Karlsruhe Institute of Technology

Probabilistic Object Tracking on the GPU

Bachelor Thesis of

Claudia Pfreundt

Department of Informatics
High Performance Humanoid Technologies Lab (H^2T)

Referees:	Prof. Dr.-Ing. Tamim Asfour
	Prof. Dr. Stefan Schaal
Advisors:	M.Sc. Manuel Wüthrich
	Dipl.-Inform. David Schiebener
	Dr. Jeannette Bohg

Duration: 1. December 2013 – 31. March 2014

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association www.kit.edu

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, March 31, 2014

(Claudia Pfreundt)

Contents

1 Introduction **1**

2 Related Work **3**
2.1 Particle filter on GPUs for real-time tracking (Montemayor et al., 2004) . . 3
2.2 A GPU-accelerated particle filter with pixel-level likelihood (Lenz et al., 2008) 4
2.3 Real-time visual tracker by stream processing (Lozano and Otsuka, 2009) . 5
2.4 6-DoF model-based tracking of arbitrarily shaped 3D objects (Azad et al., 2011) 6
2.5 RGB-D object tracking: A particle filter approach on GPU (Choi and Christensen, 2013) 6
2.6 Summary 7

3 Foundation **9**
3.1 Probabilistic object tracking 9
3.2 Parallelization 12

4 Pure OpenGL approach **14**
4.1 OpenGL 14
4.1.1 Rendering 14
4.1.2 Pipeline 17
4.2 Rendering a pose with OpenGL 21
4.3 Performance and bottlenecks 24

5 Combined approach with OpenGL and CUDA **25**
5.1 CUDA 26
5.1.1 Kernels 26
5.1.2 Memory structure 28
5.2 Parallel likelihood computation 30
5.3 Combined rendering with OpenGL 33
5.4 Performance and bottlenecks 34

6 Evaluation **35**
6.1 Work environment 35
6.2 Runtime analysis of the OpenGL approach 36
6.3 Runtime analysis of the combined OpenGL and CUDA approach 39
6.4 Scalability with the number of poses 44
6.5 Scalability with the resolution 48
6.6 Scalability with the number of triangles 49
6.7 Scalability with the number of cores 50

7 Conclusion **51**

Bibliography **52**

1. Introduction

Robots have been developed and improved for many years now, making them a self-evident part of our everyday life today. With the gradually increasing demand on robots, they are expected to work in unstructured environments, which makes a reliable interaction with this environment essential. Generally, sensors and actuators are required for this interaction, enabling the robot to gather information and adjust its behaviour to it. Common sensors include cameras which can deliver a realistic impression of the surroundings, or touch sensors which give feedback on the force applied to them. Actuators are physical parts that can translate signals into motion, making for example robot arms and hands able to move.

Many robotic tasks include grasping and manipulation of objects, for example when opening a door or carrying a tray. However, the position and orientation of an object need to be known in order to successfully interact with it, which is why object tracking algorithms have been developed. When the initial pose of an object is known and it is thereafter moved by the robot, forward kinematics can be used to determine the pose of the object. However, the precision of the actuators is not yet high enough to ensure a deterministic movement of the object. For example, it could be moved several centimeters further than expected, or it could slip in the robot hand, changing its orientation. The movement could also be intercepted by an obstacle or human intervention, which would preclude a reliable pose estimation of the object in question.
This is why object pose estimation through visual feedback is an important requirement for reliable robotic tasks. Optimally, it enables the robot to precisely track an object of interest and interact with it in real-time. While several approaches to this problem are being pursued, a general increase in precision and speed of the existing algorithms is desirable, as most approaches cannot yet robustly perform in real-time.

This thesis concentrates on optimizing the performance of an existing object tracking algorithm ([WPK+13]) which is solely based on depth measurements of the object. It can handle arbitrarily shaped objects since it does not depend on detectable edges or features. Additionally, it performs completely independently from the color or texture of an object, as this data is not used to determine the pose. This also makes the algorithm robust to illumination changes in the environment. Furthermore, it can handle clutter in the background and partial occlusion of the object.
A camera that delivers a depth image of the environment, for example a Microsoft Kinect, as well as a 3D model of the object are required to use the approach described.

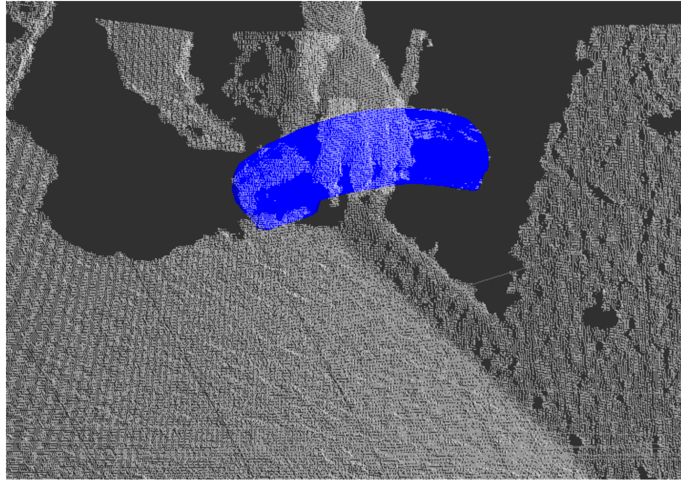

Figure 1.0.1: Example tracking of an object

The algorithm is based on a particle filter ([IB98]) that can track an object with 200 particles at 30 Hz using one CPU core. However, a downsampled resolution of 80 x 60 of the camera image is required to yield this performance, which can result in a loss of precision. Additionally, the speed at which the object can be moved is limited.
In order to improve the accuracy of the filter, either more particles have to be evaluated or the resolution of the image has to be increased. A higher speed can be achieved by evaluating more particles as well, or using a camera which can deliver depth measurements at a higher frame rate. To realize any of these aspects, the computations of this algorithm have to be accelerated.

To enable these improvements, this thesis explains and conducts a parallelization of the given algorithm. Parallelization itself can be done for multi-core CPUs, GPUs or clusters thereof. While CPU-parallelization is commonly used for general purpose applications or such with high data transfer needs, GPU-parallelization is typically applied to graphics related applications or such that use data parallelism. CPU-, GPU- or hybrid-clusters are employed in High Performance Computing to solve computationally intense scientific problems like medical imaging or seismic data analysis. They offer parallelism at a much larger scale, but are much more expensive and require a more complex code development. As a particle filter employs a lot of data-parallelism and, in our case, rendering of 3D models, a GPU-parallelization is the most promising approach. Using two different frameworks, OpenGL (see Section 4.1) and CUDA (see Section 5.1), a very efficient parallelization could be achieved, enabling the evaluation of about 10,000 particles at a rate of 30 Hz with the aforementioned resolution (see Section 6.4). Likewise, the resolution can be increased when using less particles, e.g. 200 for a resolution of 640 x 480.

The next chapter gives an overview of related work in the field, while the third chapter explains the algorithm that is parallelized and the basic concepts of parallelization. Subsequently, a naive and a more sophisticated approach to the GPU-parallelization of the algorithm are explained, including a description of the respective frameworks used. A short summary of the performance of the algorithms can be found at the end of the respective chapters, while a detailed analysis of the runtimes is conducted in Chapter 6. This chapter also compares both versions to the original algorithm with respect to their overall performance and scalability. Finally, the gained advantages and possible future work are discussed in the conclusion.

2. Related Work

Particle filters were first introduced by Isard and Blake ([IB98]) in 1997 as a robust 2D contour tracking approach. A particle filter tries to estimate the state of a system at a regular time interval. Instead of giving a point estimate, the particle filter estimates a probability distribution over the state-space. A number of hypotheses, called particles, are proposed each time step. Given an observation, e.g. a camera image, the likelihood for each of the particles to have produced this observation is computed. The collectivity of all particles and their likelihoods approximate the distribution over the state-space, of which the mean or the most probable state can be used as point estimate. For subsequent time steps, a number of particles is resampled from the distribution and propagated with a function that best models the motion of the system.
By enabling the simultaneous evaluation of alternative hypotheses, particle filters present an interesting alternative to the Kalman Filter ([Kal60]) and its variations ([WM00], [TBF05]), which is why they have been widely applied to 2D and 3D object tracking problems.
The following sections describe some examples of GPU-accelerated particle filters which show similarities to the approach developed in this thesis.

2.1 Particle filter on GPUs for real-time tracking (Montemayor et al., 2004)

Montemayor et al. ([MPASF04]) introduced a preliminary design of a particle filter implementation on a GPU. Tested on a rolling ball sequence, 2D-tracking of the object was performed in real-time by comparing the captured image to a shape template of the object. In each time step, a square window of the image is captured for each particle, according to its (x, y) coordinates, as shown in Figure 2.1.1. The collectivity of all these images is stored in a square texture which is stored on the GPU. Subsequently, a second texture is created which contains the shape template, duplicated into a square image for each particle.
By using the fragment shader stage (see Section 4.1.2), these two textures are then multiplied, yielding the likelihood for each pixel. To sum up the likelihood for each pose, a GPU-accelerated method called mipmapping is applied, which scales down a texture by interpolating the values of adjacent pixels. The results are then used by the CPU to continue computation. By using the GPU for the evaluation step, a real-time performance of the algorithm could be achieved.

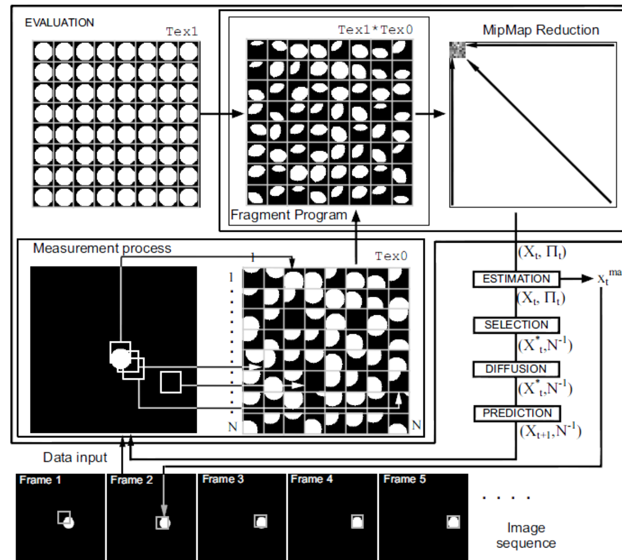

Figure 2.1.1: Hybrid GPU/CPU particle filter scheme (source: [MPASF04])

Despite using only a simple 2D-tracking mechanism, this work shows a basic approach to using the GPU for accelerating a particle filter. It also introduces the very important step of accumulating the information of all particles in one single texture, which saves a lot of latency and data transfer time.

2.2 A GPU-accelerated particle filter with pixel-level likelihood (Lenz et al., 2008)

A face- and hand-tracking algorithm published by Lenz et al.([LPK08]) uses solely OpenGL to accelerate the particle filter on the GPU. The camera image is preprocessed by an OpenGL fragment shader which uses a Gaussian Mixture Model to detect skin colored pixels and marks them as white. The renderings of the object are also conducted in white color. Afterwards, a binary XOR operation is performed on each pixel pair of the two textures which yields a residual image. The amount of non-zero pixels for each pose is then returned to the CPU to continue with normalization of the weights and resampling of the particles. Figure 2.2.1 depicts the subdivision of tasks between the CPU and the GPU.

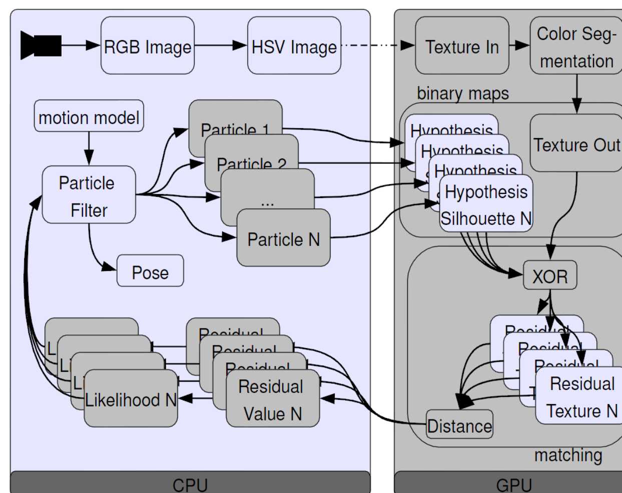

Figure 2.2.1: Subdivision of tasks between the CPU and the GPU (source: [LPK08])

This approach achieves a speedup between 3 (hand) and 7 (face) of the GPU version compared to the CPU implementation. The algorithm itself is dependent on single-colored objects and can only handle self-occlusions. It is able to evaluate 100 particles within 75 ms, while the accuracy is not described in the corresponding paper.

2.3 Real-time visual tracker by stream processing (Lozano and Otsuka, 2009)

Lozano and Otsuka ([LO09]) published an approach to object tracking, specifically faces, using the CUDA framework for the evaluation step of the particle filter. A set of feature points of a personalized 3D model of a human face is rendered (see Section 4.1.1) and their gray level is compared to the corresponding pixels in the observed camera image. These calculations are performed in parallel by a CUDA kernel (see Section 5.1.1) for each feature point in each pose. The kernel transforms each of the points by means of a weak orthographic projection and subsequently computes the matching error to the corresponding observed pixel. A second kernel is then executed for each pose, adding up their respective errors and storing these values in global memory, from which they can be transferred to the CPU for further processing.
While being able to evaluate 10,000 particles in approximately 12 ms, no information about the accuracy of the tracking is provided.

Figure 2.3.1: Face tracking with approximately 230 feature points (source: [LO09])

As our approach renders a full 3D model as opposed to a reduced number of feature points (see Figure 2.3.1), the rendering requires more time while providing a detailed and perspectively correct depth image. In our case, this rendering is performed with OpenGL (Section 4.1), as this framework is optimized for transforming pixels and rasterizing primitives. Our CUDA kernel operates in a similar fashion as the one described here, although handling multiple pixels per core and using shared memory for the summation of the weights. While handling self-occlusions of the face, occlusions through other objects are not explicitly modeled in this approach.

2.4 6-DoF model-based tracking of arbitrarily shaped 3D objects (Azad et al., 2011)

A monocular model-based approach to object tracking with a particle filter was proposed by Azad et al. ([AMAD11], [Mün10]), using edge detection. Canny edge detection as well as the sobel and prewitt operators can be used in this approach, depending on which one is the best fit for the tracked object. Each camera frame is preprocessed in a CUDA kernel with the aforementioned method and subsequently binarised to establish a monochrome image which shows distinct edges in the scene. Additionally, the object is rendered with OpenGL in the various different poses and the corresponding edge images are created. The comparison of both edge images is handled by executing a binary AND operation on all pixels with a CUDA kernel. Afterwards, the number of matching pixels is summed up per pose and transferred to the CPU. Figure 2.4.1 shows an outline of the GPU parallelization.

