementation for this part. In the SYCL implementation of the Barnes-Hut algorithm there is a one-to-one mapping between the global ID of the work-item and the index of buffer holding the sorted bodies. More specifically, the work-item with ID 0 will compute the acceleration of the first body in buffer of sorted bodies, and so on. This results into work-items with consecutive IDs to compute acceleration values for bodies that also appear in consecutive order after the in-order sorting of all bodies. The effect of this performance optimization will be more closely analyzed in chapter 5.3.3.

5 Analysis of the runtime behavior

In this chapter the runtime behavior of the two algorithms implemented with SYCL will be analyzed. First, the two algorithms will be considered separately. For each algorithm there will be experiments analyzing the impact of their different parameters. After that the performance optimizations described in chapter 4 will be evaluated. Last, the overall runtime behavior of the two algorithms will be compared on different CPUs as well as on GPUs from different vendors.

5.1 Experiment setups

Unless stated otherwise, Open SYCL will be used for the experiments since it is compatible with all the hardware used in the experiments through the CUDA, ROCm and OpenMP backends. CUDA version 11.4.3 gets used on all systems with NVIDIA GPUs and ROCm 5.3.3 is used for the AMD GPU. The DPC++ version used for all experiments is sycl-nightly/20221102. For Open SYCL commit 4a04f1c is used. Open SYCL got build against the LLVM-compiler of the DPC++ installation.

For most runs, a simulation interval of 10 earth days was chosen. With a Δt of one hour, this results in 241 acceleration computations including the computation of the initial acceleration values. For some runs it was necessary to restrict the amount of acceleration computations to 25 or 13 due to the very long runtime. All individual timings of each run have been averaged. Generally, it was observed that there are a lot more fluctuations in the runtimes on CPUs than in the runtimes on GPUs.

The runtimes of the leapfrog integration steps are not included in the time measurements. One reason for this is that it is the same for the naive algorithm and the Barnes-Hut algorithm and thus does not contribute anything to a comparison of both algorithms. Furthermore, its runtime is negligible compared to the runtime of the other kernels of the simulation. For example even for the largest dataset the integration steps take only about half a millisecond on an NVIDIA RTX 3090.

The datasets used for the simulation contain real world data of objects in our solar system. The data originates from a data base of the NASA Jet Propulsion Laboratory[1]. For the experiments six differently sized datasets were provided containing 178, 999, 2678, 19054, 138723 and 1216869 bodies respectively.

Table 5.1 show the technical specifications of all five test systems that will be used during the experiments. Hardware that is explicitly used for the experiments is highlighted in bold. For experiments on the CPU, hyper-threading (HT) was enabled and omp.accelerated of Open SYCL

[1] https://ssd.jpl.nasa.gov/tools/sbdb_query.html (visited on 04/03/2023)

	System 1	System 2	System 3	System 4	System 5
GPU	**NVIDIA A100**	**NVIDIA Quadro GP100**	**NVIDIA GeForce RTX 3090**	**AMD Radeon PRO VII**	-
GPU memory	**40GB HBM2**	**16GB HBM2**	**24GB GDDR6X**	**16GB HBM2**	-
GPU FP64 performance [TFLOPS]	**9.7**	**~5**	**0.556**	**6.5**	-
CPU	*2x AMD EPYC 7742*	*2x Intel Xeon Silver 4116*	**AMD Ryzen Threadripper 3960X**	*Intel i7-6700K*	**2x AMD EPYC 7543**
CPU core count	*128*	*24*	**24**	*8*	**64**
CPU clock speed [GHz]	*3.4*	*3.0*	**4.5**	*4.2*	**3.7**

Table 5.1: Overview of all systems used for the experiments. Hardware that is explicitly used for the experiments is highlighted in bold. All the other hardware of the systems is also shown for completeness. The CPU-clock speed corresponds to the maximum single-core speed as specified from the manufacturer. The information for the hardware specifications have been retrieved from [NVI17], [NVI20], [NVI21], [AMD20b], [AMD20a], [Intb], [AMD], [Inta], [AMD22]. If no FP64 performance was specified the amount of tera floating point operations per second (TFLOPS) was calculated using the FP32 performance and the the single precision, double precision ratio of the respective GPU.

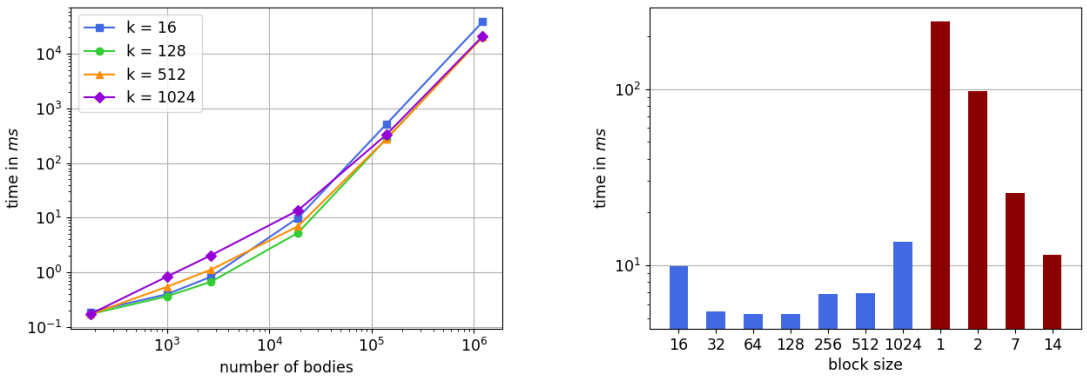
was used. All calculations were performed using FP64 double precision. If not stated otherwise, all runtimes correspond to the time of one time step without leapfrog integration. If the runtime corresponds to a specific part of the algorithm, the runtime of this part in one time step is given.

5.2 Impact of the parameters on the runtime behavior of the algorithms

In this section the impact of the different parameters for both algorithms on their runtime behavior will be analyzed. First, the block size of the naive algorithm will be considered. After that the different parameters of the Barnes-Hut algorithm will be studied, starting with the parameters that influence the runtime behavior of the octree creation: the maximum level of the top of the octree and the amount of parallelism available for the first and second phase of the octree creation. Last, the impact of the θ-value, that gets used to determine the accuracy of the Barnes-Hut algorithm, on the runtime of the acceleration kernel will be analyzed.

5.2.1 Impact of the block size on the naive algorithm

The goal of the experiments presented in this section is to analyze the impact of the block size of the naive algorithm on its runtime behavior. The first experiment analyzes the impact of the block size of the naive algorithm on GPUs. In the experiment, the runtime of the naive algorithm with different block sizes was measured on an NVIDIA Quadro GP100 using all 6 datasets.



(a) Runtime comparison between different block sizes k and different numbers of bodies.

(b) Comparison of different block sizes for 19054 bodies.

Figure 5.1: Comparison of different block sizes k for the naive algorithm on an NVIDIA Quadro GP100 GPU. Figure 5.1a shows the runtime behavior for four different block sizes on all datasets. A block size of 128 performs well for most datasets. Figure 5.1b shows the runtimes of several block sizes for the dataset containing 19054 bodies. The blue bars denote block sizes that are a power of two whereas the red bars correspond to block sizes that are divisors of 19054 and can be used as block sizes. It can be clearly seen that most block sizes that are a power of two perform better than the divisors of 19054.

Figure 5.1a shows the runtimes of the naive algorithm for all datasets and for different block sizes k. Each curve corresponds to one block size. The number of bodies used for the simulation is shown on the x-axis and the y-axis corresponds to the runtime in milliseconds. Both, the x-axis and the y-axis are logarithmically scaled.

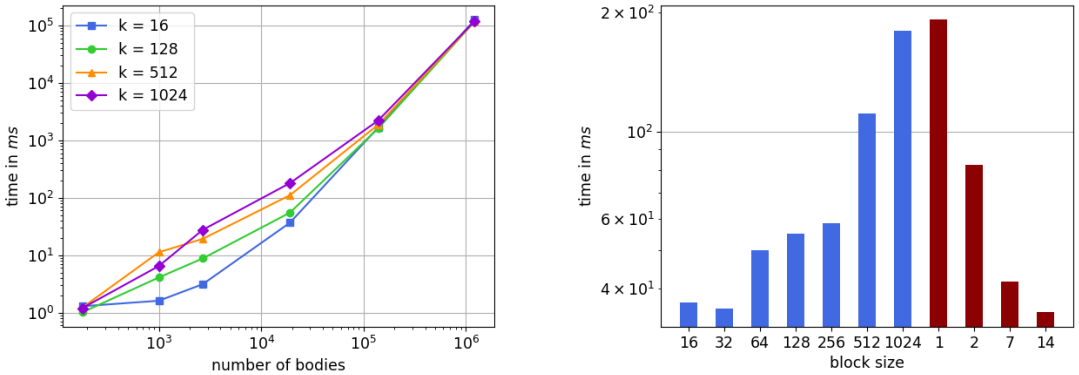
One can observe that small values for k like 16 do not perform well for a large numbers of bodies. With $k = 16$ the naive algorithm takes about 38.8s whereas with with $k = 128$ it only takes about 20.6s. This could be explained with the fact that for small block sizes the overall amount of blocks is much higher than for large block sizes. Thus, one has more overhead for the blocking compared to larger block sizes which perform much better for large body counts. However, in figure 5.1a it can be recognized that large values like $k = 1024$ do not perform well for small amounts of bodies. The latter can not be observed for the smallest dataset of 178 bodies since there can not be more than 178 work items performing calculations, thus larger k do not make a difference.

Figure 5.1b shows a bar plot with the runtimes of the naive algorithm for the dataset with 19054 bodies. Each bar corresponds to one block size k. In addition to the values for k which are highlighted in blue and are a power of two, four other block sizes are also considered: $k = 1$, $k = 2$, $k = 7$ and $k = 14$. These values correspond to all divisors of 19054 that are smaller or equal than the maximum

work-group size of 1024 and are highlighted in red. They are interesting because they would be the only valid work-group sizes and thus block sizes one could use without having a padding that allows for arbitrary k. Since the runtimes for the divisors are a lot slower than the runtimes for most of the k which are a power of two, one can conclude that the work-group size should be chosen as a power of two for optimal performance. Further testing conducted during this thesis showed that a multiple of a power of two is also sufficient.

One can see that $k = 128$ performs well for most datasets. Thus, this value will be used for all future experiments with the naive algorithm on GPUs unless stated otherwise.

In order to study how the block size influences the runtime behavior on CPUs, the previous experiment was repeated on CPUs. The system used for this experiment contains a dual socket AMD EPYC 7543 with a total of 64 cores and HT enabled.



(a) Runtime comparison of different block sizes k and different numbers of bodies.

(b) Comparison of different block sizes for 19054 bodies.

Figure 5.2: Comparison of different block sizes k for the naive algorithm on a dual socket AMD EPYC 7543 with 64 CPU cores. Figure 5.2a shows the runtime behavior for four different block sizes on all datasets. The x-axis corresponds to the number of bodies contained in the dataset. Figure 5.2b shows the runtimes of several block sizes for the dataset containing 19054 bodies. The blue bars denote block sizes that are a power of two whereas the red bars correspond to block sizes that are divisors of 19054 and can be used as block sizes. The fact that the block size is a power of two does not play an important role on CPUs. The overall size of the value is more important.

Figure 5.2 shows the the results of this experiment in the same scheme as used for the previous experiment. In figure 5.2a it can be observed that smaller values for k perform much better for smaller datasets. For a large amounts of bodies this can not be observed anymore and large k even perform a little bit better. For example for the largest dataset $k = 16$ would need about 125.6s whereas $k = 128$ needs only 117.4s.

Figure 5.2b shows the runtime for different block sizes with a dataset containing 19054 bodies in more detail. Like in the previous experiment, in addition to the values for k that are a power of two, all divisors of 19054 smaller than 1024 are shown again. Most of the divisors of 19054 do not

perform well again. However, this is most likely due to their small size which results into a large amount of blocks and thus increases the overhead. Unlike on GPUs $k = 14$ performs quite well on CPUs.

From that one can conclude that, unlike on GPUs, it is not that important that k is a power of two and the overall size of k plays a bigger role. The observation that small block sizes perform better than big block sizes for small numbers of bodies could be explained with an increased probability of cache hits when small blocks are used. However, this advantage vanishes for large numbers of bodies since the small block size creates additional overhead.

A value of $k = 128$ performs well for most datasets. Thus, this value will be used for all future experiments with the naive algorithm on CPUs unless stated otherwise.