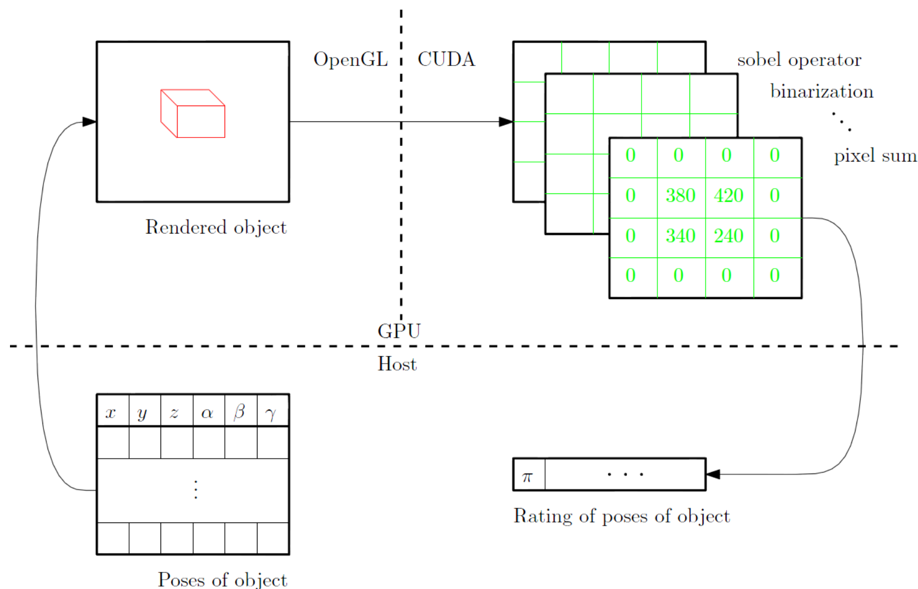

Figure 2.4.1: The poses are rendered with OpenGL and weighted with CUDA (source: [Mün10])

As OpenGL is used for rendering and CUDA kernels execute the weighting step, this particle filter implementation has some similarities with our approach. Partial occlusions are handled by the algorithm, however they are not explicitly modeled. Altogether, no real-time performance could be achieved, as issues with the data transfer between CPU and GPU were encountered. The approach presented in this thesis minimizes data transfers and avoids framework overhead as far as possible.

2.5 RGB-D object tracking: A particle filter approach on GPU (Choi and Christensen, 2013)

A recent work published by Choi and Christensen ([CC13]) uses GPU acceleration techniques that are very similar to ours. They track objects with a particle filter using not only depth, but also color and surface normal information. The renderings of the object in several poses are accumulated in one large texture (shown in Figure 2.5.1), which reduces mapping time into the CUDA framework. However, not all poses are being rendered by the OpenGL framework due to memory restrictions on the GPU. Instead, for subsequent

poses, the closest rendering is found and transformed with the respective pose. Afterwards, the information is compared to the observed image in a CUDA kernel.
While the GPU implementation shows great analogy with ours, this approach does not handle occlusions explicitly. With a runtime of approximately 40 ms for 100 particles, it is able to track complex objects in real-time by using photometric and geometric data.

Figure 2.5.1: Example object tracking with the algorithm of Choi and Christensen. Multiple poses are rendered into one large texture (source: [CC13])

2.6 Summary

Various approaches to particle filtering on the GPU have been presented in the above sections. While all of them attempt object tracking with OpenGL and / or CUDA, none of them models occlusion explicitly. The runtimes per particle differ greatly, as the complexity of the calculations depends on the purpose of the tracking and is different for each approach. Additionally, it is very likely that different hardware has been used for testing each of the presented algorithms, thus the listed runtimes are not directly comparable. The following Tables 2.1 and 2.2 give an overview of the similarities and differences between all approaches described.

Table 2.1: Comparison of various particle filter implementations on the GPU

Approach	OpenGL	CUDA	Single texture	Weights summation
Montemayor et al. [MPASF04]	weighting (multiplication)	-	yes	OpenGL mipmapping
Lenz et al. [LPK08]	image binarisation, rendering, weighting (XOR)	-	no	OpenGL fragment shader
Lozano et al. [LO09]	-	point transformation, weighting	no	CUDA kernel
Münch [Mün10]	rendering	edge detection and binarisation, weighting (AND)	no	CUDA kernel
Choi et al. [CC13]	rendering	weighting	yes	CUDA kernel
this approach	rendering	weighting	yes	CUDA kernel

Table 2.2: Comparison of various particle filter implementations on the GPU (cont.)

Approach	Time p.P. (640 x 480)	Handles occlusion	Weighting	Year	Object
Montemayor et al. [MPASF04]	(real-time)	no	color (binary)	2004	2D shape
Lenz et al. [LPK08]	750 µs	only self-occlusion	color (HSV)	2008	face (2D), hand (3D)
Lozano et al. [LO09]	1.2 µs	only self-occlusion	intensity	2009	face (3D)
Münch [Mün10]	3.04 ms	yes, implicitly	edges (binary)	2010	arb. obj. (3D)
Choi et al. [CC13]	415 µs	only self-occlusion	color (RGB), depth, normals	2013	arb. obj. (3D)
this approach	115 µs	yes, explicitly	depth	2014	arb. obj. (3D)

3. Foundation

This chapter gives an overview of the object tracking algorithm that is parallelized in this thesis, followed by an explanation of the basic principles of parallelization.

3.1 Probabilistic object tracking

The object tracking algorithm used in this thesis was developed by Wüthrich et al. ([WPK+13]) at the Max Planck Institute for Intelligent Systems (Autonomous Motion Department) in Tübingen. It requires a Microsoft Kinect or a similar sensor and a 3D model of the object for a successful object tracking. As the purpose of this thesis is the efficient parallelization of this algorithm, only the details necessary to fulfill this task are explained in this section. A more detailed explanation and the mathematical background to this algorithm can be found in the according paper ([WPK+13]).

The algorithm employs a particle filter (see Chapter 2), which means that at any given point in time, the state of the object is approximated by a discrete distribution over the state space. In our case, the state is six-dimensional, incorporating the position and the orientation of the object relative to the camera. Each particle contains such a state and a weight that represents the probability that the object is currently in this state.

In order to correct the predicted poses with respect to the real position of the object, the observation of the sensor is needed. The Kinect delivers a depth image of the observed scene at a rate of 30 Hz, meaning the algorithm gets 30 observations per second that can be evaluated. Each 1/30th of a second is referred to as a frame in this thesis.

In general, this algorithm first issues a number of predictions for the pose of the object. Using the observations obtained through the camera, it then determines how likely each of those predictions is and assigns corresponding weights to them. Subsequently, those predictions with a low weight are discarded while the ones with a high weight are reused and duplicated for predicting the pose in the next frame.

To generate the initial particles, the algorithm assumes the object to stand on a flat surface, for example a table. It then samples around clusters of points on this table. Sampling here means randomly picking potential object poses according to a given distribution. In our case, this distribution is represented by a set of gaussians, each centered around the mean of one cluster, with a variance that can be manually adjusted.

The algorithm itself consists of three steps that are executed in an infinite loop.

1. Propagation
   Given a particle, it is propagated by adding random noise to each dimension of the predicted pose, respectively sampled from a gaussian distribution. It represents our expectation of how the object is going to move in between two time frames.
   If the object is being moved by a robot, additional control inputs can be used for a more accurate propagation of the particles. As working with a robot was not feasible in the course of this thesis, no control inputs are taken into account. However, accomodating them should not have a noticeable impact on the runtime.

```
1 for each pose
2   for each dimension
3     x = x + random number sampled from gaussian distribution
4   end for
5 end for
```

Listing 3.1: Propagation step

2. Evaluation
   For each particle, the object is rendered in the corresponding pose into an image of the same size at the same angle as the camera image. Afterwards, each rendered depth image is compared with the real depth image from the sensor. This comparison is performed pixel-wise. The less difference between the values of the rendered image and the observation, the more accurate is the estimated pose. Thus, the likelihood of the particle, also referred to as weight in this case, is set accordingly. As we do not model the surroundings of the object, only the pixels that are occupied by the rendered object are compared.
   Additionally, the algorithm can handle partial occlusions of the object, for example through another object or a human hand. This is why every pixel is assigned a certain probability of being occluded. This probability is taken into account when comparing the depth values.
   Furthermore, the occlusion probability is updated each frame, depending on the computed likelihood. For instance, if the two depth values are very close together, the probability that this pixel is still occluded is very low. As the occlusion probability is propagated with time, but only a subset of pixels is updated every frame, an additional value per pixel needs to be saved, which contains its last update time.
   After evaluating each pixel separately, the logarithms of their respective likelihood are summed up to yield the overall likelihood, or weight, of the pose.

```
6  for each particle
7    render the pose and obtain depth values per pixel
8    for each pixel occupied by the object
9      compute likelihood from rendered depth, observed depth, occlusion
           probability and update time
10     add log(likelihood) to overall log_likelihood of this pose
11   end for
12 end for
13 output the weighted mean of all poses as estimate
```

Listing 3.2: Evaluation step

3. Resampling
   In this step, new particles are sampled from the distribution of the evaluated particles. This process typically discards poses with a very low weight, while poses with a high weight are picked several times. However, we do not evaluate the same pose twice, as these particles are propagated again in Step 1. The pose data, as well as the occlusion probabilities and the update times of the pixels need to be copied into

new data structures, as they will be altered for each particle individually in the next frame.

```
14 for each particle
15   sample a new particle according to the weight distribution obtained
       from the previous step
16   copy all the relevant data of that sampled particle (pose, occlusion
       probabilities, update times)
17 end for
```

Listing 3.3: Resampling step

As can be seen from the above pseudo-code listings, each of the steps in this algorithm is executed for each particle individually. Within each step, the calculations per particle are independent, which allows for optimal parallelization. For example, the computations for each particle can be handled by separate threads.
However, there has to be a synchronization in between the evaluation and the resampling step, as the likelihoods for each particle need to be known in order to contribute to the distribution from which the new samples are drawn.
To determine the runtime of each step of the sequential implementation, the **gettimeofday()** function is used. It is available in Linux and returns the time that has passed since midnight UTC on January 1, 1970. This timer is commonly used to determine the actual time that passes while executing a function. Alternatively, CPU timers can be used, which compute the time based on the cycles needed for executing the instructions and the clock speed of the CPU. Often, additional processes run in the background, like the operating system or the development environment for the code, which lets these timers yield results that are unsuitable for our purposes.

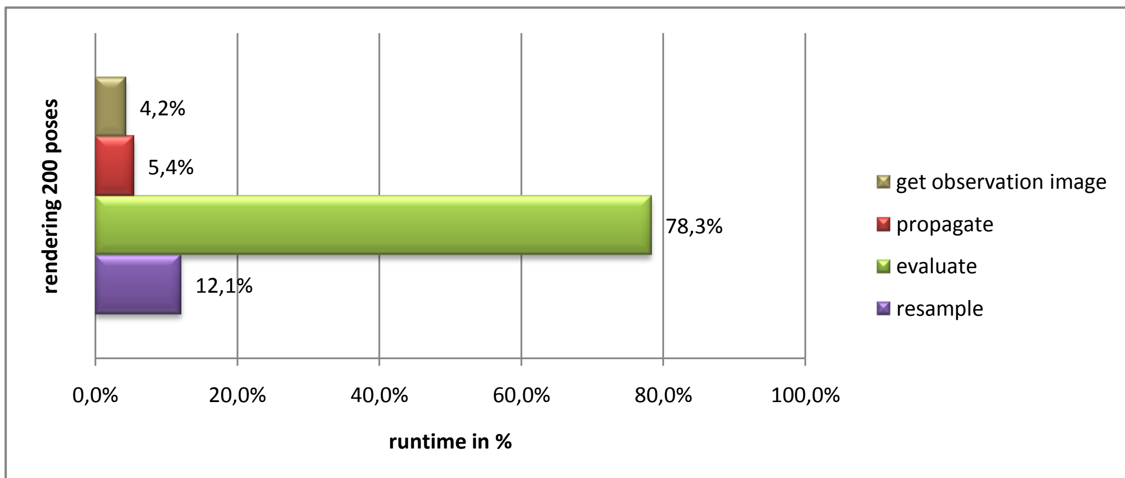

Figure 3.1.1: Runtime analysis of the original algorithm

As can be seen from Figure 3.1.1, the greatest computational effort lies within the evaluation step. It takes up 78% of the overall runtime of a frame. Because of the heavy use of object rendering in this step, a parallelization on the GPU appears very promising, as it is exactly what a GPU is designed to do. This thesis focuses on parallelizing this evaluation step, although the propagation and resampling steps have been parallelized for completion, too, which is taken into account in the overall speedup evaluation at the end of this thesis.

3.2 Parallelization

Time is of the essence in many computer applications today. Be it a computer game that has to render sophisticated graphics in real-time or a robot that is designed to play table tennis with a human - the speed of a software often defines its usability. For quite some time, speeding up applications could be achieved by increasing the clock speed of the Central Processing Unit (CPU) in a system. This way, more instructions per second can be processed, reducing the runtime of every algorithm that can be executed on a CPU. As this clock speed has reached its maximum due to heat and power restrictions, other means of accelerating computations have to be found. Dividing an application into multiple jobs that can run in parallel is one of the most promising approaches.
For this to work, the application has to provide computations that are independent of each other in the sense that, for example, one computation does not need the result of another as input (see Figure 3.2.1).

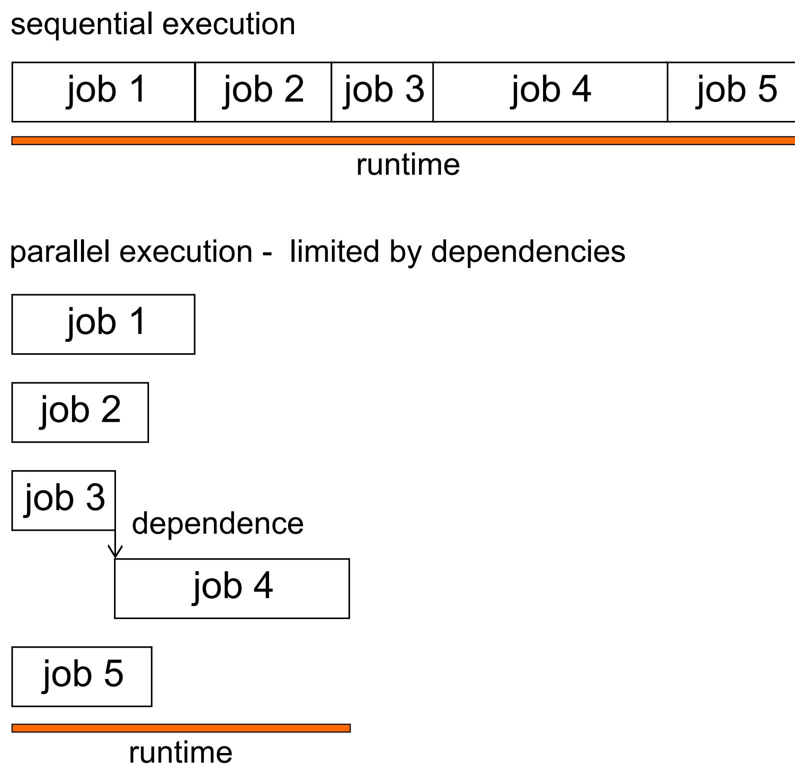

Figure 3.2.1: Parallel execution of multiple jobs

In addition, the application should be computationally intensive and should not mainly consist of data transfer. For instance, an application that merely copies big amounts of data from one location to another is limited by the bandwidth of the interconnection between the data locations (see Figure 3.2.2).

If an application is parallelizable, some of its computations can be run in multiple threads on a single CPU, a multi-core CPU or specialized hardware like Graphics Processing Units (GPUs). While a single CPU can only provide virtually parallel execution of the threads, multi-core CPUs allow for physically concurrent execution, as they comprise several Arithmetic Logic Units (ALUs) and Control Units (CUs) that can fetch and execute instructions. Modern end-consumer multi-processors usually contain between 4 and 16 ALUs (from now on referred to as cores), while the newest high performance processor Intel Xeon Phi incorporates 61 cores ([Cor12]).

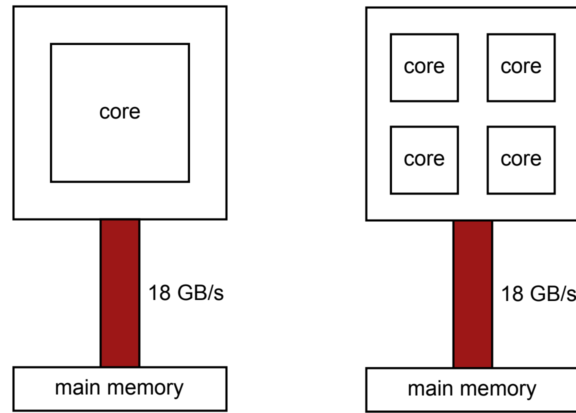

Figure 3.2.2: Many cores do not accelerate data transfer

When parallelizing an application, it has to be determined whether it is more suited for execution on a CPU or a GPU. The main advantage of the CPU is its Multiple Instruction Multiple Data (MIMD) capability, which allows parallel execution of entirely different instructions. Additionally, direct access to main memory and efficient caching of data gives the CPU an advantage in processing a lot of data. The biggest limitation of CPUs is their currently low number of cores, as the possible speedup is always limited by it.

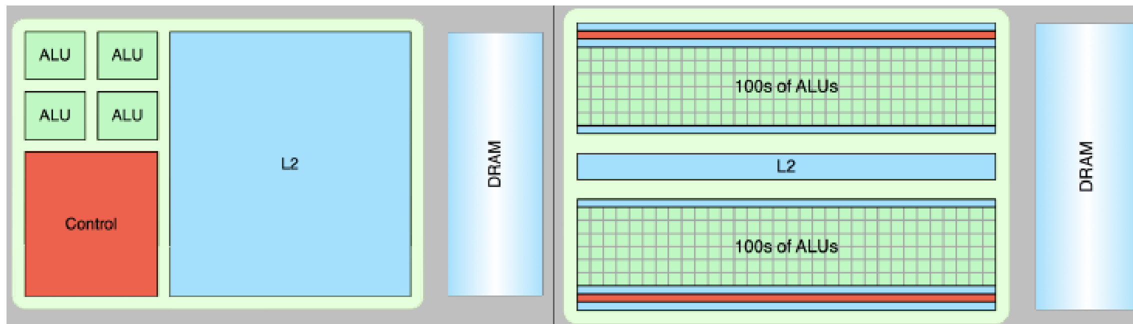

Figure 3.2.3: Hardware comparison between CPU (left) and GPU (right) (source:[Tat13])

Modern GPUs, however, typically incorporate between 256 and 2880 cores ([Cor14]). Although these cores cannot be compared to CPU cores when it comes to computational power, the theoretical speedup is much higher for applications that use a lot of Single Instruction Multiple Data (SIMD) instructions. More details about GPU cores are explained in the CUDA section (Section 5.1).
The biggest disadvantage of a GPU is that it cannot access main memory directly. The CPU first has to issue a transfer to GPU memory, which will then be executed by Direct Memory Access (DMA). This transfer is also quite slow, which is why applications with a lot of data traffic are not likely to scale on a GPU. Since invoking functions that run on the GPU also causes some communication overhead, the problem to be parallelized has to be big enough so that the speedup will even out this overhead.

In conclusion, GPUs should be used for big, SIMD-based problems with a minimum of data transfer to or from main memory. CPUs are better suited for algorithms that need to execute different instructions in parallel or that rely heavily on data transfer. Object tracking with a particle filter is highly parallelizable, since each particle can be evaluated individually. Since not a lot of data is needed and the computations for the particles are alike, parallelization on the GPU sounds the most promising.

4. Pure OpenGL approach

To achieve a notable speedup in performance it is essential to parallelize the evaluation step, as it contributes 78% of the overall runtime of the object tracking algorithm. For every frame, this step renders the propagated poses and calculates their corresponding likelihoods.
Since GPUs were specifically designed for rendering purposes, an obvious approach is to shift these calculations from the CPU to the GPU. There are several frameworks that facilitate use of GPUs; amongst it is the Open Graphics Library (OpenGL) which provides a set of functions to render a scene. This chapter explains the basic functioning of OpenGL, followed by a description of how we use it for our purposes and the performance gain achieved. Readers that are familiar with OpenGL can skip to Section 4.2.

4.1 OpenGL

The OpenGL framework is a cross-platform application programming interface (API) that is widely used for rendering 3D computer graphics on the GPU. Rendering is the transformation of 2D or 3D geometry into 2D images. OpenGL has first been published by the Silicon Graphics Inc. (SGI) in 1992 and has been further developed ever since. Because of its extensive documentation and ease of use, OpenGL has become very popular and is used mainly in scientific visualization, computer-aided design (CAD) and computer games. This section gives an overview of the rendering process in general and the OpenGL constructs that accelerate this rendering on the GPU.

4.1.1 Rendering

Rendering is the process of creating a two-dimensional image from a virtual scene. Which parts of this scene will be visible depends on the positioning of a virtual camera, from which the scene is observed. The process of rendering typically contains the following two steps:

1. Transform all objects in the scene so that their coordinates represent their position relative to the camera from which the scene is viewed
2. Given the desired resolution of the final image, determine for each pixel which object occupies it and assign the appropriate color and depth to it

Every object is defined by its object model, which consists of a set of vertices in a cartesian coordinate system (called model space) with the origin being the center of the object.

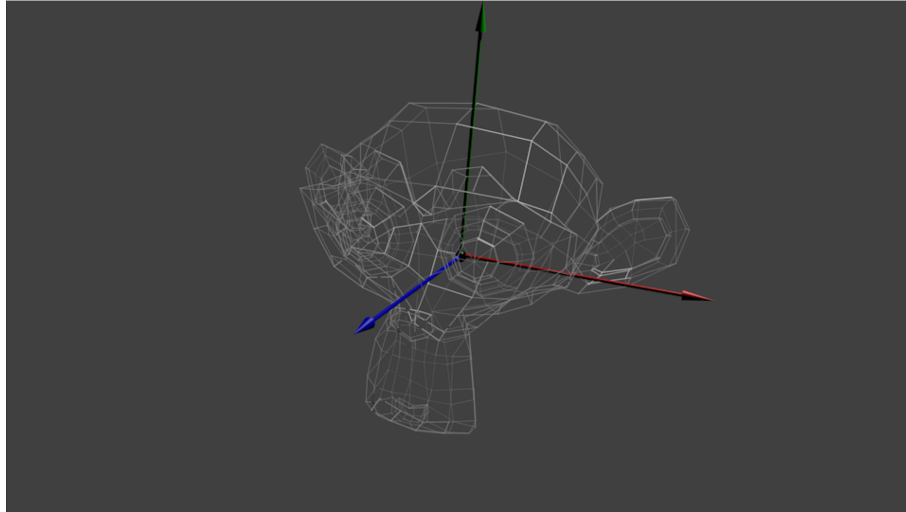

Figure 4.1.1: Model space (source: [Ope14])

The scene itself, on the other hand, is defined in world coordinates (world space), with the origin being the center of the scene. According to the position and orientation that an object is supposed to have within the scene, the definition of its vertices needs to be changed. They have to be defined relative to the scene's origin instead of the object's origin. If not changed, every object would be positioned in the center of the scene, in its standard orientation. This transformation can be expressed by a single matrix in homogeneous coordinates ([Ope14, Tutorial 3]), the model matrix.

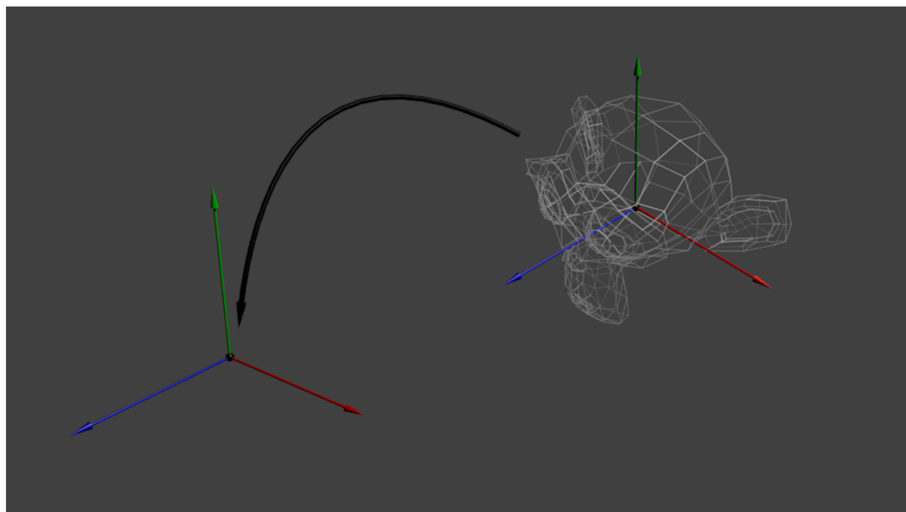

Figure 4.1.2: World space (source: [Ope14])

To view this world through the camera, another transformation with the so-called view matrix is necessary, that defines all vertices relative to the camera's position and orientation. An important characteristic of the camera coordinate system (camera space) is that its z-axis points toward the observer, while the x- and y-coordinates are aligned with the image plane.

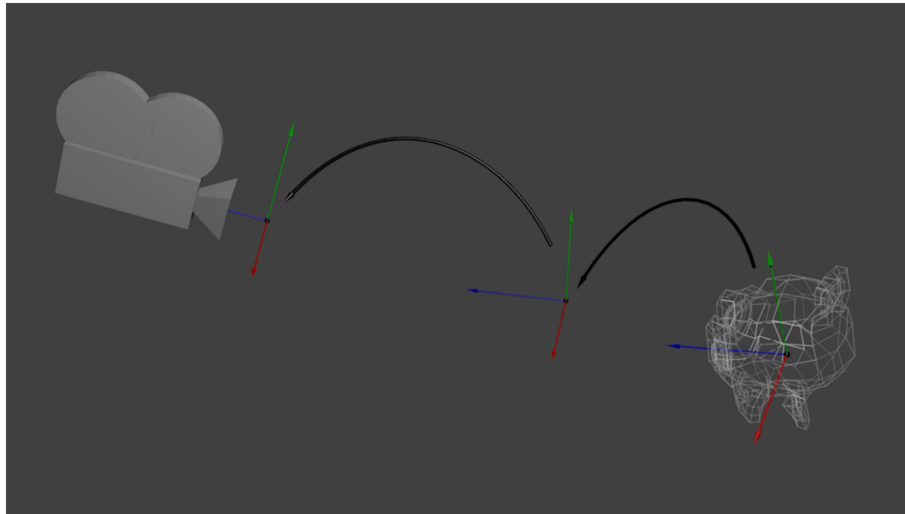

Figure 4.1.3: Camera space (source: [Ope14])

To gain perspective vision on the scene, a perspective projection is done, which lets objects that are further away from the camera appear smaller than objects right in front of it.

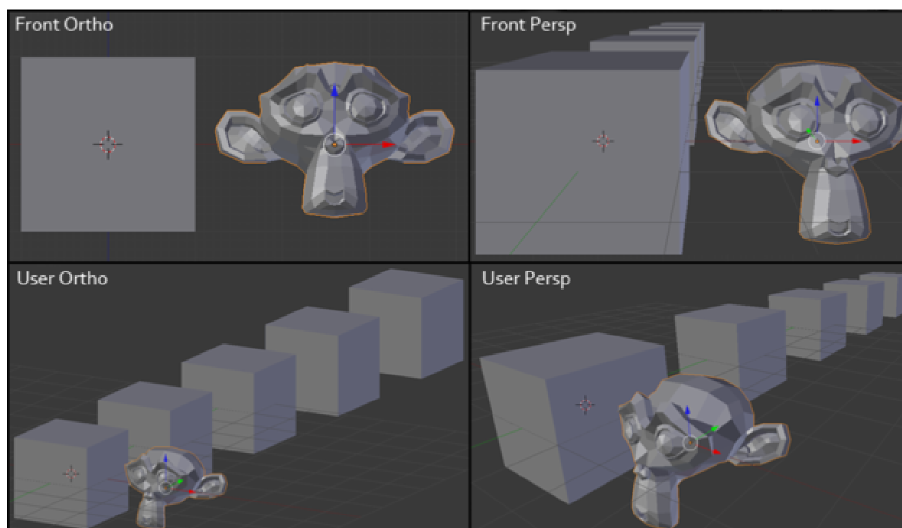

Figure 4.1.4: Orthographic projection (left) versus perspective projection (right) (source: [DeV11])

Instead, an orthographic projection can be used, which is common in computer-aided design (CAD), as parallel lines should appear parallel, for example when constructing a building element. It is also used by Lozano et al. ([LO09]) together with a scaling factor, to lower the computational cost of the algorithm. However, as we try to model a sensor that has perspective vision, we use a perspective projection.