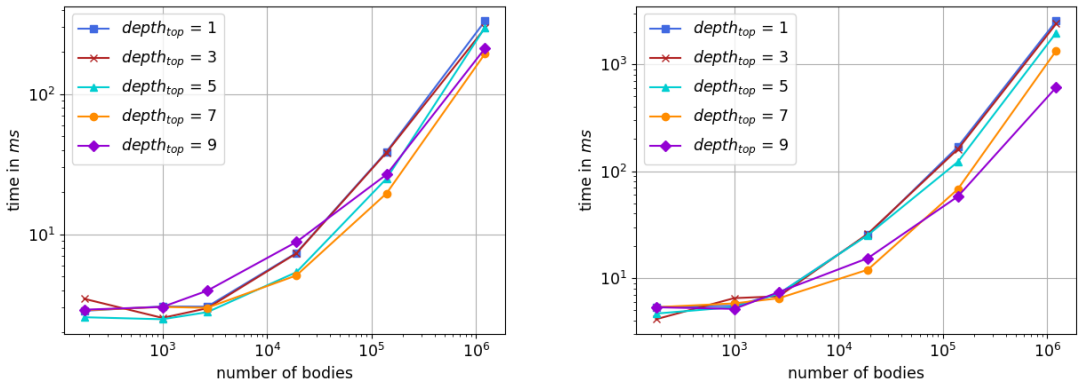
5.2.2 Impact of the Barnes-Hut tree-building parameters

The goal of the experiments presented in this section is to study how the different parameters that adjust the octree creation of the Barnes-Hut algorithm influence the runtime of the octree creation on CPUs and GPUs. The first parameter that will be considered is the maximum level to which the top of the tree gets build in the first step. The subtrees that emerge in the first step then get build in parallel in the second step. The second and the third parameter that will be analyzed in this section determine the amount of parallelism used for the first and second step of the tree creation respectively. Generally it has to be noted that the runtime behavior of the octree creation highly depends on the dataset used for the simulation since the positioning of the bodies in the space determines the structure of the tree. Thus, the parameters might behave differently on different datasets.

The depth of the top of the octree

The experiment that will be presented in this section aims to analyze how to choose the maximum tree depth for the top of the octree that gets build in the first part of the octree creation. In the experiment the runtime of the whole octree creation was measured using different values for the maximum tree depth for the top of the octree. Generally, a higher value for this maximum depth will result into more work for the first phase where less parallelism is available and generate many small subtrees. A smaller maximum tree depth for the top of the octree will result into less work for the first phase and generates fewer, but larger subtrees.

Figure 5.3 shows the results of the experiment on an NVIDIA Quadro GP100 GPU and a dual socket AMD EPYC 7543 CPU. The x-axes correspond to the number of bodies used for the simulation and the y-axes show the time for the octree creation in milliseconds. Both axes are logarithmically scaled. In Figure 5.3a, which shows the results for the NVIDIA Quadro GP100, it can be observed that a depth of seven yields the best results across most datasets. Very low values like one or three perform worst. For example the octree creation on the largest dataset requires only about 194.4ms with a depth of seven, whereas with a depth of three the computation requires about 295.4ms. This is likely because with low values for the depth of the top of the octree the amount of subtrees that emerge is too small and the subtrees itself can become very deep. Thus, the advantage of the increased parallelism is very limited. On the other hand, large values like nine also result into sub-optimal performance. Part of the reason for this behavior could be that the first part of the



(a) NVIDIA Quadro GP100 **(b)** Dual socket AMD EPYC 7543

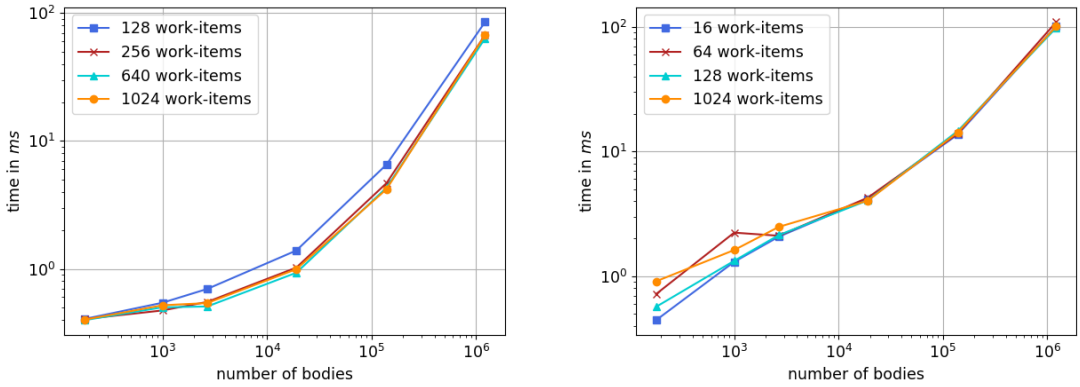
Figure 5.3: Comparison of how different values for the maximum depth for the top of the octree influence the time for the octree creation. Figure 5.3a shows the impact of the depth of the top of the octree on an NVIDIA Quadro GP100 GPU whereas figure 5.3b shows the results of the same experiment repeated on a dual socket AMD EPYC 7543. It can be observed that on the CPU the best results for larger datasets can be obtained with a depth of nine. On the GPU seven yields the best results across most datasets.

octree creation requires more work the deeper the top of the tree gets. Thus, there is more work for which one is limited to one work-group and thus can not use the full amount of parallelism available on the GPU. Since with higher values one has more subtrees, and thus more parallelism, in the second phase, one can compensate the performance loss of the first phase. However, this is only possible if the GPU can also fully use this increased amount of parallelism. If there is more theoretical parallelism than can be used there might not be a performance gain anymore. Overall, a depth of seven seems to balance the amount of work in the first phase and the available parallelism in the second phase best. Thus, this value will be chosen for all future experiments on GPUs if not stated otherwise.

Figure 5.3b shows the results of the experiment on a dual socket AMD EPYC 7543 CPU. One can see that, similarly to GPUs, small values for the depth of the top of the tree perform worst, especially for large datasets. However, nine performs best for larger datasets on CPUs. With a runtime of approximately 607.9ms it is a lot faster than with a depth value of seven which would require 1337.2ms for the octree creation on the largest dataset. This was not the case on GPUs. One reason for this behavior could be that CPUs favor the many small subtrees that emerge when using a larger tree depth for the top of the octree. These smaller subtrees can then be build very fast without the need for a lot of synchronization, since they contain fewer bodies than larger subtrees. One could also argue that the reason for this behavior is that this approach with subtrees is not suited for CPUs and larger tree depths for the top of the tree shift more work to the first phase which is more similar to the original approach without subtrees. However, as it will be shown in section 5.3.2, the approach with subtrees improves the performance over the approach without subtrees on CPUs. Thus, it is more likely that the reason for higher values for the top of the tree performing better is due to the increased amount of subtrees and not only because more work is shifted to the first phase of the tree creation. Since a depth for the top of the tree of nine performed best for the larger datasets, this value will be chosen for CPUs unless stated otherwise.

Choosing the amount of parallelism for the two phases of the octree creation

The experiments presented in this section will analyze how the amount of parallelism available for the first and the second phase of the octree creation influences the runtime behavior of the overall octree creation on CPUs and GPUs. Generally the usage of more work-items results in more parallelism but also requires more synchronization between those work-items which introduces additional overhead. This is why it is important to analyze the impact of these values. In order to do this, the runtime of the first and the second phase of the octree creation was measured on an NVIDIA Quadro GP100 GPU and on a dual socket AMD EPYC 7543 CPU using different amounts of parallelism.



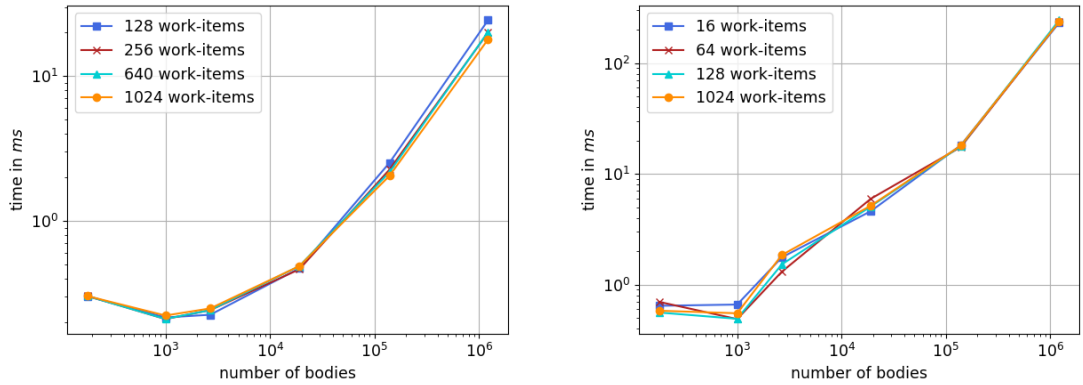
(a) NVIDIA Quadro GP100 **(b)** Dual socket AMD EPYC 7543

Figure 5.4: Comparison of different amounts of work-items for the first phase of the octree creation on an NVIDIA Quadro GP100 GPU and on a dual socket AMD EPYC 7543 CPU. On the GPU, small work-item counts like 128 perform worst and higher values perform better. For larger datasets there is not such a large difference on CPUs between small and large work-item counts.

Figure 5.4 shows the runtimes for the creation of the top of the tree on CPUs and GPUs for different work-item counts. The x-axes correspond to the number of bodies used for the simulation and the y-axes correspond to the runtime for the creation of the top of the tree in milliseconds. All axes have been logarithmically scaled.

Figure 5.4a shows the impact of changing the work-item count for the first phase of the octree creation on an NVIDIA Quadro GP100 GPU. It can be seen that smaller values like 128 result in the worst performance with a runtime of about 85ms for the largest dataset. But also the largest possible value of 1024 does not result in the best performance across all datasets. The results indicate that choosing higher values results into better performance for this part of the algorithm. Nevertheless, the highest possible value is not always the best choice, since it is about 3.8ms slower than when using 640 work-items for the largest dataset. However, smaller values like 128 clearly perform worst. A work-item count of 640 offers good performance across all datasets and even performs best for the largest dataset. Thus, this value will be chosen on GPUs for the final experiments unless otherwise stated.

Figure 5.4b presents the results when varying the amount of work-items for the first part of the octree creation on a dual socket AMD EPYC 7543 CPU. In contrast to the version of this experiment on GPUs, the variation of the runtime between the different numbers of work-items is much smaller for lager datasets. Part of the reason for this could be that if the work-item count exceeds the thread count of the CPU (which is 128 in this case) there is no additional performance to gain. However the overhead of using much more work-items seems to be very limited since these values do not perform much worse for larger datasets. A work-item count of 128 performs best with a small margin for the largest dateset where it is approximately two to three milliseconds faster than using 1024 or 16 work-items. Since it also performs well for smaller datasets, this value will be chosen for the final experiments on CPUs.



(a) NVIDIA Quadro GP100 **(b)** Dual socket AMD EPYC 7543

Figure 5.5: Comparison of different amounts of work-items per work-group when building the subtrees during the octree creation. Each work-group is responsible for building one subtree. Figure 5.5a shows the runtimes for different amounts of work-items per work-group on an NVIDIA Quadro GP100 GPU. Figure 5.5b shows these values for the same experiment on a dual socket AMD EPYC 7543 CPU. On the GPU higher work-group sizes perform better than smaller ones. On the CPU there are almost no differences between the runtimes for large and small work-group sizes when large datasets get used.

Figure 5.5 shows the runtimes for the second phase of the octree creation on an NVIDIA Quadro GP100 and and a dual socket AMD EPYC 7543 for different work-item counts. The work-item count corresponds to the amount of work-items in one work-group. Each work-group is responsible for building one of the subtrees. The x-axes correspond to the number of bodies used for the simulation and the y-axes correspond to the runtime for the creation of subtrees in milliseconds. All axes have been logarithmically scaled.

In figure 5.5a the runtimes measured on an NVIDIA Quadro GP100 are shown. One can observe that for smaller datasets, smaller values perform a little bit better whereas 1024 performs best for the largest dataset. However, this value only works reliably on all GPUs considered during this work when using Open SYCL. When DPC++ gets used the program is unable to launch this kernel with such a high number of work-items. This could be due to the amount of registers needed gets too

large. Since a work-item count of 640 also performs well across most datasets, being only 2.2ms slower than 1024 for the largest dataset, and runs stable on all GPUs with both Open SYCL and DPC++, this values will be chosen for the final experiments.

Interestingly, the total time needed for the creation of the subtrees with the dataset containing 999 bodies is lower than the time for the dataset with just 178 bodies. This is because these two datasets are very different. The dataset with just 178 bodies contains planets like Jupiter or Saturn which have a lot of moons. This results into a really deep octree. The dataset containing 999 bodies only contains asteroids which are sparsely distributed. Thus the depth of the resulting octree is much smaller. Since the depth for the top of the tree is fixed across all datasets, the subtrees that get created for the second dataset are much smaller compared to the first one.