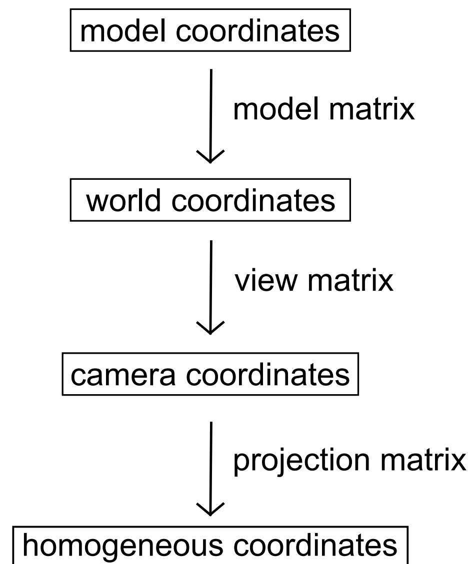

Figure 4.1.5: Summary: All transformations of the vertices (source: [Ope14])

After a successful transformation of the vertices, it has to be determined which pixels are occupied by the triangles composed of them. At this stage, a pixel is called a fragment and it can carry color as well as depth information. Assigning these values requires a discretization of the image whose step size depends on the resolution. In the process called rasterization, which is used by OpenGL, every triangle of the object is rasterized by itself and for each occupied fragment, the color- and depth-values of the closest vertices are interpolated. After the rasterization, typically more than one object occupies the same pixel in the image which is why only the foremost fragment is drawn or a combination of fragments (e.g. when using transparent objects).

In photorealistic rendering, a technique called raytracing has become very popular, which does not rasterize the objects, but instead shoots a ray through each pixel and determines which object is intersected by it in the scene. This is useful to represent realistic lighting, as these rays can be transmittted through or reflected at the object, to take into account indirect lighting as well. However, raytracing is a computationally very expensive process. As our algorithm solely uses the depth values of the pixels, we use the rasterization technique for rendering.

4.1.2 Pipeline

To render a scene, OpenGL runs it through a sequence of stages denoted as the rendering pipeline. In early versions of OpenGL, this was a very simple pipeline that only contained the most fundamental stages necessary for rendering. Some of these stages were hardwired on the graphics chip, which made their execution very fast. With time and with increasing demand from the developers, the pipeline was frequently refined to provide more flexibility and additional features. In this section, a modern version of the OpenGL pipeline is described; stages, which are however negligible for this thesis, are omitted.
The essential steps of the pipeline are:

1. Vertex data is sent to a GPU buffer by the host program running on the CPU
2. The vertices are projected into image space
3. They are assembled into triangles

4. They are rasterized into pixel-sized fragments
5. These fragments are assigned color values and drawn to the framebuffer

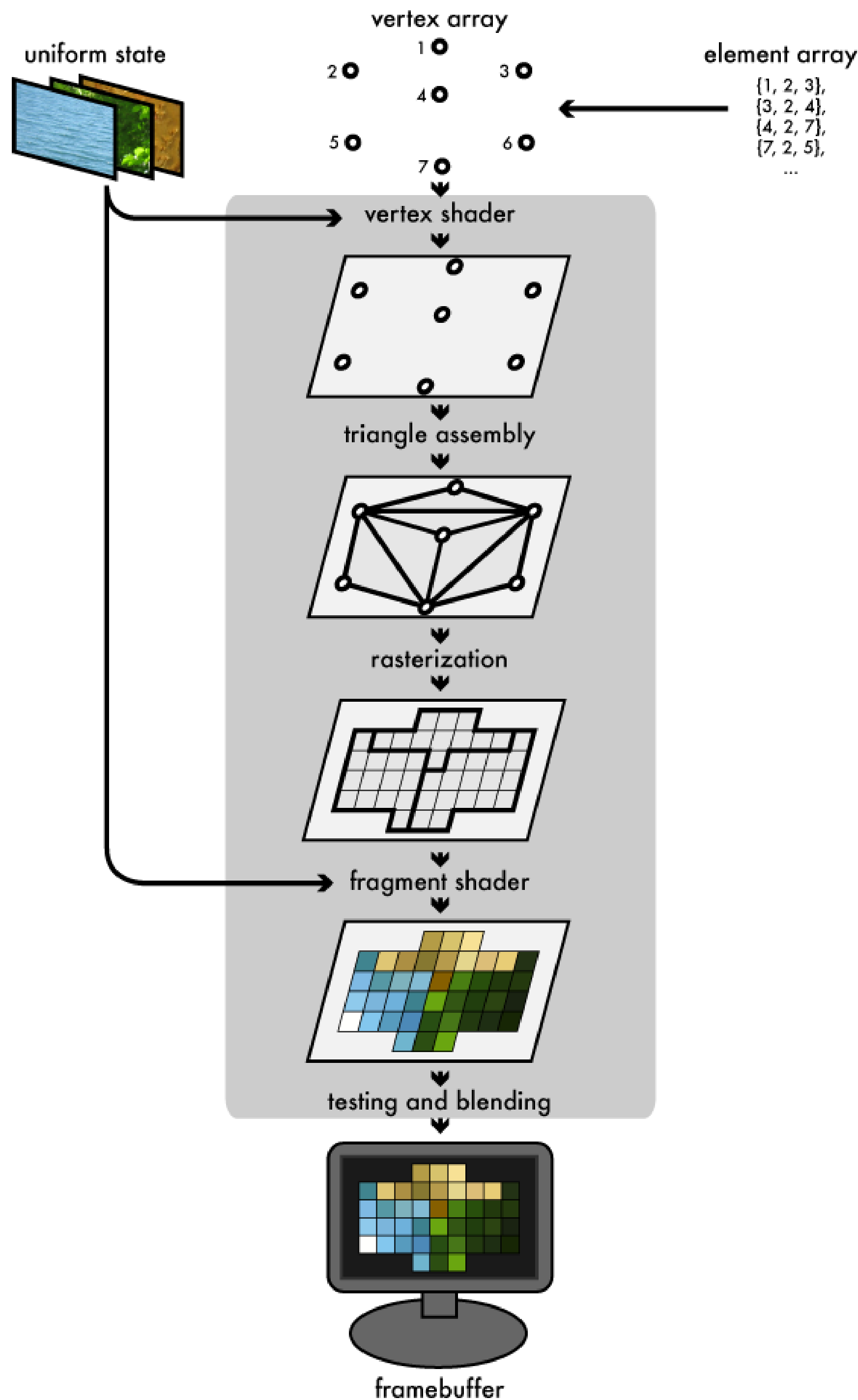

Figure 4.1.6: OpenGL rendering pipeline (source: [Gro12])

Initially, vertex attributes of all vertices (of every object) in a scene are copied to buffers on the GPU, called vertex buffers. Typically, these attributes include the position in model space and the color of the vertex. Generally, an attribute can possess any value. The various vertex buffers are collectively called a vertex array, which is indexed by the element array. The indices select the order in which the vertices get fed into the pipeline.

This order later determines which vertices are assembled into a primitve, for example, in a basic case, every three subsequent vertices build one triangle, as illustrated in Figure 4.1.7.

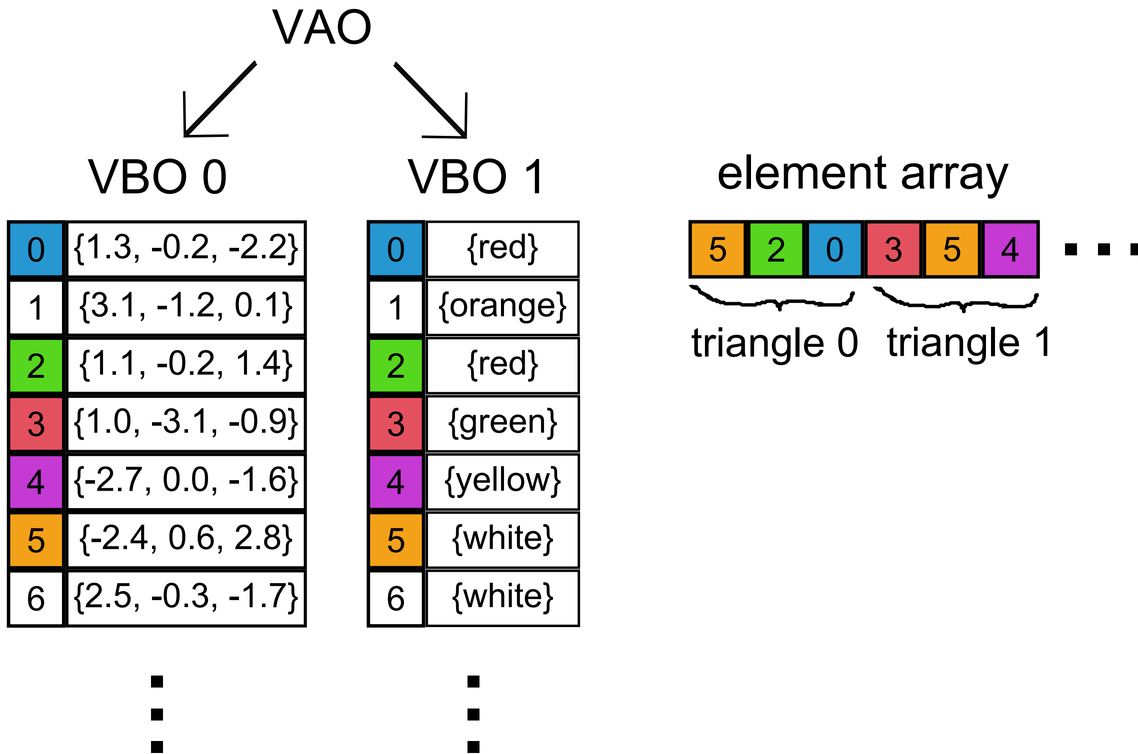

Figure 4.1.7: OpenGL object representation: The Vertex Array Object (VAO) accumulates pointers to all Vertex Buffer Objects (VBOs), which contain vertex attributes like positions or colors. The element array indexes the VAO and thus the VBOs.

The vertex attributes are used as input to the vertex shader, an uploadable program written by the developer in a high-level language called OpenGL Shading Language (GLSL). This shader is executed for every vertex separately and it usually transforms the attributes and feeds the results to the next pipeline stage. At a minimum, the vertex shader performs the transformations mentioned in the previous Section 4.1.1 to project the vertices into image space. The needed matrices can be sent to the shader by the host program as uniform variables, which means every instance of the shader gets the same value.

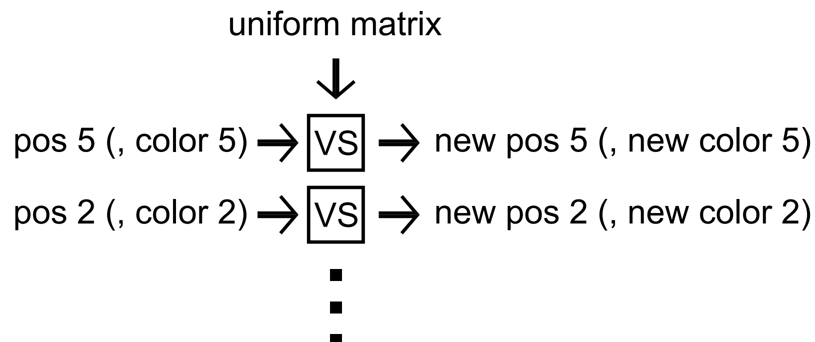

Figure 4.1.8: The vertex shader (VS) program is executed for every vertex individually.

The vertex shader can also perform other calculations on the vertex attributes, for example the color value can be changed according to lighting conditions in the scene.
Theoretically, the programmer can perform any desired calculation in this shader. This flexibility is commonly used to perform general purpose computing with OpenGL, although it can be tricky to adjust a general computation to the graphics environment.
The next stage assembles the vertices to primitives depending on their order and the assembly mode specified by the developer. In our case, this mode is set to connect every three vertices to a triangle, but in general, several ways of connecting the vertices can be chosen and primitives can also be points or lines. If a triangle lies outside of the viewing frustum (which means the visible area), it is discarded in this stage to save redundant computations. If only part of the triangle lies outside, it gets clipped to the frustum and possibly sub-divided into multiple triangles.

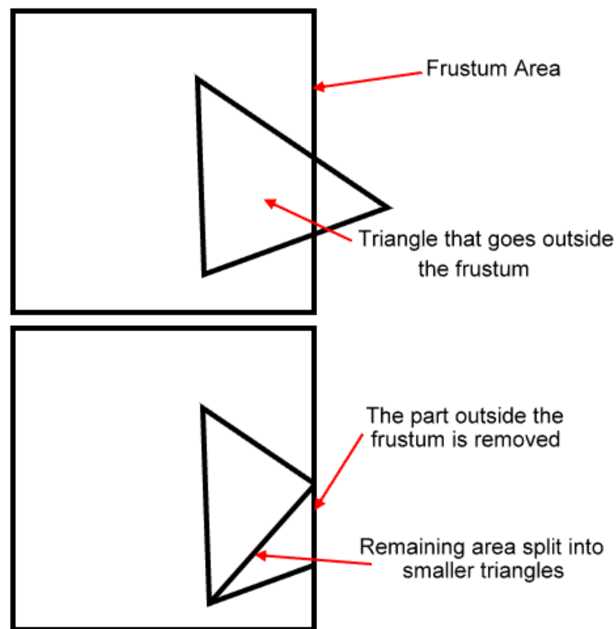

Figure 4.1.9: Triangles outside the viewing frustum get clipped to the frustum planes (source: [cli])

The following rasterizer breaks the triangles into pixel-sized fragments, where each fragment contains the interpolated values of the respective vertex attributes. For example, the color of a fragment is composed of the three colors of the vertices of that triangle (see Figure 4.1.10.

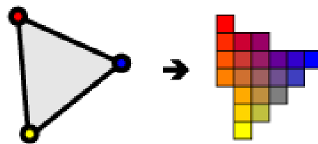

Figure 4.1.10: OpenGL rasterization (source: [Gro12])

The same is true for the position including the depth of a pixel. Even though the depth will not be visible in the resulting 2D image, it is necessary to handle occlusions in the scene. If two fragments are allocated to the same pixel, the one with the smaller depth value will be visible in the image. This procedure is called z-testing and it uses a separate

depth buffer to keep track of the smallest depth values of the fragments already processed. If the z-value of a fragment is larger than the one stored in the buffer, it can be discarded.

The last relevant stage in the pipeline used here is the fragment shader. This is another programmable element that receives the fragments with their interpolated values as input. Again, this shader can perform any desired calculation, but it is typically used to output a final color per fragment which gets drawn into the framebuffer. The framebuffer contains not only a single 2D-image for screen output, but instead provides the possibility to write to multiple textures. A texture is an OpenGL object that contains an image. Often, the terms texture and buffer are used interchangeably, but a buffer can typically contain any sort of data while a texture is meant to store only image data. In our program, we use a depth texture to store the transformed depth values for z-testing and a color texture to store the real depth in meters. Before every render pass, it is important to reset all values in these textures to a default value, so that data from previous render passes is erased. This process is referred to as clearing the texture.
After all fragment shader instances have finished execution, the result in the framebuffer can be used to present a picture to the user or for further processing.

4.2 Rendering a pose with OpenGL

In this approach, the OpenGL pipeline is used to render the object in a specific pose. Afterwards, the depth values are extraced from the framebuffer and further processed on the CPU to compute the likelihood. In each frame, both of these processes are executed for each propagated pose.

The data needed by the pipeline for a successful rendering is merely the object mesh and the transformation matrices that define at which position and in which pose the object should be rendered into the 2D image. The object mesh is sent to the GPU as a list of vertices, whose attributes are stored in OpenGL vertex buffers. For our purposes, the only attribute needed is the vertex position. Additionally, the OpenGL element array needs to be filled with the indices that determine which vertices are assembled into triangles. Since the program only needs to track this one object, it can be transferred once in the initialization phase and it can be reused for every rendering.

This leaves the matrices as the only data that has to be transferred to the GPU for every pose. All matrices can be sent to the shader programs as uniform variables with a simple OpenGL call. These variables keep their value throughout the whole rendering process, so they are the same for every vertex of the object. The projection- and the view-matrix are constant in our application. The former can be calculated from the intrinsic camera parameters of the Kinect sensor or the respective camera used. The view matrix equals the identity matrix in our case, because the Kinect sensor assumes itself to be positioned in the origin of the world coordinate system. The model matrix has to be calculated for every pose from the position and the rotation given.

The program for the vertex shader, which processes every vertex by itself, is listed below. The abbreviations used are explained in Table 4.1.

Table 4.1: Abbreviations used in various code listings

C	CPU
G	GPU
s	sequential
p	parallel
->	transfer to

```
18          #version 330
19
20          // tell OpenGL which buffer corresponds to which input
21          layout(location = 0) in vec3 vertexPosition_modelspace;
22          uniform mat4 MV;
23          uniform mat4 P;
24          out float depth;
25
26 (G, p)   void main() {
27              // makes the vector homogeneous
28 (G, p)       vec4 v = vec4(vertexPosition_modelspace, 1);
29 (G, p)       vec4 tmp_position  = MV * v;
30 (G, p)       depth = tmp_position.z;
31 (G, p)       gl_Position = P * tmp_position;
32 (G, p)   }
```

Listing 4.1: Vertex shader program

Essentially, the incoming vertex position has to be transformed by the matrices and passed to the next pipeline stage. However, the perspective projection, which is necessary for a correct rasterization, poses a problem in regard to the depth values. It maps all of them to a range of [0, 1] which means they do not correspond to the real distance in meters anymore, as the Kinect sensor has a range of [~ 0.7 m, ~ 7 m]. Naturally, we could retransform the depth values to this range after the rendering process is completed, but there is a simpler and faster solution. Before performing the perspective projection, the depth values can be saved in a separate output value of the vertex shader which will be interpolated by the rasterization stage and can subsequently be read by the fragment shader. This adds a little overhead to the rasterization stage, but it is negligible in comparison to the otherwise unavoidable retransformation cost.

The fragment shader, which is executed for every fragment of each triangle, is the last programmable pipeline stage. Its only purpose in our application is to write the incoming depth value per fragment into the framebuffer. Additionally, z-testing has to be enabled to discard fragments that are occluded by others.
The default framebuffer used for OpenGL rendering contains a color- and a depth-texture, where the former is sent to the screen to present the final image while the latter is used for z-testing. Given that we do not wish to present any visible data of this rendering to the user, we need to create a custom framebuffer. OpenGL provides some functions to attach textures with arbitrary formats to this framebuffer. As the integrated z-testing mechanism can only work with the projected depth values of range [0, 1], we need a depth-texture to store these values. Additionally, we create a color-texture that stores the real depth values in meters. Caution has to be taken when chosing the format of this texture, which consists of the data type and the channel (red, blue, green, alpha) type. For most formats, OpenGL automatically clamps the values in the texture to the range of [0, 1]. This

means, values that exceed this range are set to the respective border value. For us, this would prohibit the use of depth values that are greater than 1, thus making the tracking of objects that are futher away than 1 meter from the camera impossible. Therefore, the format GL_R32F is chosen for the color-texture, which uses the red channel with a 32-bit floating point value per pixel.
Once every fragment has been processed, the rendering process (see Figure 4.2.1) is completed.

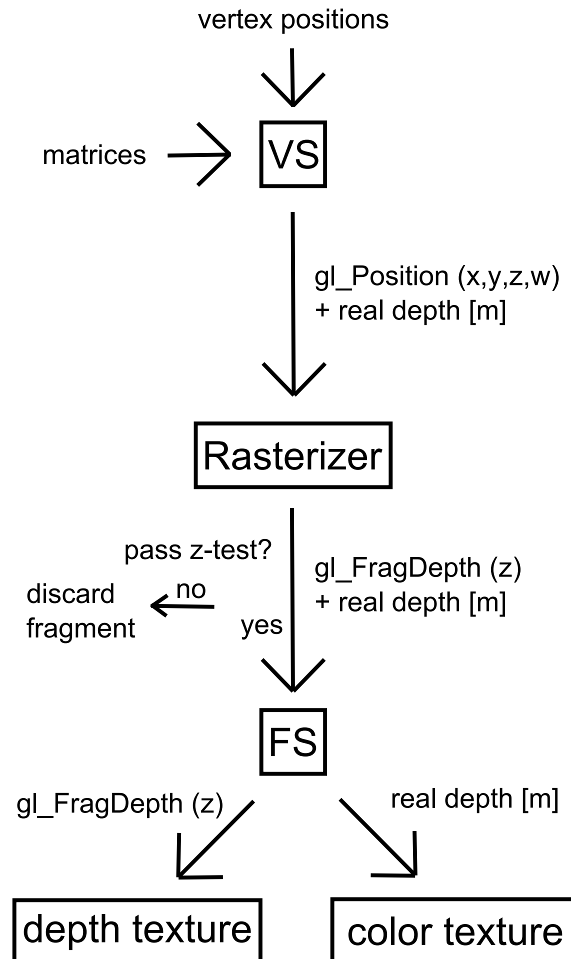

Figure 4.2.1: OpenGL pipeline usage in this approach

A CPU cannot access GPU memory directly, which is why the resulting depth values have to be copied from the framebuffer to main memory. This is done using a Pixel Buffer Object (PBO), a construct provided by OpenGL to allow for asynchronous data transfer between main memory and GPU memory. Once initiated, the transfer is executed via Direct Memory Access (DMA) while the GPU can continue execution of successive commands. While this approach does not benefit from an asynchronous transfer, using a PBO still proved to be faster than copying the data manually.
Once the data is transferred, it has to be filtered to identify the pixels that are occupied by the object, since only these pixels are used for computing the likelihood. For this, a comparison to the default depth value of 1.0 is the only option. The default value can gerenally only be set in the range of [0, 1]. In the rare case that a pixel's depth should be exactly 1.0 meter, this test would fail and the pixel would be discarded. Since the depths are saved as float values, it is highly unlikely that this problem will ever occur and even in that case, discarding one pixel does only have a small effect on the overall likelihood.

Once the relevant pixels have been extracted, the CPU can continue with the likelihood computation. Afterwards, the algorithm proceeds with the next pose in the same manner. To conclude this section, the following pseudo-code listing summarizes the above depicted algorithm. Here, the abbreviations depicted in Table 4.1 are used to describe the mode of execution for each instruction.

```
33 (G, s) set up custom framebuffer
34 (G, s) set up the shaders
35 (C, s) for each pose
36 (C, s)   calculate model_matrix from state
37 (C->G)   send matrices to shaders
38 (G, p)   clear the framebuffer
39 (G, p)   render on GPU
40 (G->C)   receive depth data from GPU
41 (C, s)   for each pixel
42 (C, s)     if pixel_depth != 1.0 then
43 (C, s)       add pixel_index, pixel_depth to list
44 (C, s)     end if
45 (C, s)   end for
46 (C, s)   call likelihood_calculation(list)
47 (C, s) end for
```

Listing 4.2: Evaluating the poses by using OpenGL for the rendering process

4.3 Performance and bottlenecks

All in all, the rendering step of this implementation has a runtime of 9.70 ms for 200 poses, which yields a speedup of 2.12 compared to the 20.58 ms needed in the original algorithm. Taking into account the weighting, propagating and resampling which is still executed on the CPU, the speedup of the overall runtime is only 1.61. This allows us to evaluate about 320 poses, though the speedup is considerably smaller than expected from a GPU parallelization. Especially with 448 cores as opposed to just one core, a higher speedup should be possible, even though the GPU cores are not directly comparable to CPU cores.
On the other hand, GPUs were designed to handle bigger problem sizes than provided in this algorithm. Typically, millions of triangles have to be rendered with a GPU, not only 420 as for the tracked object used here. A higher speedup of the OpenGL approach can be observed when using highly complex objects, as is discussed in Section 6.2.

To shed light on the bottlenecks in this approach, a detailed runtime examination of the algorithm was conducted. It is presented in Section 6.2, while the most important findings are mentioned here.
Whereas the actual rendering only constitutes 12% of the runtime, the main bottleneck is the data transfer of the depth values to the CPU. It takes 64% of the runtime, while the remaining 24% are used to filter the relevant depth values on the CPU.

All in all, this approach shows that rendering the poses with OpenGL shows promise, though a solution to the data transfer problem needs to be devised. Therefore, the next chapter describes an approach that shifts the likelihood computation to the GPU as well, thus enabling the data to stay on the GPU.

5. Combined approach with OpenGL and CUDA

The first approach showed that the data transfer of the rendered depth values poses a big problem. The only chance to avoid this transfer is to put the likelihood calculation on the GPU, too. However, this computation needs the occlusion values for each pixel for each pose, as described in Section 3.1, which is twice as much data as before. Obviously, transferring this data for every computation would have a huge impact on performance which is why it has to be stored on the GPU permanently.
Implementing this step with OpenGL is possible, however it requires some effort, because data structures have to be disguised as textures and calculations have to be performed in shaders, which are only called for a certain subset of fragments. Furthermore, it is driver-dependent where data is stored with OpenGL, which can lead to non-deterministic performance. All in all, OpenGL proved to be very efficient for rendering, but the transparency of its calls leaves a lot to be desired. This is why CUDA, a framework specifically designed for general purpose computing on the GPU, presents an interesting alternative. It gives the user maximal flexibility and control over data structures and where and when to execute computations. Thereby, CUDA is a much better fit to compute the likelihood of each pose.
Combining the two frameworks is simple, as the CUDA framework allows the user to access textures created by OpenGL for read- and write-operations. This means we can render a pose with OpenGL and subsequently read the depth values from the framebuffer texture with CUDA for the likelihood computation. As every mapping of a texture to CUDA introduces some latency, frequent switching between the two frameworks is not desirable. Thus, the rendering of all poses in one frame should be completed before switching to CUDA for the likelihood computation. However, the CUDA Compute Capability used (see Section 6.1) only allows a constant number of OpenGL textures to be accessed. This number needs to be known at compile time which poses a problem if we want to pass the number of poses and thus the number of textures as a parameter to the executable program.
A possible solution to the problem is allocating a large constant amount of textures that will most likely never be met by the number of poses specified. Apart from a possible waste of GPU memory, the scalability of this solution is limited. A far cleaner and common alternative is to place all rendered poses in one large texture, which requires a bit of restructuring in the rendering process.
This chapter briefly describes the CUDA framework and its most important components,

followed by the CUDA implementation of the likelihood calculation and the required modifications to the OpenGL rendering process. Finally, the performance of this approach is compared to the previous solution. Readers that are familiar with the CUDA framework can skip to Section 5.2.

5.1 CUDA

Unlike OpenGL, the Compute Unified Device Architecture (CUDA) was not designed specifically for graphical applications. Its purpose is to serve as an easy-to-use API for general purpose computing on the GPU. CUDA was developed by NVIDIA in 2007, because of the increasing demand for a flexible, powerful GPU-API amongst the High Performance Computing community. Other APIs like the Open Computing Language (OpenCL, released in 2008) are equally suited for our purposes, but CUDA has been chosen for implementation in this thesis because of its ease of use and its broad acceptance amongst the community. One drawback of using CUDA is its limitation to NVIDIA GPUs, so the implementation used here will not run on non-NVIDIA graphics cards like ATI models. It should be possible to adapt the code to the OpenCL API if needed.

5.1.1 Kernels

The CUDA API allows the user to execute functions, called kernels, in parallel on the GPU. These are ordinary C-like functions which are marked through a simple keyword to be recognized by the compiler. The code running on the CPU, which is referred to as host code, can call these kernels with a specific kernel configuration which defines how many threads should be issued on the GPU. These threads are grouped as blocks within which each thread has a unique threadID. The blocks in turn are grouped into a grid, where each block has a unique blockID. Both IDs as well as the grid and block dimension are accessible in the kernel function, which enables the programmer to give each thread a different task.

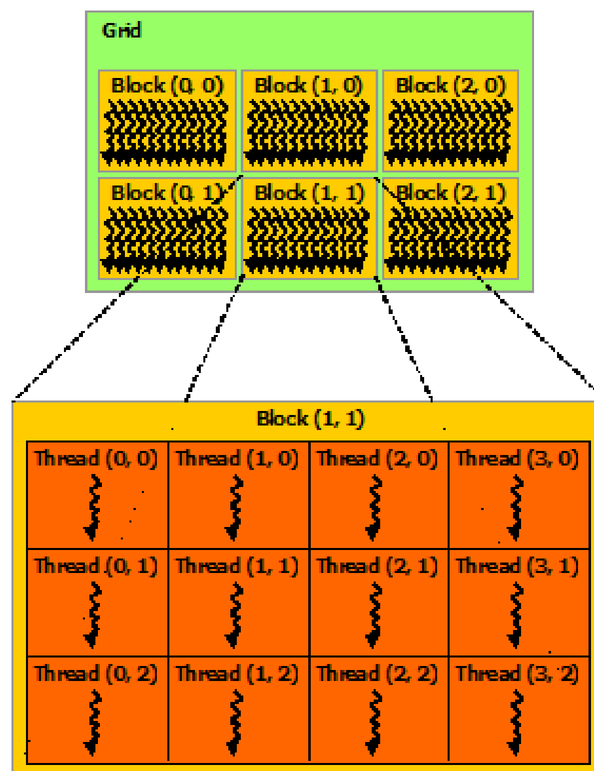

Figure 5.1.1: CUDA kernel configuration (source: [Cor13])

To understand how to use these kernel execution parameters, it is helpful to look at the CUDA architecture. The architecture described in this thesis is called the Fermi architecture, which differs from other architectures like the newest Kepler architecture mainly with respect to the number of cores, the size of caches, the data transfer speed and the clock frequency.