Figure 5.5b presents the runtimes for the creation of the subtrees on a dual socket AMD EPYC 7543 CPU. It can be observed that similar to the results from the first phase of the octree creation the choice of the amount of parallelism does not make a huge difference, especially for larger datasets. Since 16 performs well for small and large datasets this value will be chosen for the final experiments on CPUs.

5.2.3 Effect of the parameters for the acceleration computation of the Barnes-Hut algorithm

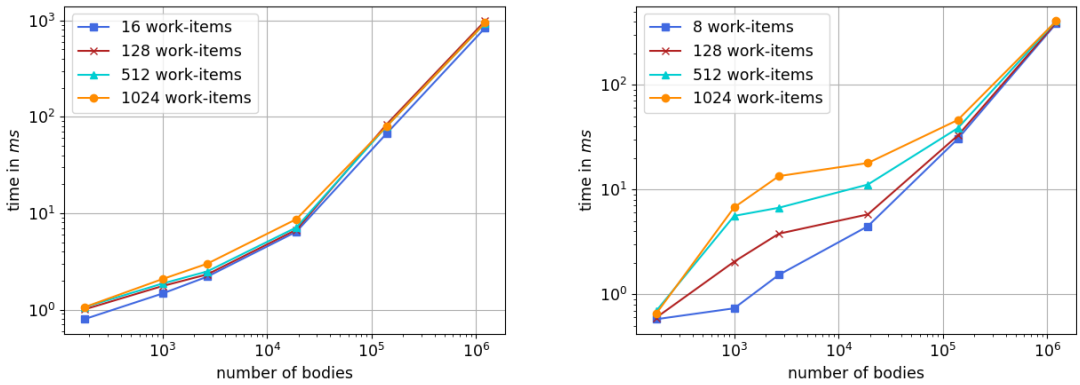
The goal of the experiments presented in this section is to analyze the impact of the parameters that influence the runtime of the acceleration kernel in the Barnes-Hut algorithm. First, the work-group size of acceleration kernel will be considered. After that the impact of the θ-value on the accuracy and runtime of the Barnes-Hut algorithm will be analyzed.

Impact of the work-group size on the acceleration kernel

Since no optimizations making use of local memory are made to the acceleration kernel of the Barnes-Hut algorithm and it also does not require any form of synchronization, it is not necessary to explicitly group the work-items into work-groups. However, in practice the specification of the work-group size gives some control over how work-items are mapped to, e.g., thread-blocks on NVIDIA GPUs. The experiment presented in this section will analyze the impact of the work-group size for the acceleration kernel on CPUs and GPUs.

Figure 5.6 shows the runtimes for different work-group sizes for the Barnes-Hut acceleration kernel on an NVIDIA Quadro GP100 GPU and a dual socket AMD EPYC 7543 CPU for different work-group sizes. The x-axes correspond to the number of bodies used for the simulation and the y-axes correspond to the runtime of the acceleration kernel. All axes have been logarithmically scaled.

In figure 5.6a the results of the runtimes for different work-group sizes on an NVIDIA Quadro GP100 are presented. One can see that a work-group size of 16 results into the best performance on this GPU. With a runtime of about 837.2ms for the largest dataset it performs much better than a work-group size of, e.g., 1024 which results into a runtime of about 944.8ms for this kernel. For the final experiments 16 will be chosen as work-group size for the acceleration kernel on the NVIDIA Quadro GP100. However, further testing revealed that this parameter has to be chosen differently



(a) NVIDIA Quadro GP100 **(b)** Dual socket AMD EPYC 7543

Figure 5.6: Comparison of different work-group sizes for the acceleration kernel of the Barnes-Hut algorithm. Figure 5.5a shows the runtimes for different amounts of work-group sizes on an NVIDIA Quadro GP100 GPU. Figure 5.5b shows these values for the same experiment on a dual socket AMD EPYC 7543 CPU. For the GPU a value of 16 performs best across most datasets. For the CPU a work-group size of 8 performs best across most datasets, especially for small ones.

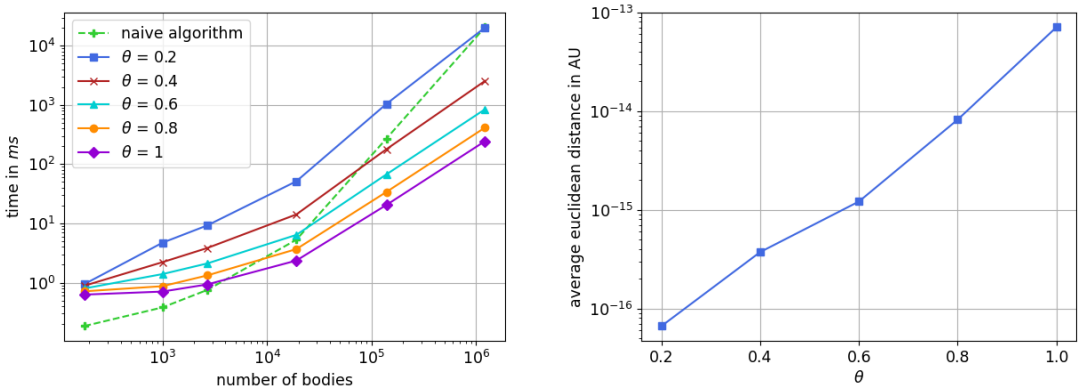
on other GPUs in order to achieve good performance. Since the performance differences are not insignificant, the value was chosen differently for all GPUs that will be considered in this thesis. The decision which value gets used is based on which of the work-group sizes performed best across most of the datasets in my experiments. For the NVIDIA RTX 3090 a work-group size of 32 will be chosen. The experiments with the NVIDIA A100 will use a work-group size of 1024 and for the AMD Radeon PRO VII a size of 64 gets used for each work-group.

Figure 5.6b shows the effect of different work-group sizes on a dual socket AMD EPYC 7543 CPU. It can be observed that especially for small numbers of bodies there are huge differences between different work-group sizes. For example with the dataset consisting of 2678 bodies the acceleration kernel takes about 1.5ms whereas with a work-group size of 1024 the runtime is increased to about 13.4ms. Generally it can be observed that smaller work-group sizes perform better than bigger work-group sizes. One reason for this could be that the specification of the work-group size might influence how work-items are mapped to threads in the background and smaller work-groups result into situations that improve the cache performance. However, further explaining this observation would require more research about how work-items of work-groups are mapped to threads by Open SYCL. Since a work-group size of eight performs best across most datasets, this value will be chosen for the final experiments on CPUs.

Impact of the θ-value on runtime and accuracy

The θ-parameter is a central part of the Barnes-Hut algorithm. It is used to control its accuracy and also effects the performance of the algorithm. In order to analyze the impact of this parameter on the runtime of the Barnes-Hut algorithm, the runtime of the acceleration kernel was measured using different values for θ. Furthermore, in order to determine the impact of this value on the accuracy

of this algorithm, one earth year was simulated with the dataset containing 2678 bodies and a Δt of one hour. This was done for several θ-values and the resulting final positions of all bodies got compared to the same simulation conducted with the naive algorithm.



(a) Runtimes for different θ-values on an NVIDIA Quadro GP100.

(b) Average euclidean distance between the final states of the Barnes-Hut algorithm using different theta values and the final state of the naive algorithm after one earth year.

Figure 5.7: Effect of the θ-value of the Barnes-Hut algorithm on the runtime and accuracy. Figure 5.7a shows the runtimes of acceleration kernel on an NVIDIA Quadro GP100 for different θ-values. Figure 5.7b shows how the θ-value impacts the accuracy of the algorithm. It compares the last state of the simulation performed by the Barnes-Hut algorithm with the last state of the simulation that used the naive algorithm. One can see that bigger θ-values also result into a lower runtime. However, when a high value for θ gets chosen the accuracy of the Barnes-Hut algorithm is decreased.

Figure 5.7a shows the results of the experiment regarding the impact of the θ-value on the runtime of the Barnes-Hut algorithm. The experiment was conducted on an NVIDIA Quadro GP100 GPU. The x-axis corresponds the size of the dataset used for the simulation. The y-axis shows the runtime of the acceleration kernel in milliseconds. Both axes have been logarithmically scaled. Each curve corresponds to one run of the simulation using a different θ-value. One can see that the θ-value has a big impact on the runtime of the acceleration kernel. With $\theta = 0.2$ the runtime of the kernel in one time step is about 20.4s. When using $\theta = 0.4$ the runtime is decreased substantially to 848.8ms. With $\theta = 1$ the runtime of the acceleration kernel further decreases to about 245.23ms. However, as it will be shown in next paragraph, these lower runtimes come at the cost of accuracy.

Figure 5.7b shows the results of the experiment that analyzes the impact of the θ-value on the accuracy of the simulation. The x-axis corresponds to the θ-value used for the simulation. The y-axis denotes the average euclidean distance between the positions of the same bodies in the final states of the Barnes-Hut algorithm and the naive algorithm after simulating one earth year. The unit for the distance is the astronomical unit (AU). It can be observed that with higher θ-values the precision of the simulation gets reduced significantly. The overall average distance of deviation might not be that high, however there are many bodies in the dataset where no deviation can be

observed with the accuracy used for the visualization of the simulation. Nevertheless, with higher θ-values some planet moon-systems can become unstable if the simulation spans across a large amount of time steps.

For this thesis the overall runtime behavior of the Barnes-Hut algorithm is most important. In order to reflect a good middle ground between accuracy and runtime of the algorithm a θ-value of 0.6 will be chosen for all future experiments.

5.3 Evaluation of the performance optimizations

The goal of the experiments described in this section is to evaluate the three performance optimizations described in chapter 4. First, the usage of local memory in the naive algorithm will be analyzed. After that the performance impact of using subtrees during the octree creation of the Barnes-Hut algorithm will be studied. Last, the impact of the sorting step will be examined.

5.3.1 Evaluation of the optimizations for the naive algorithm

The experiments presented in this section aim to analyze the impact of the performance optimizations performed on the naive algorithm. This analysis will deal with the following topics:

1. Impact of the optimizations on GPUs
2. Different effects on consumer and data center GPUs
3. Differences between SYCL implementations
4. Impact on the runtime on CPUs

In order to gain a better understanding of the impact of the individual optimization steps, three optimization stages for the naive algorithm will be considered. Stage zero corresponds to the implementation of the naive algorithm without any optimizations. Stage one introduces the grouping of acceleration computations into work-groups, including a padding to allow arbitrary work-group sizes. Stage two corresponds to the final implementation, including the usage of local memory.

Impact of the optimizations on GPUs

In order to analyze the impact of the optimizations on GPUs the naive algorithm was run on all datasets with all three optimization stages using an NVIDIA Quadro GP100 GPU. For stage one and stage two, a work-group size of 128 has been chosen. Figure 5.8 shows the results of theses runs. The x-axis corresponds to the number of bodies. The y-axis shows the runtime in milliseconds and is logarithmically scaled. Each set of three bars corresponds the runs of the three different stages on one dataset.

It can be seen that stage two is always faster than stage one and stage zero. For the largest dataset the optimizations from stage two improve the runtime from about 22.2 seconds to 20.6 seconds per acceleration computation. Since n-body simulations usually consist out of a lot of time steps, this improvement adds up over time and can significantly improve the overall runtime of the simulation.

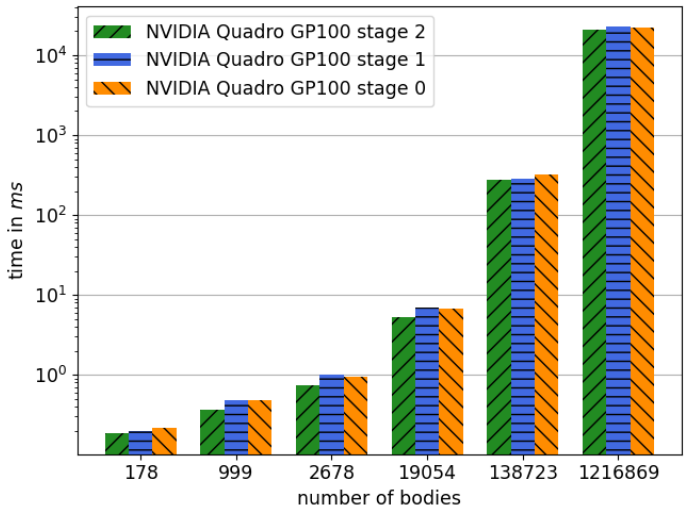


Figure 5.8: Analysis of the optimizations for the naive algorithm on an NVIDIA Quadro GP100. Stage 0 is the non optimized version of the naive algorithm and stage 2 is the final, most optimized version of the naive algorithm. It can be observed that stage 2 can improve the runtime of the naive algorithm most of the time

Furthermore, it can be seen that stage one does not always offer an improvement over stage zero and sometimes even results into worse performance. This indicates that the performance improvement indeed comes from the usage of local memory and not just from the division of work-items into work-groups. The fact that stage one sometimes results into slightly worse performance can be explained with the fact that the work-group size which was chosen as 128 for all datasets. This value produces good results for all datasets, however, there might be specific work-group sizes that perform better on some datasets.

Different effects on consumer and data center GPUs

To study this topic, the previous experiment was repeated using a different GPU, an NVIDIA GeForce RTX 3090. As opposed to the NVIDIA Quadro GP100 used previously, this is not a data center GPU but a consumer GPU. The main difference between those two categories of graphics cards is that data center GPUs typically have a lot more double precision performance than consumer GPUs. This is especially relevant for the implementation of the naive algorithm conducted in this work, since all calculations are performed using double precision.

Figure 5.9 shows the results of this experiment in the same manner as previously. It can be observed that, as opposed to the run of the experiment on the Quadro GP100, no performance differences between the three stages can be seen on the GeForce RTX 3090. This can be explained with the much worse double precision performance of this consumer GPU compared to the data center GPU used in the first experiment. Because of this, the NVIDIA GeForce RTX 3090 is likely heavily bottle-necked by its compute performance. Thus, the memory optimization of stage 2 does not come into effect. The much higher overall compute performance of the Quadro GP100 can also be

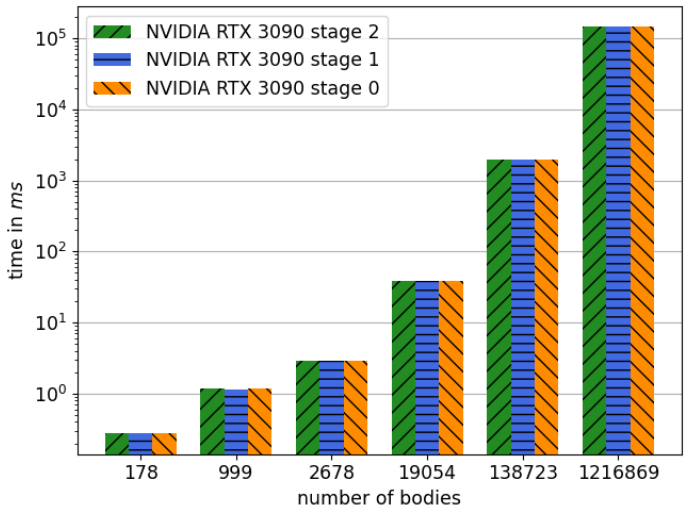


Figure 5.9: Impact of the optimizations for the naive algorithm on an on an NVIDIA GeForce RTX 3090. Stage 0 is the non optimized version of the naive algorithm and stage 2 is the final, most optimized version of the naive algorithm. It can be seen that stage 2 does not improve the runtime of the naive algorithm on consumer GPUs.

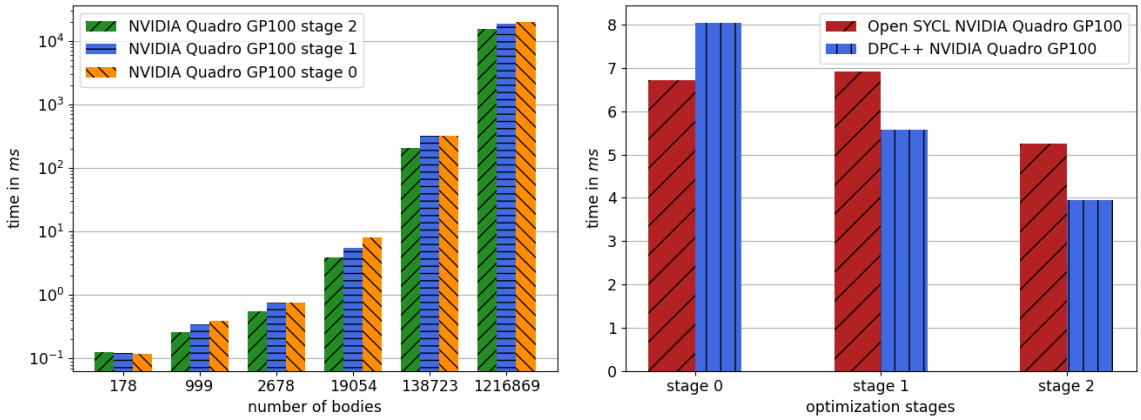
clearly seen in the runtime differences between the two cards. The Quadro GP100 needs about 20.6 seconds for one acceleration computation on the largest dataset whereas the RTX 3090 takes about 2.4 minutes. This makes consumer GPUs impractical for the naive algorithm on larger datasets.

Differences between SYCL implementations

In order to analyze differences between the two different SYCL implementations that are supported, the experiment was repeated using DPC++. Furthermore, in order not to limit the observations to NVIDIA GPUs, the experiment was repeated again on an AMD Radeon PRO VII GPU with both, Open SYCL and DPC++.

Figure 5.10a shows the results of the first experiment on a Quadro GP100 with the usage of DPC++. One can see that the optimization also works with DPC++ since stage two improves the runtime on most datasets with one exception being the smallest dataset. An explanation for this exception could be again the choice of the work-group size as 128 for all datasets. For the smallest dataset with just 128 bodies this is likely not the best value, however, it performs better for larger datasets and the performance loss for 178 bodies is not that large. As opposed to the results of the analogous experiment with Open SYCL one can observe that stage one often improves performance over stage zero. This phenomenon could not really be observed with Open SYCL.

Figure 5.10b shows a comparison between the runtime of the 3 different stages with DPC++ and Open SYCL. The dataset used for the experiment contains 19054 bodies. The y-axis denotes the runtime in milliseconds. The left bar of each group shows the runtime of Open SYCL and the right bar of a group of two bars shows the runtime of DPC++. It can be observed that Open SYCL is much faster than DPC++ at stage 0. With stage one, the performance of DPC++ improves a lot from about 8.04ms for one time step to only 5.57ms. This is faster than Open SYCL which needs about 6.91ms for one time step with stage one. Stage one, however, has almost no effect when using



(a) Impact of the performance optimizations on an NVIDIA Quadro GP100 using DPC++

(b) Impact of the performance optimizations with DCP++ and Open SYCL for 19054 bodies

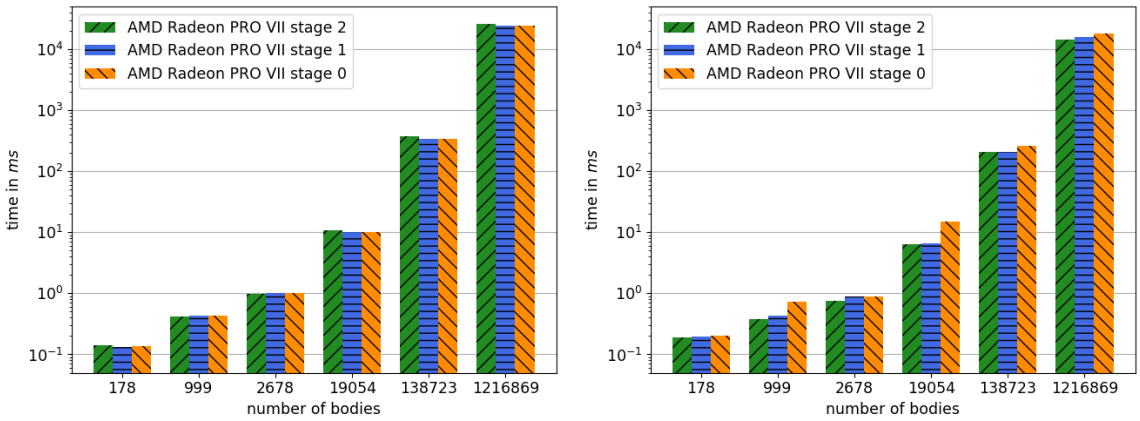
Figure 5.10: Differences between DPC++ and Open SYCL for the different optimization stages of the naive algorithm on an NVIDIA Quadro GP100. Stage 0 is the non optimized version of the naive algorithm and stage 2 is the final, most optimized version of the naive algorithm. Figure 5.10a shows the runtimes of the naive algorithm with the different optimizations on an NVIDIA Quadro GP100. Figure 5.10b shows a comparison between DPC++ and Open SYCL with the different stages of the naive algorithm. It can be seen that, in contrast to Open SYCL, stage 1 also offers performance improvements in some cases.

Open SYCL, since Open SYCL also only needed 6.7ms with stage zero. This is means that stage one is even slightly worse than stage zero when using Open SYCL, but the difference could also be a measuring error. Stage two improves the runtime for both, DPC++ and Open SYCL. Overall, with stage two DPC++ only needs about 3.96ms for one time step. With 5.26ms, Open SYCL needs slightly longer in this situation.

In order to explain the differences between Open SYCL and DPC++ when going from stage zero to stage one, the application was profiled. Profiling was done on different system than the one where the experiment was run using slightly different SYCL versions and a different GPU. However, the same behavior could also be observed on this system. Stage one only introduced the division of work-items into work-groups including a padding to allow for arbitrary work-group sizes. The likely reason why this has such a big impact on DPC++ but not on Open SYCL can be seen when profiling the application. In the CUDA backend, DPC++ maps the 19054 work-items of the acceleration kernel to 1361 thread blocks consisting of 14 threads each. These values were likely chosen since 14 divides the total amount 19054 work-items. However, as shown in section 5.2.1, 14 is not a good choice for the size of a thread block. A better choice would be a power of two. This is why stage one introduced the padding to allow for arbitrary work-group sizes which allows for more control over how the work-groups are mapped to thread blocks.

When profiling the application which was build with Open SYCL it turned out that such a manual padding is not necessary when using Open SYCL. In the CUDA backend of Open SYCL the 19054 work-items got mapped to 149 thread blocks of size 128. This size is a much better choice and thus results into better performance. Open SYCL was able to choose this thread block size since it increased the overall amount threads from 19054 to 19072. This is basically the same concept

as applied in stage one where the total amount of work-items is increased in order to be divisible through the desired work-group size. The additional work-items then just do not perform any work at all in stage one. Since in stage one the manually specified work-group size was also 128 there is almost no difference between the two stages for Open SYCL. The fact that the manual implementation in stage one results into slightly worse performance could be an indicator the implementation of Open SYCL is slightly more efficient. However this manual implementation is needed to achieve good performance on all datasets with DPC++ and the usage of local memory in stage two requires a manual grouping of work-items into work-groups.



(a) Impact of the performance optimizations on an AMD Radeon PRO VII using Open SYCL

(b) Impact of the performance optimizations on an AMD Radeon PRO VII using DPC++

Figure 5.11: Impact of the optimizations of the naive algorithm on an AMD GPU with Open SYCL and DPC++. Stage 0 is the non optimized version of the naive algorithm and stage 2 is the final, most optimized version of the naive algorithm. It can be recognized that stage 2 does not result into an improvement on the AMD GPU when using Open SYCL. However, with DPC++ an improvement can be observed in some cases.

Figure 5.11 shows the results of the second experiment which was performed on an AMD Radeon PRO VII. Figure 5.11a shows the runtimes of the naive algorithm on all six datasets with the three different optimization stages using Open SYCL, whereas figure 5.11b shows these runtimes under the usage of DPC++. The y-axes corresponds to the runtime of the acceleration kernel and have been logarithmically scaled. The x-axes denote the number of bodies contained in the dataset used for the simulation.

In figure 5.11a it can be observed that with Open SYCL the optimization of stage two with local memory has a negative effect on the runtime of the acceleration kernel, especially for large datasets. For example, with the largest dataset the naive algorithm needs about 24.1s with stage zero and 26.1s with stage 2 when using Open SYCL. When using DPC++ however, it can be observed in figure 5.11b that stage two does improve the performance of the acceleration kernel. For the largest dataset stage two improves the runtime from 17.9s to only 14.5s when DPC++ gets used. The overall times with DPC++ are also much faster. This will be further analyzed in chapter 5.4.4. Similarly as on the NVIDIA GPUs stage one often improves the performance when using DPC++ but has almost no effect on the runtimes of Open SYCL. This could be due to a similar reason as on NVIDIA GPUs, however no profiling was done to further analyze this observation on AMD GPUs. Determining why stage three does not improve the naive algorithm on AMD GPUs with

Open SYCL and even results into worse performance would also require further research. However, since stage three improves performance with DPC++ it shows that this optimization approach can also improve performance on AMD GPUs and is not limited to NVIDIA GPUs.

Impact on the runtime on CPUs

Since the optimization approach by Nyland et al. was initially thought for GPUs, the question arises if this results into a performance penalty on CPUs. The experiment presented in this section aims to analyze this by measuring the runtimes of the two optimized and the non-optimized stage of the naive algorithm on a dual socket AMD EPYC 7543 with a total of 64 cores. For stage one and two the work-group size was chosen as 128.