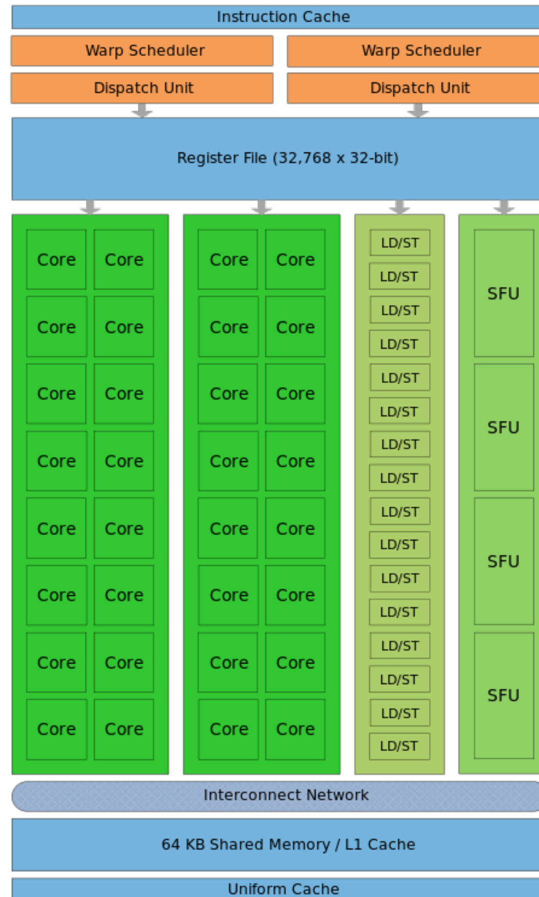

Figure 5.1.2: CUDA Fermi architecture (source: [Cor09])

A CUDA-capable GPU is subdivided into several Streaming Multiprocessors (SMs), each of which contains several Scalar Processors (SPs), often referred to as cores. For example, the GPU used for this thesis contains 14 SMs with 32 cores each. It is important to know that these 448 cores are not independent of each other. Rather, there are 14 independent processors that can perform 32 similar computations at a time, respectively.
When executing a kernel, each thread block is assigned to an SM, whereas the threads themselves are each scheduled to execute on one of its cores. The blocks are executed independently of each other, so the result of the program should not depend on a certain order in which the blocks have to be called. Any dependency between blocks requires synchronization, which means some blocks will have to remain idle while waiting for others to finish their task.

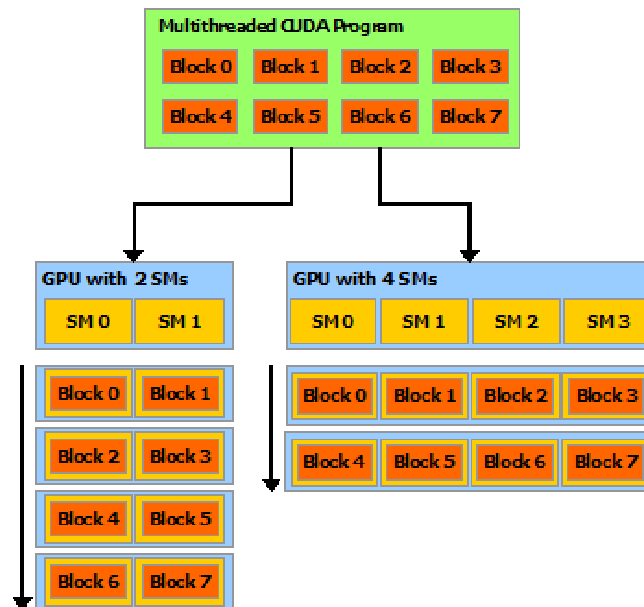

Figure 5.1.3: CUDA block scheduling (source: [Cor13])

The threads themselves are grouped into warps, a set of 32 threads, which execute kernel code in a Single Instruction Multiple Data (SIMD) fashion. This means that the instruction given in the kernel is executed on each core within a block simultaneously, usually with different data. A typical SIMD calculation is vector addition, where for each dimension one core can add the respective values.
If the cores in one warp do not execute the same instructions, for example because they are separated in two groups by an if-statement, some of the cores have to wait for the others to finish execution of their if-branch. Thus, if-instructions and the like should be avoided as much as possible to avoid idling cycles.
Apart from the processor layout, it is crucial to understand the memory structure on the GPU, as data transfer needs to be handled with care due to its significant latency and transfer times.

5.1.2 Memory structure

When writing a CUDA kernel that handles a lot of data, careful consideration of the used data structures and available memory types is advised. Every core has access to several different types of memory, each of which is designed for a distinct usage. A general rule applies: The faster the storage, the smaller its capacity.
Each SM has several registers at its disposal that are distributed among all threads in the block assigned to this SM. Therefore, the more threads are launched in a block, the less registers are available per thread. They are used for variables declared in the kernel code and access to them is instant. For dynamic arrays, other large data structures and variables that do not fit into the acquired number of registers, local memory is used. Locality in this case is not spacial, but strictly logical. While it can solely be accessed by the core assigned to it, local memory is actually off-chip memory which has a very high latency and low throughput, so its usage should be avoided as much as possible. Hence, issuing fewer threads can sometimes deliver higher performance, because less variables per thread need to be sourced out to local memory. NVIDIA provides a so-called occupancy calculator that can suggest an optimal number of threads per block given the hardware details and registers used per thread ([cor]).

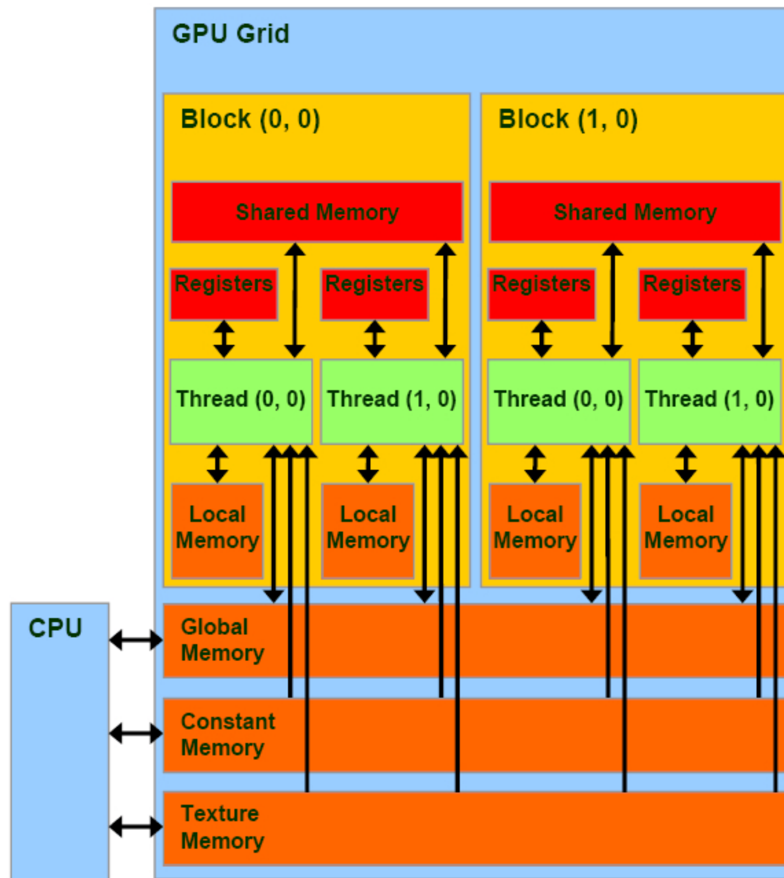

Figure 5.1.4: CUDA memory hierarchy (source: [Cor13])

Another very fast on-chip storage is shared memory, which has only a few cycles of access latency and a high throughput. There are usually a couple of KB of shared memory per SM which can be accessed by all threads in a block. It can be used for data exchange between the cores or for caching global memory data that is used frequently among the threads. Shared memory is subdivided into memory banks, where each bank can only access one dataset at a time, which means accesses of different cores to the same bank have to be serialized, causing a so-called bank conflict. If each thread accesses successive values in shared memory, they can be accessed all at once.
Apart from shared memory, every SM additionally contains caches for constant memory and texture memory. Both of these memories are read-only and are located off-chip which makes access to them very slow. Thus, storing frequently accessed constant data in constant or texture memory can gain performance compared to using global memory. Texture memory is useful especially for data structures that are accessed with spacial locality, whereas constant memory is used for general data.

The slowest GPU memory is global memory. It is stored off-chip and its latency can add up to a couple of hundred cycles while delivering a throughput of approximately 90 GB/s, depending on the architecture. The CPU can only read and write to off-chip memory, so this is the only possibility for data exchange with the GPU. Consequently, global memory is mostly used to receive data from the CPU that the different cores should work on. After kernel execution has finished, the results can be transferred back to the CPU from global memory. Data transfer to and from global memory should be minimized for both CPU- and GPU-accesses. The CPU can transfer data from main memory to global memory using the PCIe bus, which can have a transfer rate between 1 GB/s and 32 GB/s.

Because there is always some latency for invoking a data transfer, reads and writes should be combined as much as possible. The same is true for global memory accesses through the GPU-cores, although here it is especially important to issue coalesced reads and writes, meaning the values accessed by the cores should be contiguous. In case of random access, every fetch instruction would have to be issued separately. This way, the GPU would handle these fetch instructions sequentially which would yield an increased latency overhead and waste bus transfer space, since always at least 32 bytes are being transferred ([Har13]), regardless of the size of the requested data.
The above two sections give a rough overview of the most important things to look out for when programming in CUDA. However, this summary was reduced to information relevant for the optimization of the object tracking algorithm described in this thesis. A far more detailed description of the CUDA memory structure and hints for optimization can be found here: [Cor13].

5.2 Parallel likelihood computation

The likelihood of the pose given the observation equals the product of the likelihoods of each pixel. This can also be expressed as the sum of the logarithms of the pixel likelihoods, which is what we use for numerical reasons.
The likelihood of a pixel with a rendered depth of 1.0, which is the default value, is set to zero. This means, that pixels which are not occupied by the rendered object are neglected. Thus, the likelihood computation only has to be executed for a fraction of the pixels of each rendered pose. All of these calculations are independent of each other which is why they are optimal for parallelization. To achieve maximum performance with a CUDA kernel, it is important to distribute the workload evenly among the GPU cores. As mentioned above (Section 5.1.1), kernels are run in blocks of threads where each group of threads should preferably perform the same computations applied to different data. While the likelihood calculation is similar for each pose, the pixels occupied by the object differ between them. Thus, when computing one pose per thread, the threads would branch differently on several pixels which would make them wait on each other. Therefore, better performance can be achieved by evaluating one pose per block, within which one or rather several pixels are evaluated per thread.
Just one pixel per thread would perform badly, as every thread needs a certain amount of registers for local variables and the number of registers per multiprocessor has to be distributed among all threads within it. All values that do not fit into the registers have to be stored in slow local memory. Hence, the kernel should be able to evaluate a couple of pixels successively. The abbreviations listed in Table 4.1 are used to describe the mode of execution for each instruction in the listing below.

```
48 (G, p)   pixel_nr = threadId
49 (G, p)   while pixel_nr is smaller than #pixels
50 (G, p)       load rendered depth
51 (G, p)       if rendered depth is different from the default then
52 (G, p)         call compute_likelihood()
53 (G, p)       end if
54 (G, p)       pixel_nr += #threads
55 (G, p)   end while
```

Listing 5.1: Weighting kernel in abstracted form

Depending on the amount of threads, each thread will evaluate $\lceil \frac{\#pixels}{\#threads} \rceil$. For example, with a resolution of 80 x 60 and 128 threads, this would make 38 pixels per thread. It is crucial that the workload within a warp is distributed evenly among its 32 threads. A

worst case scenario would be giving each thread 38 subsequent pixels (see Figure 5.2.1). This would leave some threads completely idle, while others have to perform most of the computations, because the relevant pixels are all next to each other. Additionally, the threads could not coalesce their global memory accesses, as for this, thread 0 would have to access pixel 0, thread 1 pixel 1, etc..

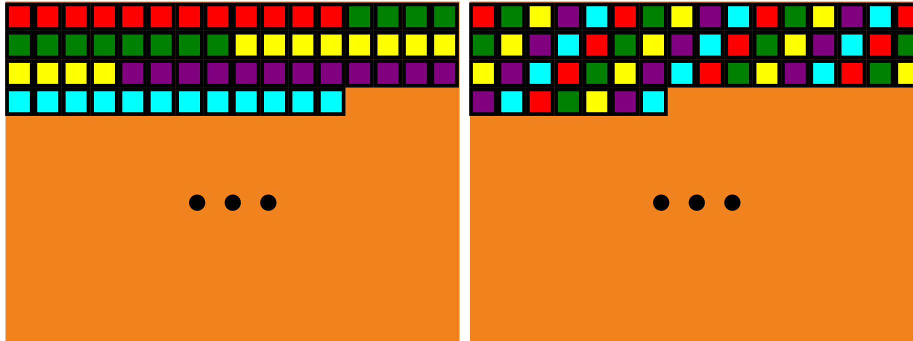

Figure 5.2.1: Kernel workload balancing: Assigning consecutive pixels to a thread (left) causes only few threads to do the main work while others are idle. Instead, subsequent pixels should be assigned to subsequent threads (right).

A good approach is to distribute subsequent pixels among subsequent threads. However, there might still be some room for improvement in this workload balancing, for example by distributing adjacent pixels in both x- and y-direction among subsequent threads.

Attention also has to be paid to internal data transfer on the GPU. The following data is used in computing the likelihood for a pose:

- rendered depth
- observed depth
- occlusion probability
- occlusion update time
- likelihood

The OpenGL texture containing the rendered depth values is automatically allocated in texture memory by CUDA. This memory is read-only and can be cached by each multiprocessor. However, the implementation cannot benefit from this cache, since every depth value is only read once.
The observed depth has to be copied from main memory to global or constant memory every frame. Since this data is independent of the number of poses, the transfer does not carry much weight. Constant memory is read-only and can be cached as well, but the values are only requested once per pose. Depending on the hardware, there might be an L2 cache that contains all observed depth values once the first block has finished execution. Subsequent blocks can then read these values and save the global memory accesses.

Both occlusion arrays are initialized with a default value in the beginning and reside permanently in global memory, while being updated by the threads every frame. The occlusion probability for a pixel depends on its previous occlusion probability and the time that has passed since the last update. Therefore, the algorithm uses two arrays, one to store the probabilities and one to store the last update time of that pixel. This approach

Table 5.1: The relevant data structures and their properties

Data	Memory used	#Values
rendered depth	OpenGL -> texture memory	#pixels * #poses
observed depth	CPU -> global memory	#pixels
occlusion probability	global memory	#pixels * #poses
occlusion update time	global memory	#pixels * #poses
likelihood	shared -> global memory -> CPU	#poses

avoids unnecessary computations, as for each frame only the probability of the occupied pixels needs to be recalculated. Updating these values requires the cores to have write access to them, which is why they have to be stored in global memory.

The likelihoods of all pixels of a pose should be aggregated before they are transferred to global memory, thus reducing data transfer to off-chip memory and, eventually, to the CPU. As this value needs to be accessible for all threads in a block, it is best stored in shared memory. After having finished execution, each thread can request exclusive access to it and add its accumulated likelihoods to it. Eventually, one thread has to copy the value to global memory, from where it can be transferred to main memory for CPU access. Table 5.1 gives an overview of all relevant data structures and their properties.