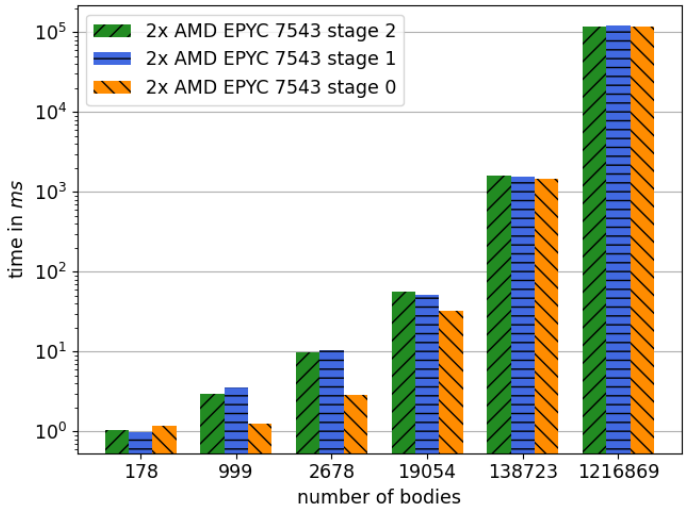
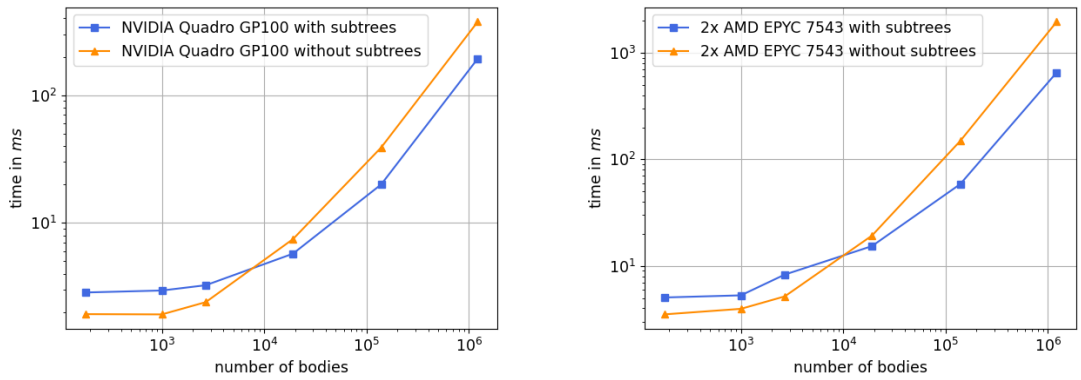


Figure 5.12: Impact of the performance optimizations for the naive algorithm on a dual socket AMD EPYC 7543 with a total of 64 cores The x-axis denotes the number of bodies. The y-axis is logarithmically scaled and corresponds to the runtime. The colors of the bars correspond to the different stages of the naive algorithm. Stage 0 is the non optimized version of the naive algorithm and stage 2 is the final, most optimized version of the naive algorithm.

Figure 5.12 shows the results of this experiment for all three stages of the naive algorithm on all six datasets. For the small datasets the optimized stages of the naive algorithm are slower than the non-optimized stage. Part of the reason for this could be the work-group size of 128 which is not optimal for small datasets and performs better for large numbers of bodies. However, it is hard to find a work-group size that works well with all datasets. Since the non-optimized version performs better for small datasets it is probably best to not specify work-group sizes at all for CPUs if one does not have to do it and uses only small datasets. However, for the largest dataset, there is a small improvement from 118.7s for one time step with stage zero to 117.2s with stage two.

5.3.2 Impact of using subtrees for more parallelism in the Barnes-Hut algorithm

The experiments described in this section aim to analyze the impact of using subtrees for more parallelism on CPUs and GPUs. To do this, the runtime of the octree creation was measured on an NVIDIA Quadro GP100 GPU and on a dual socket AMD EPYC 7543 CPU once with the usage of subtrees and once with the approach that does not use subtrees and is thus limited to one work-group.



(a) Impact of the performance optimization on an NVIDIA Quadro GP100

(b) Impact of the performance optimizations on a dual socket AMD EPYC 7543

Figure 5.13: Impact of using subtrees during the octree creation of the Barnes-Hut algorithm on CPUs and GPUs. Figure 5.13a shows the time needed for the octree creation with and without subtrees on an NVIDIA Quadro GP100 GPU. Figure 5.13b shows the results of the same experiment on a dual socket AMD EPYC 7543. It can be observed that using the subtrees during the octree creation results into better performance for larger datasets.

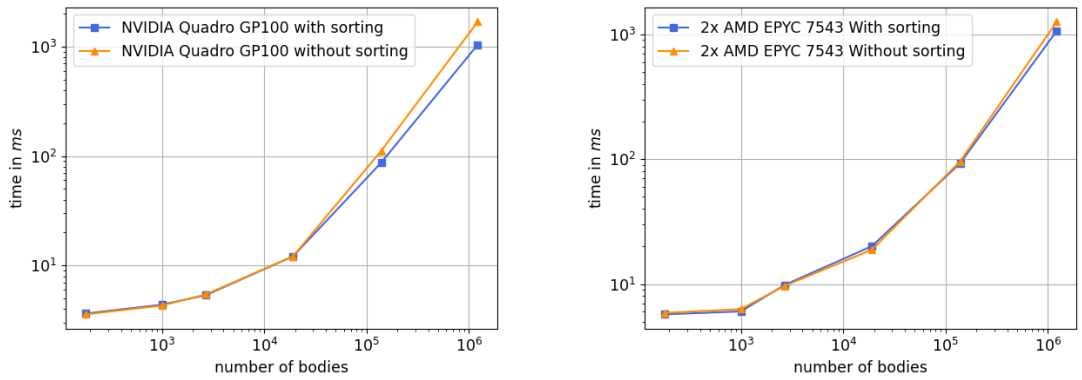
Figure 5.13 shows the results of this experiment. The x-axes correspond to the number of bodies used for the simulation. The y-axes shows the runtime of the octree creation in milliseconds. Both axes are logarithmically scaled. In figure 5.13a the results of the experiment on an NVIDIA Quadro GP100 GPU are shown. It can be seen that for small datasets the approach without using subtrees is faster. This is likely due to the additional overhead introduced when using subtrees. With small datasets the performance gain is not large enough to make up for this overhead. For larger datasets however, one can see that the GPU can actually benefit from the additional amount of parallelism and the approach with subtrees speeds up the octree creation. For example for the largest dataset the runtime of the octree creation decreases from 375.8ms without using subtrees to about 193.1ms when subtrees get used.

Figure 5.13b shows the results of the experiment on a dual socket AMD EPYC 7543 CPU. Interestingly, the behavior on CPUs is similar to the behavior on GPUs. One can observe that for smaller datasets the approach without subtrees is faster than the approach that uses subtrees. When larger datasets get used, the approach with subtrees results into a lower runtime. When using the approach with subtrees on CPUs the runtime for the octree creation with the largest dataset decreases from about 1953.7ms to 653.9ms. The cross-over point where the approach that uses subtrees gets faster is similar to the one that can be observed on GPUs. It is interesting that the

approach with subtrees also increases the performance on CPUs since even when one is limited to one work-group, there should be more than enough parallelism to fully utilize the 64 cores of the dual socket AMD EPYC 7543. The reason for this behavior could be that the approach with subtrees decomposes the octree creation into several smaller sub-problems. This would allow for greater scheduling flexibility since more work-groups get used. Furthermore, since the individual subtrees are smaller, there has to be less synchronization between the individual work-items in one work-group. Another observation from the experiment shown in 5.3 is that the CPU favors a larger top of the octree and thus more smaller subtrees. This could support the theory that the CPU can handle many small subtree creations better than creating the whole tree at once.

5.3.3 Effect of sorting the bodies before the acceleration computation

The focus of the experiment described in this section is the impact of the additional sorting step in the Barnes-Hut algorithm. In the experiment the runtime of one time step (without the time integration) has been measured on an NVIDIA Quadro GP100 GPU and on a dual socket AMD EPYC 7543, once with the additional sorting step and once without the sorting step.



(a) Impact of the sorting step on an NVIDIA Quadro GP100

(b) Impact of the sorting step on a dual socket AMD EPYC 7543

Figure 5.14: Impact of the sorting step of the Barnes-Hut algorithm on CPUs and GPUs. Figure 5.14a shows the time needed for one time step with and without the additional sorting step on an NVIDIA Quadro GP100 GPU. Figure 5.14b shows the results of the same experiment on a dual socket AMD EPYC 7543. It can be observed that sorting the bodies before the acceleration computation of the Barnes-Hut algorithm results into better performance for larger datasets on both GPUs and CPUs. The impact however, is much more noticeable on GPUs.

Figure 5.14 shows the results of this experiment. The x-axes denote the number of bodies used for the simulation. The y-axes show the runtime for one time step. This means that the time for the sorting step is included in the timings. Both axes are logarithmically scaled. The results of the experiment conducted on the GPU are shown in 5.14a. One can see, that there is no improvement on small datasets. However, for larger datasets there is an improvement. When using the largest

dataset on the GPU the sorting step improves the runtime for one time step from approximately 1681.7ms to 1038.32ms. Thus, sorting the bodies according to the tree can have a positive impact on the performance of the Barnes Hut algorithm.

In figure 5.14b the results of the experiment on the CPU are shown. One can observe that the impact of the sorting step is much smaller on CPUs. However, for large datasets there is a small improvement of the overall runtime. For example for the largest dataset the additional sorting step decreases the runtime from about 1261.1ms to 1065.1 ms. This improvement is not as high as on GPUs, however it shows that sorting the bodies according to the tree can also increase the performance on CPUs. This observation could be explained with an increased probability of cache hits in the acceleration kernel since threads that belong to one work-group are more likely to step down into the tree in a similar way and thus access the data of the same nodes of the tree.

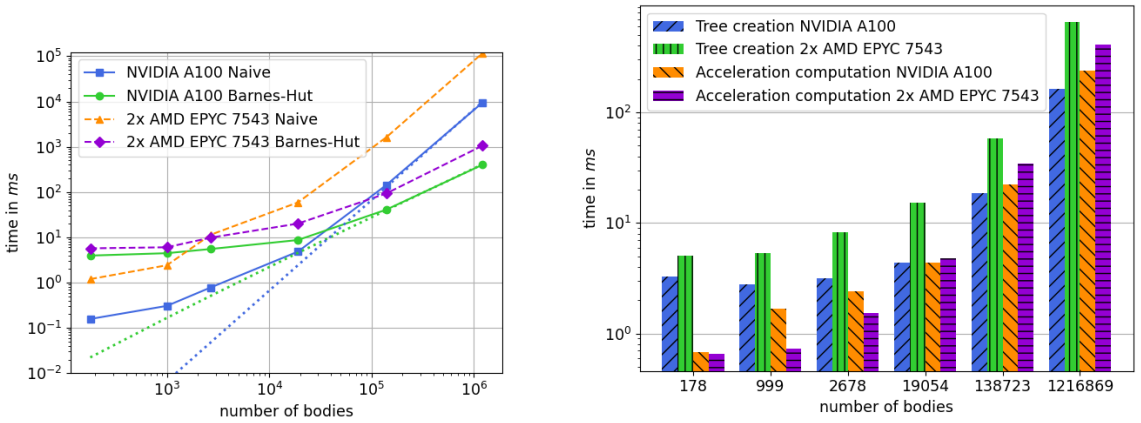
5.4 Comparison of the runtime behavior on different GPUs and CPUs

The goal of the experiments presented in this section is to compare the runtime behavior of the Barnes-Hut algorithm and the naive algorithm on different CPUs and GPUs. First the overall runtime behavior on GPUs and CPUs will be compared. After that the differences of the runtime behavior on consumer and data center GPUs will be analyzed. Next, the runtimes of both algorithms will be compared on NVIDIA and AMD GPUs, followed by an analysis of the differences in the runtime behavior when using DPC++ and Open SYCL. Finally, both algorithms will be compared on two different CPUs to analyze the impact of core count and clock speed on both algorithms.

5.4.1 Comparison of CPUs and GPUs

The experiment presented in this section aims to study the runtime behavior of the Barnes-Hut algorithm and the naive algorithm on CPUs and GPUs. For both algorithms runtimes were measured on an NVIDIA A100 GPU and on a dual socket AMD EPYC 7543 CPU using all datasets.

Figure 5.15a shows the runtimes of both algorithms on the CPU and GPU for all datasets. The x-axis corresponds to the number of bodies used for the simulation and the y-axis shows the time for one time step in milliseconds. Both axes are logarithmically scaled. The dotted lines correspond to the theoretical complexity of $O(n^2)$ for the naive algorithm and $O(n\ log(n))$ for the Barnes-Hut algorithm. It can bee sen that both algorithms reach their theoretical complexity for large datasets. When considering the runtimes of the naive algorithm on the CPU and on the GPU, one can observe that the GPU is clearly faster by a substantial margin. For the largest dataset the dual socket AMD EPYC 7543 CPU needs about 117.97s for one time step. The NVIDIA A100 GPU only requires about 9.6s. This can be seen as an example of how the highly parallel nature of the naive algorithm makes it very well suited for execution on GPUs. When considering the Barnes-Hut algorithm, one can observe that the NVIDIA A100 is still faster than the dual socket AMD EPYC 7543 CPU but the margin between the two has decreased a lot. With the largest dataset the GPU needs about 403.4ms for one time step. The CPU needs about 1065.2ms for the same amount of work.



(a) Comparison of the naive algorithm and the Barnes-Hut algorithm on CPUs and GPUs

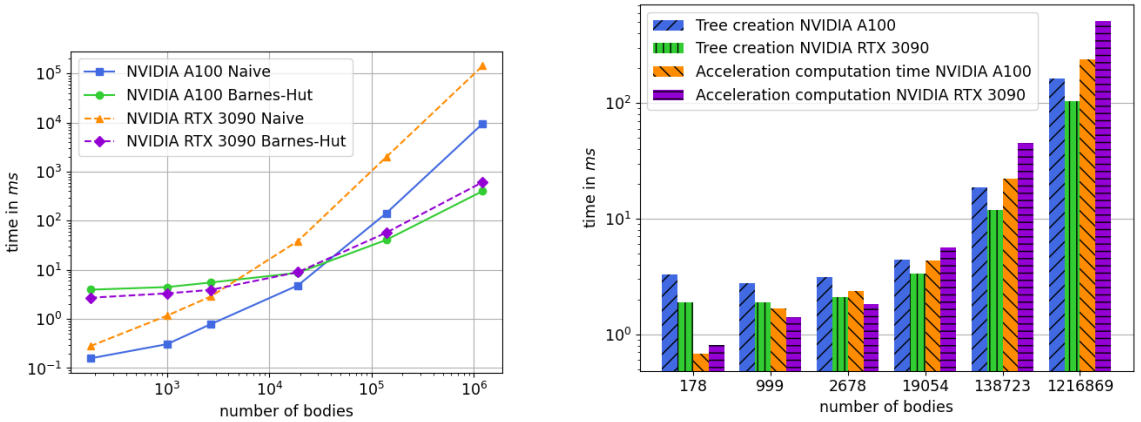
(b) Comparison of the time proportions of tree creation and acceleration computation

Figure 5.15: Comparison of the naive algorithm and the Barnes-Hut algorithm on an NVIDIA A100 GPU and on a dual socket AMD EPYC 7543 CPU. Figure 5.15a shows the runtimes of both algorithms on the GPU and the CPU. The dotted lines correspond to the theoretical complexity of $O(n^2)$ for the naive algorithm and $O(n\ log(n))$ for the Barnes-Hut algorithm. It can be observed that the GPU is faster for both algorithms. For the naive algorithm the difference between the runtimes on the CPU and GPU is larger than with the Barnes-Hut algorithm. For large datasets both algorithms reach their theoretical complexity. Figure 5.15b shows the runtimes of the Barnes-Hut algorithm in more detail. One can see that the GPU can build the octree faster than the CPU. The CPU, however, performs better during the acceleration computation of the Barnes-Hut algorithm.

Figure 5.15b shows the runtimes of the individual steps of the Barnes-Hut algorithm during this experiment. The x-axis shows the number of bodies used for the simulation. The y-axis is logarithmically scaled and shows the runtime in milliseconds. Each bar in a group of four bars corresponds to one of the two steps of the Barnes-Hut algorithm on the GPU or CPU: the octree creation and the acceleration computation. When building the octree the NVIDIA A100 GPU is faster than the AMD EPYC 7543 CPU. The GPU takes only about 163.4ms for this step on the largest dataset whereas the CPU completes this step in 653.9ms. This could be due to the fact that the approach chosen for the octree creation, which is based on the approach by Burtscher et al. [BP11], was initially designed for GPUs and not for CPUs. Especially the high amount of synchronization required for this approach could make it less ideal for CPUs. When comparing the runtime of the acceleration kernel, the NVIDIA A100 GPU completes this step in about 240ms whereas the AMD EPYC 7543 CPU needs about 411.3ms. Even though the GPU is still faster in this kernel, the CPU can keep up better. In summary, one can conclude that for the hardware considered during this experiment both algorithms perform better on the GPU. However, the difference between the runtimes on the CPU and GPU is much smaller for the Barnes-Hut algorithm.

5.4.2 Differences between consumer and data center GPUs

The objective of the experiment presented in this section is to compare the runtime behavior of the naive algorithm and the Barnes-Hut algorithm between consumer and data center GPUs. For the comparison an NVIDIA A100 will be used as the data center GPU. It will be compared to an NVIDIA RTX 3090 consumer GPU. Generally, data center GPUs have a lot more double precision performance than consumer GPUs. This can also be seen in table 5.1 where the NVIDIA A100 has about 9.7 TFLOPS of double precision performance, and the NVIDIA RTX 3090 has only about 0.556 TFLOPS.



(a) Comparison of the naive algorithm and the Barnes-Hut algorithm on a consumer and a data center GPU

(b) Comparison of the time proportions of the tree creation and the acceleration computation in the Barnes-Hut algorithm

Figure 5.16: Comparison of the naive algorithm and the Barnes-Hut algorithm on an NVIDIA A100 data center GPU and an NVIDIA RTX 3090 consumer GPU. Figure 5.16a shows the runtimes of both algorithms on the data center and the consumer GPU. It can be observed that for larger datasets the data center GPU is faster for both algorithms. For the naive algorithm the difference between the runtimes on the consumer and the data center GPU is larger than with the Barnes-Hut algorithm. For smaller datasets the consumer GPU is even faster than the data center GPU. Figure 5.16b shows the runtimes of the Barnes-Hut algorithm in more detail. One can see that the consumer GPU can build the octree faster than the data center GPU. The data center GPU, however, performs better during the acceleration computation of the Barnes-Hut algorithm.

Figure 5.16a shows the results of the experiment. In this figure the runtimes of both algorithms on the consumer and the data center GPU are presented. The x-axis corresponds to the number of bodies used for the simulation and the y-axis shows the time for one time step in milliseconds. Both axes are logarithmically scaled. One can see that, similar to the behavior between CPUs and GPUs presented in 5.15a, the performance of the naive algorithm is much better on the data center GPU than on the consumer GPU. For the smallest dataset the difference between both GPUs is less noticeable. For larger datasets however, the data center GPU is considerably faster. The NVIDIA A100 needs about 9.56s for one time step with the largest dataset and the naive algorithm. The NVIDIA RTX 3090 on the other hand needs over 144 seconds in this situation. This behavior can be explained with the much higher double precision performance of the NVIDIA A100.

When looking at the runtimes of the Barnes-Hut algorithm on both GPUs, one can observe that both GPUs perform very similar. For small datasets the NVIDIA RTX 3090 consumer GPU performs slightly better than the NVIDIA A100 data center GPU. With the dataset containing 19054 bodies, both GPUs perform almost equally. For larger datasets the data center GPU performs better than the consumer GPU. For example, for the largest dataset the NVIDIA A100 needs about 403.4ms per time step and the NVIDIA RTX 3090 needs 614.1ms in the same situation. Here, the difference is much smaller than for the naive algorithm.

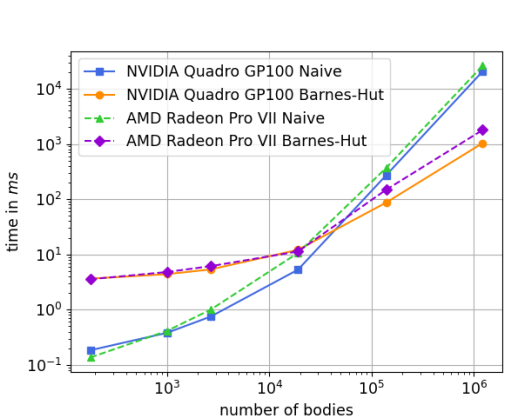
To explain this behavior one can take a look at figure 5.16b which shows the time proportions of the octree creation and the acceleration computation for the Barnes-Hut algorithm on both GPUs. The x-axis shows the number of bodies used for the simulation. The y-axis is logarithmically scaled and shows the runtime in milliseconds. It can be observed that the NVIDIA RTX 3090 can build the octree much faster than the NVIDIA A100. For the largest dataset it takes the consumer GPU only 104.2ms for one octree creation step whereas the data center GPU needs 163.4ms in this case. But the NVIDIA A100 performs much better during the acceleration computation as it only needs about 240ms for this phase. With about 509.9ms the NVIDIA RTX 3090 needs much longer for the acceleration kernel. The latter could be explained again with the much higher double precision performance of the NVIDIA A100 which plays an important role during the acceleration computation since most of the calculations that take place in this kernel use double precision. For the octree creation phase double precision performance does not play such an important role since this phase of the Barnes-Hut algorithm is dominated by memory operations, so the behavior during the octree creation likely has another cause. As it can be seen in table 5.1, both GPUs use different kinds of memory. The NVIDIA A100 uses HBM2 memory whereas the NVIDIA RTX 3090 uses GDDR6X memory. The bandwidth of HBM2 is much higher, than the one of GDDR6X (see [NVI20] and [NVI21]). However, even though there are a lot of memory operations during the octree creation, the overall amount of data that gets accessed by one work-item is not that large. Thus, the high bandwidth of HBM2 might not bring an advantage here. This factor could play a role in why the NVIDIA RTX 3090 is faster during the octree creation, however to fully explain this observation further research would have to be done.

In summary the data center GPU always offers substantially better performance for the naive algorithm due to its much higher double precision performance. However, the consumer GPU can keep up with the data center GPU for the Banes-Hut algorithm and even offers better performance for small datasets. Nevertheless, when using larger datasets the acceleration computation plays a more predominant role in the Barnes-Hut algorithm. As this happens the NVIDIA RTX 3090 can not use its advantage during the octree creation phase to fully compensate its lack of double precision performance that worsens the runtimes of the acceleration kernel anymore. However, as it is shown in figure 5.7a, when one would use higher θ-values, the amount of work for the acceleration computation gets reduced by a lot which could result into the a higher importance of the octree creation phase where the NVIDIA RTX 3090 has an advantage.

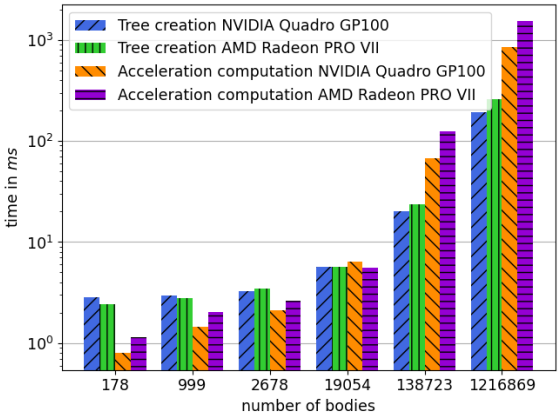
5.4.3 Differences between AMD and NVIDIA

The experiment that will be presented in this section will study the differences in the runtime behavior of the naive algorithm and the Barnes-Hut algorithm on GPUs from NVIDIA and AMD. During the experiment runtimes of both algorithms have been measured on an NVIDIA Quadro GP100 and on an AMD Radeon PRO VII. The two GPUs have been chosen for the comparison

since they can both be considered as data center GPUs. Furthermore, as it can be seen in table 5.1, they have a similar theoretical performance of about 5 TFLOPS for the NVIDIA Quadro GP100 and 6.5 TFLOPS for the AMD Radeon PRO VII. Thus, in theory, the AMD GPU should be slightly faster.