As the threads work on subsequent pixels, the global memory accesses per warp are coalesced, which means they can be dealt with in one transfer without serializing these accesses. Additionally, CUDA has the possibility to hide the transfer time by scheduling different warps for execution while waiting for the data. This requires that there are enough warps available per block, which is why a number of 128 - 256 threads per block usually yields a good performance. Naturally, the amount of threads should always be a multiple of the warp size.

The data structures used to track occlusion reside on the GPU permanently and are updated for each pose in each time step. In the resampling phase, some of the particles are discarded, making their occlusion information abundant. Others are selected multiple times, thereby requiring copies of the respective occlusion values. This copying of data was implemented with CUDA as well and requires another two arrays that store the copies.

A problem arises from the limited global memory space on the GPU. The OpenGL texture and the two arrays for occlusion grow with an increasing number of poses as well as with a higher resolution, which is why the memory presents a bottleneck in this approach. To alleviate this problem, a second version of the kernel was implemented, which does not use the occlusion update times per pixel. Instead, a global update time is saved and the occlusion probability for all pixels is updated each frame. This massively increases data transfer to and from global memory, which is why this solution introduces a performance penalty, especially when using high resolutions. In exchange, more poses can be evaluated per frame; a more detailed comparison of the two solutions can be found in Section 6.3.

To conclude, the following pseudo-code gives an overview of the likelihood computation. The abbreviations listed in Table 4.1 are used to describe the mode of execution for each instruction.

```
56 (C->G)   send observations to GPU
57 (G, s)   map OpenGL framebuffer texture
58 (C, s)   call likelihood kernel with <#poses, #threads>:
59 (G, p)     for each thread
60 (G, p)       pixel_number = thread_id
61 (G, p)       while pixel_number is smaller than #pixels do
62 (G, p)         load rendered_depth
63 (G, p)         if rendered_depth is different from the default then
64 (G, p)           load occlusion_probability
65 (G, p)           load occlusion_update_time
66 (G, p)           load observed_depth
67 (G, p)           likelihood += call compute_log_likelihood(observed_depth,
                        rendered_depth, occlusion_probability,
                        occlusion_update_time)
68 (G, p)         end if
69 (G, p)         update occlusion probability
70 (G, p)         pixel_number += #threads
71 (G, p)       end while
72 (G, s)       add likelihood to likelihood of other threads
73 (G, p)     end for
74 (G, s)     copy likelihood to global memory
75 (G, s)   end kernel
76 (G->C)   copy likelihood to CPU
77 (G, s)   unmap OpenGL framebuffer texture
```

Listing 5.2: Parallel likelihood computation with a CUDA kernel

5.3 Combined rendering with OpenGL

To make CUDA interoperation easier, all poses have to be rendered into one large texture, instead of many small ones. Allocating this texture should be done in the initialization phase, as allocation is a very costly operation. Later on, this memory space can be reused every frame. This texture is subdivided into rectangular areas of the size of the resolution (e.g. 80 x 60 pixels), of which each rectangle will contain a rendered pose after the drawing process.
This process, which includes the vertex- and the fragment-shader execution, is conducted the same way as described in Section 4.2. To avoid drawing to the whole texture, the so-called viewport has to be set previously to issuing the draw call. The viewport specifies a rectangular area within the framebuffer texture that should be drawn to.
By adding an offset to the viewport for each pose, all poses can be rendered adjacently into the texture. The maximum dimensions of a texture depend on the hardware used, in our case they are 65536 x 65536 which would allow 894,348 poses to be rendered at the resolution of 80 x 60, which should be more than enough. It is more likely to reach the memory limit of the GPU before reaching the maximum allowed dimension of the texture. It does not matter in what pattern the poses are arranged, as long as this information is communicated to the CUDA kernel. In our case, the poses are arranged quadratically. The easiest way to assign the CUDA blocks to the corresponding poses is to execute the kernel in a two-dimensional grid that follows the same pattern.

Since the texture has to be read by the CUDA threads, it has to be mapped into the CUDA context, meaning OpenGL has to provide a pointer to the texture to CUDA. Additionally, the texture has to be detached from the OpenGL framebuffer to guarantee that OpenGL will not modify it while it is read by CUDA.
All in all, this combined rendering approach is not only easier for interoperation, but also faster than rendering each pose into a separate texture, as is explained in the next section.

5.4 Performance and bottlenecks

The combined CUDA + OpenGL approach achieves an average runtime of 1.02 ms for 200 poses for the evaluation step, which is a strong improvement over the pure OpenGL approach with 12.84 ms. A speedup of 23.3 can be observed with respect to the original algorithm when evaluating 200 poses. Taking into account the propagation and resampling steps and obtaining the observation from the camera, the speedup decreases to 11.3 for the overall runtime. This is due to the fact that obtaining the observations takes almost as long as the rest of the algorithm. With an increasing number of poses, this constant runtime becomes irrelevant and the speedup rises to as much as 41.4 for 12800 poses.
A detailed analysis of the internal runtimes of the rendering and the weighting, as well as scalability tests of this combined approach can be found in Section 6.3. To summarize, the major improvement results from following two key principles:

1. Avoiding data transfer as much as possible
2. Aggregating the rendering of all poses for a bigger workload

The data transfer to the CPU, which constituted the main bottleneck in the pure OpenGL approach, was circumvented by parallelizing the weighting step with CUDA, such that the data can stay in GPU memory. Additionally, the slow, sequential filtering of the depth values is performed by the CUDA kernel in parallel. The aggregation of all poses in one texture also improves the runtime significantly, as it avoids synchronization with the CPU in between rendering each pose.

Extensive testing of the weighting kernels described in Section 5.2 was conducted and is presented in Section 6.3. Different kernel configurations can yield significantly different runtimes, which is why such tests are important to find the optimal number of threads for executing a kernel. In this case, 128 threads yield the best result, as they are sufficient to hide memory access latencies and not too many to run out of registers per thread.

All in all, this combined approach allows to evaluate up to 11,293 poses in real-time and scales very well with the number of poses and the resolution, as is discussed in the next chapter.

6. Evaluation

The two parallelized implementations of the object tracking algorithm were tested extensively with regard to their internal runtimes as well as their overall runtime and scalability. The results and a detailed analysis thereof are presented in this chapter.
All tests were conducted on the system described in Section 6.1. The two sections after it analyse the internal runtimes of the pure OpenGL approach and of the combined OpenGL + CUDA approach, respectively. Subsequently, both approaches are compared to the original approach in regard to the scalability with the number of poses and the resolution. Finally, the scalability with the number of triangles and the number of cores is tested for the combined approach in Sections 6.6 and 6.7.

6.1 Work environment

In this section, the software requirements needed to execute the algorithms are specified. Additionally, the hardware configuration and standard parameters used for testing are enumerated.

Software requirements:

- Ubuntu 12.04
- Robot Operating System (ROS) Groovy
- CUDA Toolkit and NVIDIA driver of Compute Capability 2.0 or higher
- OpenGL version 3.2
- Drivers for the Kinect sensor (e.g. OpenNI)

Performance results are very dependent on the hardware that is used to run the program. The results presented in this thesis are based on the following off-the-shelf hardware:

- CPU: Intel Core i5-3550, LGA1155
- GPU: 1280MB Gainward GeForce GTX 560 Ti 448 Cores Limited Edition, PCIe 2.0 x16
- Motherboard: MSI B75MA-P45
- RAM: 8GB-Kit Corsair Vengeance DDR3 1600 MHz CL9

- Xbox 360 Kinect sensor

Increased performance should be expected with newer graphics cards.
To compare the original tracking algorithm to its parallel versions, the following parameters are used in the presented tests unless otherwise mentioned:

- Number of poses: 200
- Resolution: 80 x 60
- Object: A handset (see Figure 6.1.1) with 210 vertices and 420 triangles

Figure 6.1.1: The handset object used for testing

Although the tests have been performed on this relatively simple object, more complex objects can be tracked without a noticeable difference in runtime.
The number of poses and the resolution was selected in this way, because this is close to the maximum that the original algorithm can handle. The resolution is very low, but sufficient for a robust object tracking with these algorithms.
In general, all runtime measurements have been obtained by running at least 500 iterations of the respective function.

6.2 Runtime analysis of the OpenGL approach

The OpenGL approach achieves an average runtime of 48.47 µs per pose for the rendering step which compared to the 102.92 µs needed by the original algorithm yields a speedup of 2.12. To shed light on the bottlenecks of this approach, this section analyses the internal runtimes of the implemented rendering with OpenGL.

Timing an OpenGL application is not straight-forward. While the execution time for the entire pose evaluation can be measured with a CPU timer that is started before and stopped after calling the function, the timing of individual OpenGL calls requires a different construct. Most of these calls are asynchronous, for example **glUniformMatrix3fv()**

that sends a matrix to the shaders. They submit a command into the OpenGL command queue and immediately return to the CPU. Without waiting for the GPU to finish execution of this command, the CPU can continue executing the host code in parallel. Thus, the CPU timer cannot yield useful information about the runtime on the GPU. OpenGL provides so-called query objects ([Use13]), that are able to start a GPU timer once the OpenGL command starts executing in the queue and stop the timer once the command has finished. These query objects were used for timing OpenGL calls in the scope of this thesis.
Obtaining the relevant depth values for a pose consists of five steps:

1. Sending the matrices to the shaders
2. Resetting the framebuffer to the default
3. Rendering the object
4. Transferring the depth values back to the CPU
5. Filtering the depth values

Besides these steps, a couple of buffers on the GPU have to be allocated, the object mesh has to be transferred, various OpenGL parameters have to be set and the framebuffer has to be configured. However, all of these steps can be moved to the initialization phase since they only have to be executed once. The following diagram gives an overview of the time measurements obtained for the five steps mentioned above, using a GPU or CPU timer, respectively.

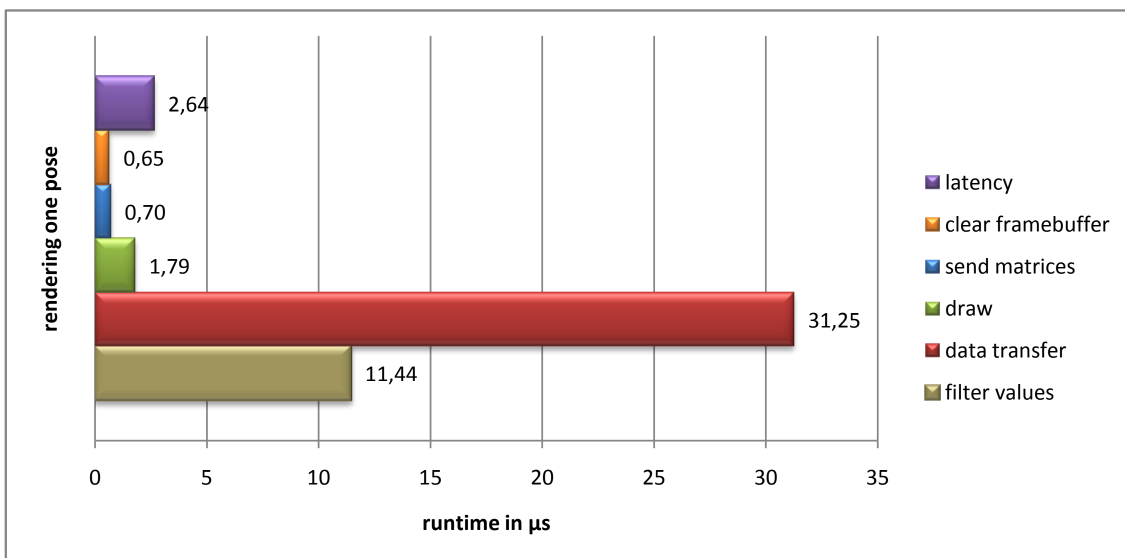

Figure 6.2.1: Runtimes of all steps required for rendering a pose

It becomes apparent by looking at Figure 6.2.1, that the draw process itself is executed very quickly on the GPU. With a total of 3.14 µs, the drawing by itself would deliver a speedup of 32.12. The data transfer is the main bottleneck followed by the sequential filtering on the CPU. The slow data transfer is due to the PCIe bus that connects main memory and GPU memory. It can transmit only 1 to 32 GB/s, depending on the PCIe version, the graphics card and the main memory used.

The transfer speed also varies with the amount of data to be transmitted, since the overhead of invoking the transfer preponderates when sending only a few kilobytes. In our case, the depth data of each pixel is stored in a 32-bit float value in the framebuffer texture and has to be copied to a previously allocated buffer in main memory, to which a pointer is then provided by the GPU. With a resolution of 80 x 60 pixels, 19.2 KB of data have to be transferred per pose. According to the measurements, the transfer speed in our experiments lies around 0.72 GB/s. Unfortunately, this data transfer is unavoidable as long as the likelihood computation is implemented on the CPU, as it needs access to the pixels' depth information.
Additionally, there seems to be an overhead of around 2.64 µs for each pose. This overhead is very difficult to measure and is thus only an approximate value. It has been obtained in this way: First, the program is executed such that it measures the internal runtimes (clearing framebuffer, sending matrices, etc.). As taking these measurements adds some overhead to the overall rendering runtime, the program is run again without measuring internal runtimes. From this second run, the total time used for the rendering can be obtained. The difference between it and the accumulated internal runtimes is assumed to be latency introduced by communicating with the GPU. I have measured it for the graphics card used here with the CUDA Toolkit to be around 3 µs. I assume that submitting the first command into the empty OpenGL queue is responsible for this overhead, as subsequent latencies should be hidden by the previous latency or the execution of commands that are already in the pipeline.

One should not forget that OpenGL and rendering in general is typically used for very complex scenes which generally consist of several millions of triangles. This guarantees a very good utilization of the GPU, while using only a few hundred triangles, as in our case, can lead to idling cycles on several cores. The problem size is thus very small relative to typical GPU applications which makes it particularly difficult to achieve a high speedup. To give an example, the following diagram shows the runtime of the algorithm when using bigger problem sizes.

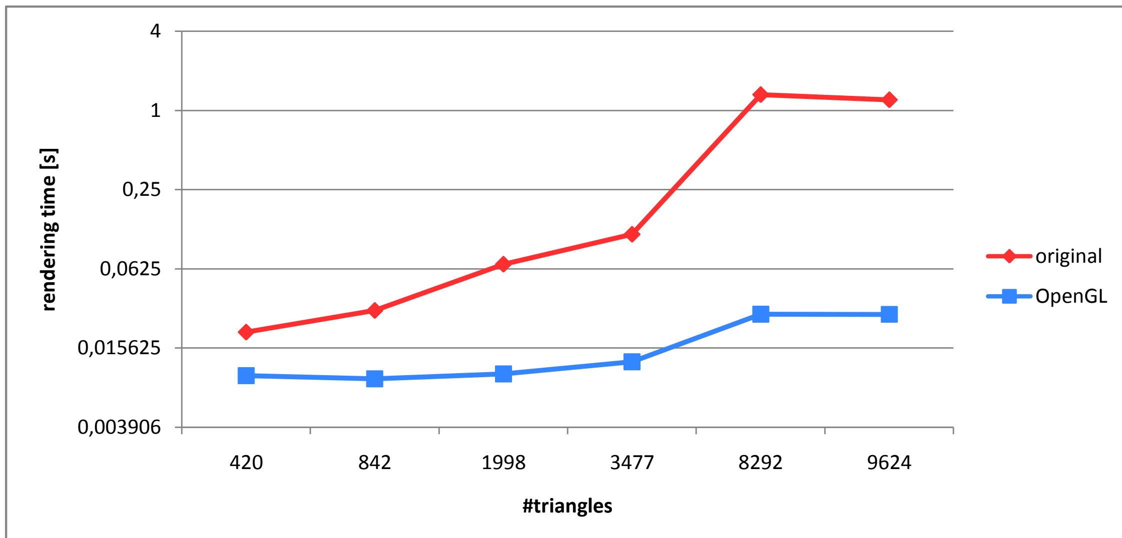

Figure 6.2.2: Comparison of the rendering time needed for the original and the OpenGL algorithm

Figure 6.2.2 shows a considerable superiority of the GPU version when using objects with many triangles. Thus, it is suited to track more complex objects than the CPU version.

However, a significant increase in precision is not to be expected from a more detailed object model, since the differences in depth are negligible. The ability to evaluate more poses is more important, especially when aiming for a higher tracking speed.

6.3 Runtime analysis of the combined OpenGL and CUDA approach

The combined CUDA + OpenGL approach achieves an average runtime of 5.10 µs per pose for the entire evaluation step, which is a strong improvement over the pure OpenGL approach with 64.19 µs (48.47 µs for the rendering and 15.71 µs for the weighting on the CPU). With regard to the 118.63 µs needed by the original approach, a speedup of 23.3 is achieved. Figure 6.3.1 shows a comparison of the runtimes of the three approaches.

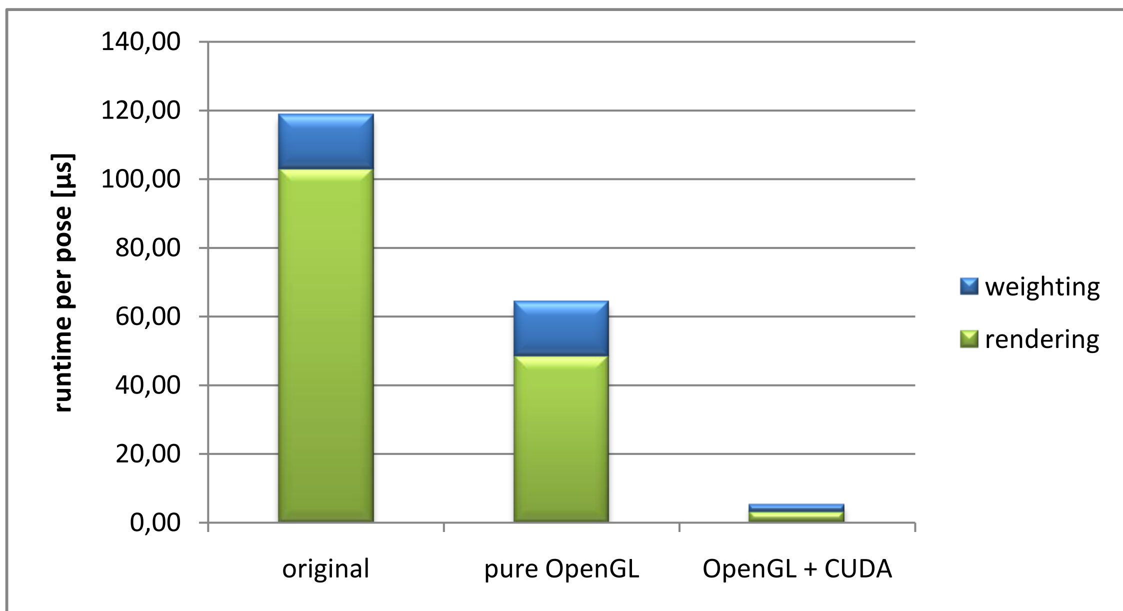

Figure 6.3.1: Runtimes for the evaluation step for 200 poses: The pure OpenGL approach achieves a speedup of 1.8, while the OpenGL + CUDA approach delivers a speedup of 23.3.

As is shown in Section 6.4, the relative speedup increases with the number of poses, allowing to evaluate more than 10,000 poses in real-time. For comparison to the pure OpenGL approach, the internal runtimes of the rendering of both implementations are shown in Figure 6.3.2.

While the data transfer and the sequential filtering of the depth values is avoided by parallelizing the weighting step with CUDA, additional advantages arise from rendering all poses into only one texture. For example, the time needed for clearing the framebuffer textures is reduced significantly as only one large texture needs to be cleared instead of many small ones. In general, the aggregation of all poses into one texture saves a lot of latency that was introduced in the pure OpenGL approach by synchronizing with the CPU in between the rendering of each pose.

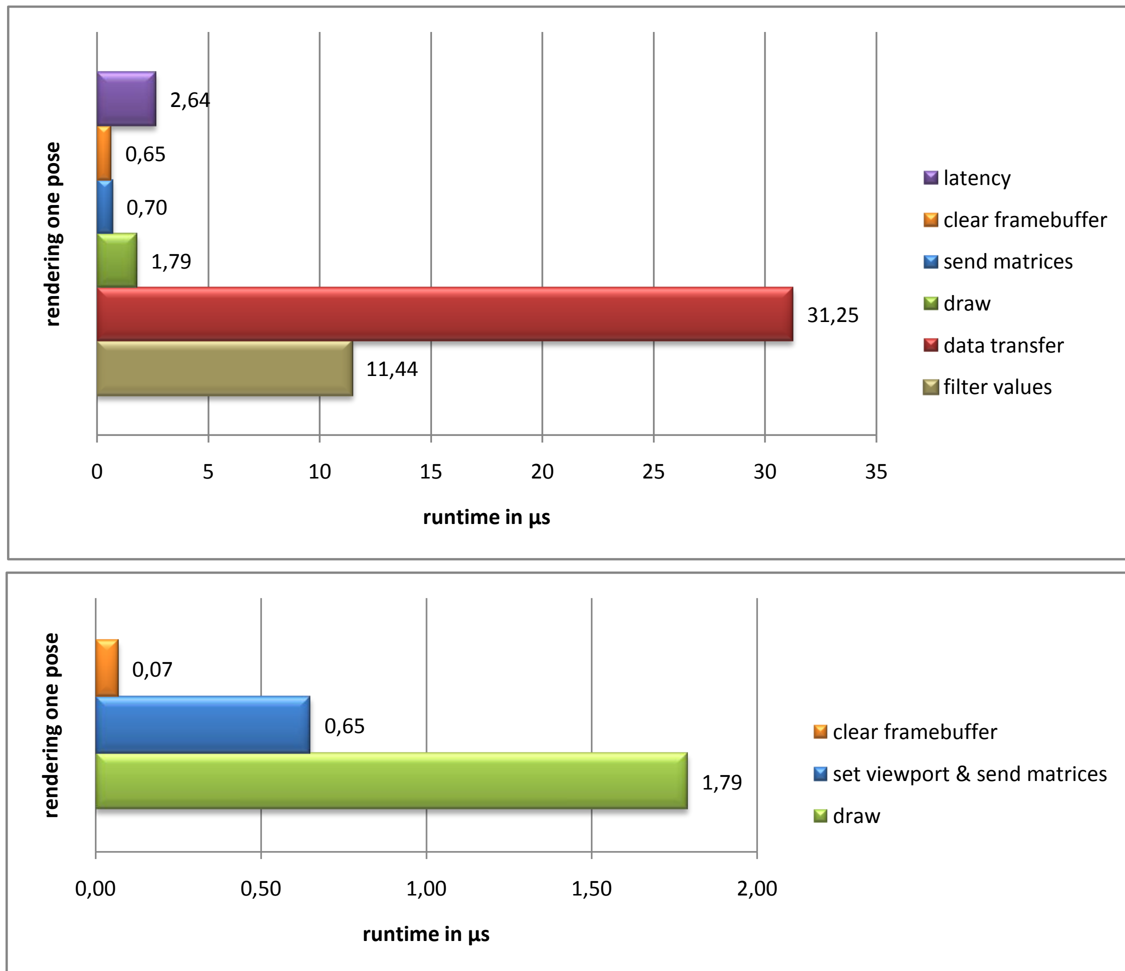

Figure 6.3.2: Internal runtimes for the rendering step for one pose: While the data transfer presents a bottleneck for the pure OpenGL approach (top), the OpenGL + CUDA approach (bottom) avoids this transfer which yields a total runtime for the rendering of 2.51 µs opposed to 48.47 µs per pose.

As can be seen in Figure 6.3.2, the overhead of sending the matrices and resetting the viewport for every pose still has a significant impact on the runtime. The scalability of the rendering is discussed in the following sections.

Mapping the OpenGL texture into the CUDA context takes an average 120 µs, which denotes $\frac{1}{8}$ of the total runtime of the evaluation step for 200 poses. While the mapping cost increases with the number of textures (Figure 6.3.3), it does not seem to increase with the size of the texture, which is another strong argument for using only one large texture.

As the weighting step is performed with CUDA, the observation image needs to be sent to the GPU once per frame and the weights of all particles need to be retrieved after the computation. The transfer times are negligible since merely a few KB have to be transferred.

The CUDA kernel itself is very difficult to time. As with OpenGL, a custom timing function (cudaEvents, see [Cor13]) is provided for measuring the runtime of a single kernel. However, internal timings of a kernel cannot be retrieved with this method, which makes it difficult to find potential bottlenecks. Thus, although the total runtime of the weighting kernel is only 357.83 µs for 200 poses, it is possible that it can be optimized further.

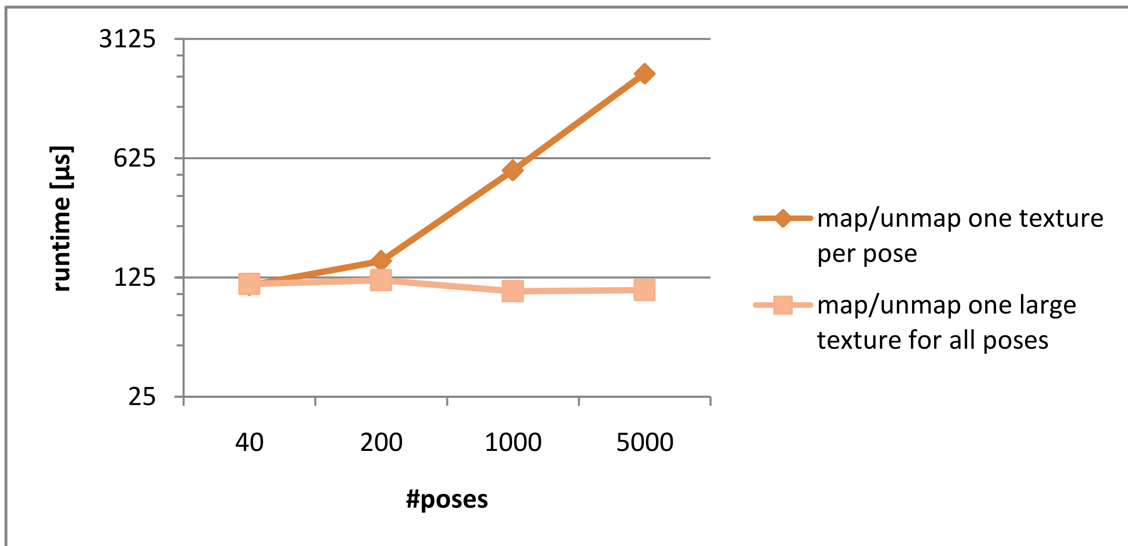

Figure 6.3.3: Scalability of the mapping call with the number of poses: The mapping cost increases significantly with the number of textures, while it seems to be independent of the texture size. Thus, only one large texture for all poses should be used.

Yet, different versions of the kernel have been tested extensively. Different approaches are compared in the following.

One important factor is the distribution of the workload among the cores, as mentioned in Section 5.2. To illustrate this, Figure 6.3.4 shows the runtime of a kernel which distributes the pixels to the threads in continuous blocks. Opposed to this, continuous pixels can be assigned to continuous threads, which does not only result in coalesced memory access, but also in a more balanced workload.

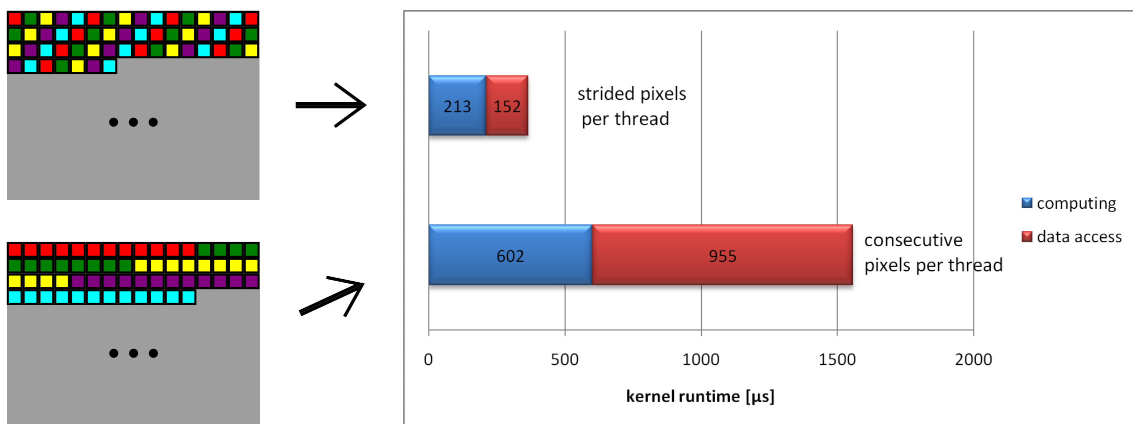

Figure 6.3.4: Kernel workload balancing: Assigning consecutive pixels to a thread results in uncoalesced memory accesses and causes only few threads to do the main work while others are idle.

Besides the workload per thread, the total number of threads per block plays an important role. The more threads are issued, the more flexibility is provided to the CUDA scheduler, which can hide memory access latencies. While the data for one warp of threads is fetched,

e.g. from global memory, another warp can continue computing on the cores. The tradeoff here is with the number of available registers per thread, as these resources are shared among all threads in a block. If a thread has fewer registers than required for its local variables, it has to allocate these in slow local memory (see Section 5.1.2).