(a) Comparison of the naive algorithm and the Barnes-Hut algorithm on GPUs from AMD and NVIDIA



(b) Comparison of the time proportions of tree creation and acceleration computation in the Barnes-Hut algorithm

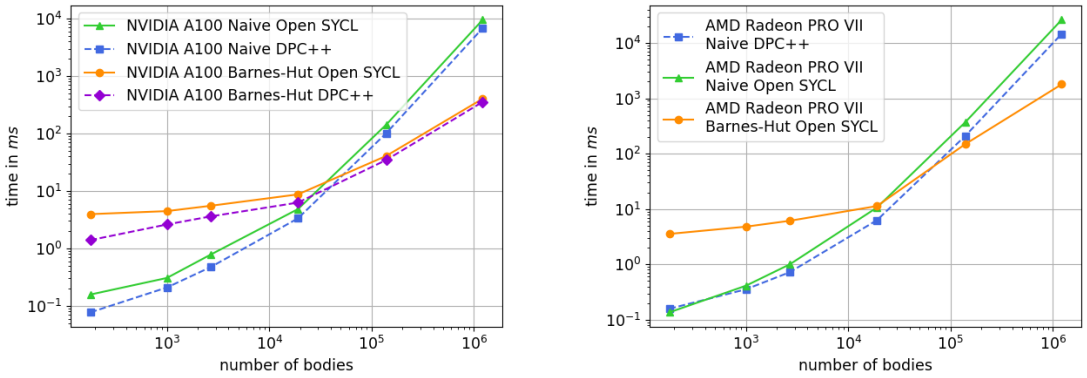
Figure 5.17: Comparison of the naive algorithm and the Barnes-Hut algorithm on an NVIDIA Quadro GP100 GPU and on an AMD Radeon Pro VII GPU. Figure 5.17a shows the runtimes of both algorithms on the NVIDIA and AMD GPU. It can be observed that both GPUs have a similar runtime behavior for both algorithms. The NVIDIA GPU is faster when larger datasets get used. Figure 5.17b shows the runtimes of the Barnes-Hut algorithm in more detail. One can see that the time proportions of the octree creation and the acceleration computation are similar for the NVIDIA and AMD GPU. Since both GPUs are data center GPUs this is also what one would expect.

The results of the experiment are shown in figure 5.17a. It can be observed that both GPUs have a similar runtime behavior for both algorithms. For larger datasets the NVIDIA GPU is faster than the AMD GPU. For example, the NVIDIA GPU needs only 20.64s for the largest dataset when using the naive algorithm. With 26.1 seconds the AMD GPU needs much longer. If one looks at the theoretical performance one would expect that the AMD GPU performs better, especially with the naive algorithm since it has a higher double precision performance. This observation can be explained with the fact that Open SYCL got used for this experiment. As it will be shown in section 5.4.4, DPC++ performs much better on AMD GPUs than Open SYCL. When one would use DPC++ in the example above the runtime of the naive algorithm on the NVIDIA GPU would be reduced to 15.6s. The runtime on the AMD GPU with DPC++ is 14.5s which is lower than the runtime on the NVIDIA GPU as one would expect from the theoretical performance. Figure 5.17b shows the time proportions of the two different steps in the Barnes-Hut algorithm for the NVIDIA Quadro GP100 and the AMD Radeon Pro VII. One can see that the time proportions of the octree creation and the acceleration computation do not differ a lot between the AMD and the NVIDIA GPU. Since both GPUs are data center GPUs with similar double precision performance and memory configuration one would expect this behavior.

Even though the theoretically weaker NVIDIA Quadro GP100 performs better for both algorithms with most datasets than the AMD Radeon Pro VII when Open SYCL gets used the overall runtime behavior of both algorithms is similar. When DPC++ gets used the results are more like one would expect for the naive algorithm. One can conclude that, albeit a few compromises and problems with specific SYCL implementations, performance portability can be achieved between GPUs from NVIDIA and AMD when using SYCL.

5.4.4 Comparison of DPC++ and Open SYCL on different GPUs

The experiments described in this section aim to analyze the performance differences between the two different SYCL implementations considered in this work: DPC++ and Open SYCL. In the first experiment the performance of the two SYCL implementations will be compared on an NVIDIA A100 GPU. The second experiment analyzes the performance on an AMD Radeon Pro VII GPU.



(a) Comparison of DPC++ and Open SYCL on an NVIDIA A100 GPU

(b) Comparison of DPC++ and Open SYCL on an AMD Radeon PRO VII GPU

Figure 5.18: Comparison of two different SYCL implementations on an NVIDIA A100 GPU and on an AMD Radeon PRO VII GPU. Figure 5.18a shows the runtimes of both algorithms once with DPC++ and once when using Open SYCL on an NVIDIA A100 GPU. Figure 5.18b shows the runtimes of the naive algorithm with DPC++ and Open SYCL on an AMD Radeon PRO VII GPU. One can see that DPC++ performs better in almost all situations. On the AMD GPU the difference between the two implementations is much bigger than on the NVIDIA GPU.

Figure 5.18a shows the results of the first experiment which was conducted on the NVIDIA A100 GPU. The x-axis corresponds to the number of bodies used during the simulation and the y-axis shows the runtime of one time step in milliseconds. Both axes have been logarithmically scaled. One can see that DPC++ performs better across all datasets. For the largest data set one time step of the naive algorithm takes about 9.56s with Open SYCL whereas DPC++ only needs about 6.83s. When looking at the differences for the Barnes-Hut algorithm on can see that DPC++ performs faster again. Open SYCL needs about 403.4ms for one time step of the Barnes-Hut algorithm and DPC++ needs only 355.5ms.

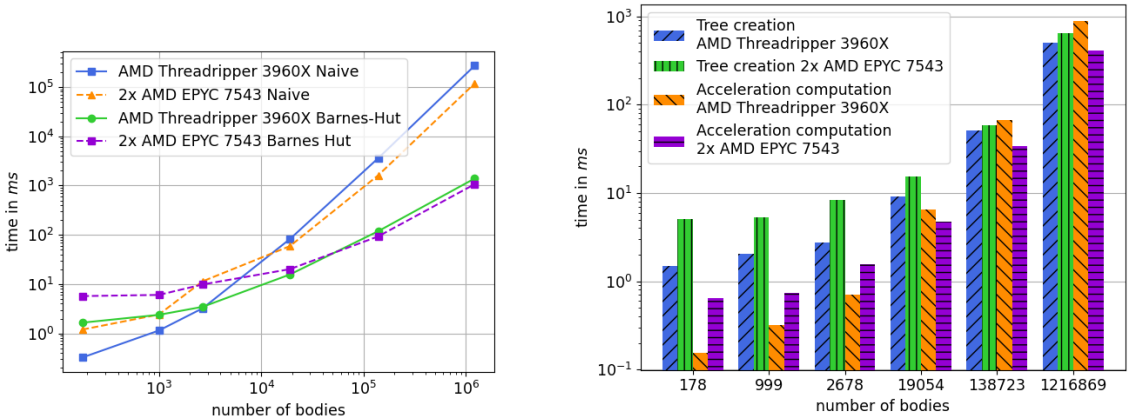
Figure 5.18b shows the results of the second experiment on the AMD Radeon PRO VII GPU. The x-axis shows the number of bodies used during the simulation and the y-axis denotes the runtime of one time step in milliseconds. Both axes have been logarithmically scaled. The experiment only considers the naive algorithm on the AMD GPU since the Barnes-Hut algorithm did not run on the AMD GPU with the version of DPC++ considered for all of the experiments. The implementation of this algorithm uses more advanced features like atomics and synchronization. It could be that this leads to problems since with DPC++ the support of the HIP backend on AMD GPUs is still experimental. However, as shown in section 5.4.3 with Open SYCL the algorithm works on this GPU. The runtimes for the naive algorithm on the AMD GPU are very different for Open SYCL and DPC++. For the largest dataset the naive algorithm needs about 26.19s when using Open SYCL. With DPC++ the runtime decreases substantially to only 14.57s. The fact that Open SYCL struggles on AMD hardware can also be observed in the second experiment presented in section 5.3.1 where Open SYCL yields worse results for the performance optimizations of the naive algorithm on the AMD GPU.

In summary, one can conclude that DPC++ mostly offers better performance than Open SYCL. On AMD GPUs DPC++ is substantially faster than Open SYCL when using the naive algorithm. However, the Barnes-Hut algorithm did not work on the AMD GPU when using DPC++ but works with Open SYCL.

5.4.5 Comparison of CPUs with different properties

The experiment presented in this sections investigates how the runtime behavior of both algorithms is affected when using CPUs with different characteristics. For the experiments the runtimes of both algorithms will be measured on an AMD Threadripper 3960X and on a dual socket AMD EPYC 7543. This comparison can be interesting since, as it can be seen in table 5.1, with 64 the AMD EPYC CPU has a much higher core count than the AMD Threadripper with only 24 cores. The AMD Threadripper, however, has a much higher clock speed of 4.5GHz compared to only 3.7GHz of the AMD EPYC CPU.

Figure 5.19a presents the results of this experiment. The x-axis corresponds to the number of bodies used during the simulation and the y-axis shows the runtime of one time step in milliseconds. Both axes have been logarithmically scaled. It can be observed that the AMD Threadripper CPU performs better on smaller datasets for both algorithms. For larger datasets the AMD EPYC CPU performs better. For the largest dataset the difference between the two CPUs is much higher for the naive algorithm than for the Barnes-Hut algorithm. Here, the AMD EPYC needs 117.97s for one time step of the naive algorithm and the AMD Threadripper needs about 271.26s. When considering the runtimes of the Barnes-Hut algorithm for the largest dataset, the AMD EPYC needs about 1.06s for one time step and the AMD Threadripper needs 1.39s. The behavior observed with the naive algorithm could be explained with the fact that the AMD EPYC processor has a clear advantage in absolute core count and the AMD Threadripper can not make up for this with its higher clock speed anymore. In order to explain the runtime behavior of the Barnes-Hut algorithm on both CPUs one can take a look at figure 5.19b which shows the runtimes of the individual steps of the Barnes-Hut algorithm in more detail. One can see that the AMD Threadripper 3960X is faster during the octree creation. For the largest dataset it only needs 506.5ms for one octree creation whereas the AMD EPYC needs 653.9ms. This could indicate that this step of the Barnes-Hut algorithm benefits from a higher clock speed. Furthermore, the AMD EPYC CPU might not fully



(a) Comparison of the naive algorithm and the Barnes-Hut algorithm on two different CPUs

(b) Comparison of the time proportions of tree creation and acceleration computation in the Barnes-Hut algorithm

Figure 5.19: Comparison of the naive algorithm and the Barnes-Hut algorithm on an AMD Threadripper 3960X and a dual socket AMD EPYC 7543 CPU. Figure 5.19a shows the runtimes of both algorithms on the two different CPUs. It can be observed that the AMD Threadripper performs better for smaller datasets with both algorithms. The AMD EPYC processor can benefit from its higher core count when larger datasets are used and performs better than the AMD Threadripper in these situations. Figure 5.19b shows the runtimes of the Barnes-Hut algorithm in more detail. One can see that for the tree creation the AMD Threadripper 3960X performs much better than the dual socket AMD EPYC 7543. However, the AMD EPYC processor performs much better during the acceleration computation.

benefit from its higher core count in this part of the algorithm, since the additional parallelism could also result in additional overhead for the synchronization. However, the AMD EPYC processor performs much better during the acceleration computation. In this phase, the dual socket AMD EPYC 7543 needs only about 411.3ms whereas the AMD Threadripper 3960X needs 884.1ms. This is likely because in this phase of the algorithm the dual socket AMD EPYC 7543 can fully benefit from its higher core count compared to the AMD Treadripper 3960X. For the smallest four datasets the Threadripper can use its advantage during the octree creation to make up for the time it loses to the AMD EPYC during the acceleration computation. For all larger datasets the time needed for the acceleration computation becomes more and more predominant and eventually the AMD EPYC CPU performs better for the Barnes-Hut algorithm.

One can conclude that eventually a higher CPU core count results into better performance than a higher CPU clock speed and fewer cores for both algorithms. However especially for the Barnes-Hut algorithm one can observe that with the tree creation phase there might be a situation where fewer cores and a higher clock speed might have an advantage over a higher core count. This observation is especially important when one uses higher θ-values which reduce the amount of work for the acceleration computation.

5.5 Summary of the results

This section will summarize the findings from this chapter. The experiments presented in the last section only showed the differences between the runtimes of the naive algorithm and the Barnes-Hut algorithm for one time step. However, n-body simulations can consist of many time steps. Thus, even small differences can add up over time and can result into non-negligible differences for the overall runtime. In order to demonstrate this with the runtimes measured in the previous experiments, these runtimes are used for a projection of the runtime of a longer simulation. If one would simulate approximately one earth year with a Δt of one hour, the simulation would consist of $24 \cdot 365 = 8760$ time steps. Including the computation of the initial accelerations this would result into 8761 acceleration computations that have to be performed by each of the algorithms. For the projection of the overall runtime of a simulation of approximately one earth year the runtimes of the final experiments with the largest data set containing 1216869 bodies from section 5.4 have been multiplied with 8761. This means that the time for the leapfrog integration is excluded in this projection. However, this step does not contribute a lot to the overall runtime of the simulation and is the same for the naive algorithm and the Barnes-Hut algorithm.

Device	Naive algorithm	Barnes-Hut algorithm
NVIDIA A100	23.26h	58.9min
NVIDIA A100 (DPC++)	16.63h	51.9min
NVIDIA Quadro GP100	50.24h	2.52h
NVIDIA RTX 3090	14.6d	1.49h
AMD Radeon PRO VII	63.7h	4.36h
AMD Radeon PRO VII (DPC++)	35.3h	-
Dual socket AMD EPYC 7543	11.96d	2.59h
AMD Threadripper 3960X	27.5d	3.38h

Table 5.2: Projection of the runtime for a simulation of one earth year based on the measured runtimes of one time step. Projections based on the measurements when using DPC++ are marked explicitly. All other projections are based on the runtimes with Open SYCL.

Table 5.2 show the results of the projection. If not stated otherwise the projected runtimes are based on the measurements with Open SYCL. It can be realized that only data center GPUs would finish the simulation with the naive algorithm in an acceptable time frame. CPUs and consumer GPUs need much more than a week for a simulation that an NVIDIA A100 could finish in under one day according to the projection. For the Barnes-Hut algorithm the differences are less drastic. Even though the differences amount to several hours in some cases, this not as extreme as for the naive algorithm. Also in this projection the NVIDIA RTX 3090 consumer GPU can keep up quite well with an NVIDIA A100. When looking at the differences between the two SYCL implementations considered in this work, it stands out that with the naive algorithm DPC++ would be several hours faster than Open SYCL for this simulation. With the Barnes-Hut algorithm the difference is only about seven minutes on the NVIDIA GPU. As explained before, the Barnes-Hut algorithm did not run on the AMD GPU with DPC++. Thus, there is no projected time in this case.

In summary, it can be seen that data center GPUs are the only hardware that would finish the simulation with the naive algorithm in an acceptable time period. The Barnes-Hut algorithm also performs best on GPUs, nevertheless, CPUs can keep up better than with the naive algorithm.

6 Conclusion

In this thesis, two different n-body algorithms, the naive algorithm and the Barnes-Hut algorithm were implemented using SYCL. The implementation is available on GitHub[1]. Both algorithms were compared on GPUs from NVIDIA and AMD as well as on different CPUs. The results show that GPUs can provide better performance for both algorithms. When considering longer simulations with several thousand time steps, a projection based on the runtimes measured on different devices shows that data center GPUs like the NVIDIA A100 are the only device type that would finish such a simulation in an acceptable time frame. CPUs and consumer GPUs would sometimes take several weeks to complete the simulation. For the Barnes-Hut algorithm the NVIDIA A100 also yields the best results. However, consumer GPUs like the NVIDIA RTX 3090 can keep up much better here than with the naive algorithm. When looking more deeply into the causes for this observation, it turned out that consumer GPUs can build the octree much faster than data center GPUs. Data center GPUs on the other hand have an advantage during the acceleration computation with large datasets due to their much higher double precision performance.

When studying CPUs more closely, similar differences between different types of CPUs have been found. CPUs with higher core counts but lower clock speeds perform much better with the naive algorithm than a CPU that has fewer cores but a higher clock speed. With the Barnes-Hut algorithm, CPUs with fewer cores and higher clock speeds can keep up much better since they can use their advantage during the octree creation to make up for the time they loose during the acceleration computation. Nevertheless, the CPU with the higher core count was still faster for large datasets in the end.

This work also compared the performance of both algorithms with two different SYCL implementations: Open SYCL and DPC++. The results show that DPC++ mostly offers better performance than Open SYCL. The differences were especially noticeable with the naive algorithm on AMD GPUs where DPC++ achieved considerably faster results. However, the implementation of the Barnes-Hut algorithm did not work the AMD GPU and DPC++ but worked with Open SYCL on this device.

The results show that it is important to understand the runtime behavior of the algorithms on different hardware. Even though the NVIDIA A100, which is also the theoretically best performing GPU used for the comparisons, offered the overall best results other GPUs like the NVIDIA RTX 3090 can keep up with the NVIDIA A100 in the Barnes-Hut algorithm. Considering the price difference between those two graphics cards, the NVIDIA RTX 3090 might be a better choice when only targeting the Barnes-Hut algorithm.

[1] https://github.com/TimThuering/N-Body-Simulation (visited on 04/03/2023)

6.1 Future work

There are several interesting research areas that could not be covered in this thesis and thus would be possible directions for future work. The octree of the Barnes-Hut algorithm gets constructed top-down in the implementation conducted with SYCL during this work. However, there are different approaches that construct an octree bottom-up like, for example, the work by Alpay [Alp19b], [Alp19a]. It would be interesting to implement such an algorithm with SYCL for the octree creation of the Barnes-Hut algorithm and compare it to the implementations performed during this thesis. Furthermore, one could investigate how algorithms with a lower theoretical complexity like the fast multipole method perform on different hardware when using SYCL. For the execution on CPUs only Open SYCL was used during this thesis. DPC++ also offers support for execution on CPUs. Future work could investigate how this backend performs compared to the OpenMP backend of Open SYCL with the n-body algorithms implemented during the course of this thesis.

Bibliography

[AH20] A. Alpay, V. Heuveline. “SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL”. In: *Proceedings of the International Workshop on OpenCL*. IWOCL ’20. New York, NY, USA: Association for Computing Machinery, Apr. 27, 2020, p. 1. DOI: 10.1145/3388333.3388658 (cit. on p. 20).

[Alp19a] A. Alpay. *SpatialCL - a high performance library for the spatial processing of particles on GPUs*. 2019. URL: https://github.com/illuhad/SpatialCL (cit. on pp. 13, 62).

[Alp19b] A. Alpay. *Teralens - A parallel (quasar) microlensing code for multi-teraflop devices*. 2019. URL: https://github.com/illuhad/teralens (cit. on pp. 13, 62).

[AMD] AMD. *AMD Ryzen™ Threadripper™ 3960X*. URL: https://www.amd.com/en/product/8946 (cit. on p. 34).

[AMD20a] AMD. *AMD EPYC™ 7002 Series Processors: A New Standard for the Modern Data Center*. 2020. URL: https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf (cit. on p. 34).

[AMD20b] AMD. *Develop & test workloads intended for largescale AMD Radeon Instinct™ deployments*. 2020. URL: https://www.amd.com/system/files/documents/hpc-industry-solution-brief-radeon-pro-vii.pdf (cit. on p. 34).

[AMD22] AMD. *AMD EPYC™ 7003 SERIES PROCESSORS*. 2022. URL: https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf (cit. on p. 34).

[ASV09] N. Arora, A. Shringarpure, R. W. Vuduc. “Direct N-body Kernels for Multicore Platforms”. In: *2009 International Conference on Parallel Processing*. 2009 International Conference on Parallel Processing. Sept. 2009, pp. 379–387. DOI: 10.1109/ICPP.2009.71 (cit. on p. 13).

[BG91] E. Bertschinger, J. M. Gelb. “Cosmological N-Body Simulations”. In: *Computers in Physics* 5.2 (Mar. 1991), pp. 164–179. DOI: 10.1063/1.4822978 (cit. on p. 16).

[BH86] J. Barnes, P. Hut. “A hierarchical O(N log N) force-calculation algorithm”. In: *Nature* 324.6096 (Dec. 1986), pp. 446–449. ISSN: 1476-4687. DOI: 10.1038/324446a0. URL: https://www.nature.com/articles/324446a0 (cit. on pp. 11, 15–17, 25).

[BP11] M. Burtscher, K. Pingali. “Chapter 6 - An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm”. In: ed. by W.-m. W. Hwu. Applications of GPU Computing Series. Boston: Morgan Kaufmann, Jan. 1, 2011, pp. 75–92. ISBN: 9780123849885. DOI: 10.1016/B978-0-12-384988-5.00006-1. URL: https://iss.oden.utexas.edu/Publications/Papers/burtscher11.pdf (cit. on pp. 13, 25, 29, 53).

[CS13] R. Capuzzo-Dolcetta, M. Spera. “A performance comparison of different graphics processing units running direct N-body simulations”. In: *Computer Physics Communications* 184.11 (Nov. 1, 2013), pp. 2528–2539. DOI: 10.1016/j.cpc.2013.07.005 (cit. on p. 13).

[EH86] D. J. Evans, W. G. Hoover. “Flows Far From Equilibrium Via Molecular Dynamics”. In: *Annual Review of Fluid Mechanics* (1986). URL: https://doi.org/10.1146/annurev.fl.18.010186.001331 (cit. on p. 15).

[Gre87] V. R. Greengard Leslie. “A Fast Algorithm for Particle Simulations”. In: *Journal of computational physics* 73 (1987), pp. 315–348 (cit. on pp. 13, 16).

[GSK+10] P. Gibbon, R. Speck, A. Karmakar, L. Arnold, W. Frings, B. Berberich, D. Reiter, M. Mašek. “Progress in Mesh-Free Plasma Simulation With Parallel Tree Codes”. In: *IEEE Transactions on Plasma Science* 38.9 (Sept. 2010), pp. 2367–2376. DOI: 10.1109/TPS.2010.2055165 (cit. on p. 15).

[HLW03] E. Hairer, C. Lubich, G. Wanner. “Geometric numerical integration illustrated by the Störmer–Verlet method”. In: *Acta Numerica* 12 (May 2003), pp. 399–450. DOI: 10.1017/S0962492902000144 (cit. on p. 16).

[Inta] Intel. *Intel® Core™ i7-6700K Prozessor*. URL: https://www.intel.de/content/www/de/de/products/sku/88195/intel-core-i76700k-processor-8m-cache-up-to-4-20-ghz/specifications.html (cit. on p. 34).

[Intb] Intel. *Intel® Xeon® Silver 4116 Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/120481/intel-xeon-silver-4116-processor-16-5m-cache-2-10-ghz.html (cit. on p. 34).

[Ken17] D. C. O. Kenneth R. Meyer. *Introduction to Hamiltonian Dynamical Systems and the N-Body Problem*. Springer Cham, 2017. Chap. 3. Celestial Mechanics. DOI: https://doi.org/10.1007/978-3-319-53691-0 (cit. on p. 15).

[Khr] Khronos-Group. *SYCL - C++ Single-source Heterogeneous Programming for Acceleration Offload*. URL: https://www.khronos.org/sycl/ (cit. on p. 19).

[Khr20] Khronos®-SYCL™-Working-Group. *SYCL™ Specification*. Ed. by R. Keryell, M. Rovatsou, L. Howes. Apr. 27, 2020. URL: https://registry.khronos.org/SYCL/specs/sycl-1.2.1.pdf (cit. on p. 26).

[Khr22] Khronos®-SYCL™-Working-Group. *SYCL™ 2020 Specification (revision 6)*. Ed. by M. Rovatsou, L. Howes, R. Keryel. Nov. 13, 2022. URL: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html (cit. on pp. 26, 29).

[NHP07] L. Nyland, M. Harris, J. Prins. *GPU Gems 3*. 2007. Chap. Chapter 31. Fast N-Body Simulation with CUDA, pp. 62–66. URL: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda (cit. on pp. 13, 16, 23, 24).

[NVI17] NVIDIA. *NVIDIA Powers New Class of Supercomputing Workstations with Breakthrough Capabilities for Design and Engineering*. Feb. 5, 2017. URL: https://nvidianews.nvidia.com/news/nvidia-powers-new-class-of-supercomputing-workstations-with-breakthrough-capabilities-for-design-and-engineering (cit. on p. 34).

[NVI20] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (cit. on pp. 34, 55).

[NVI21] NVIDIA. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. 2021. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (cit. on pp. 34, 55).

[RAB+21] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, X. Tian. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL*. Springer Nature, 2021. DOI: https://doi.org/10.1007/978-1-4842-5574-2 (cit. on p. 20).

[SRJ+22] S. Slattery, S. Reeve, C. Junghans, D. Lebrun-Grandié, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, A. Isner, K. Chong, S. Moore, T. Germann, J. Belak, S. Mniszewski. "Cabana: A Performance Portable Library for Particle-Based Simulations". In: *Journal of Open Source Software* 7 (Apr. 10, 2022), p. 4115. DOI: 10.21105/joss.04115 (cit. on p. 13).

[SW94] J. K. Salmon, M. S. Warren. "Fast Parallel Tree Codes for Gravitational and Fluid Dynamical N-Body Problems". In: *The International Journal of Supercomputer Applications and High Performance Computing* 8.2 (June 1, 1994), pp. 129–142. DOI: 10.1177/109434209400800205 (cit. on p. 15).

[TLA+22] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, J. Wilke. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283 (cit. on p. 13).

[WS93] M. S. Warren, J. K. Salmon. "A parallel hashed Oct-Tree N-body algorithm". In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. Supercomputing '93. New York, NY, USA: Association for Computing Machinery, Dec. 1, 1993, pp. 12–21. DOI: 10.1145/169627.169640 (cit. on p. 13).

[YB11] R. Yokota, L. Barba. "Treecode and Fast Multipole Method for N-Body Simulation with CUDA". In: *GPU Computing Gems Emerald Edition* (2011), pp. 113–132. DOI: 10.1016/B978-0-12-384988-5.00009-7 (cit. on p. 13).

All links were last followed on April 03, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature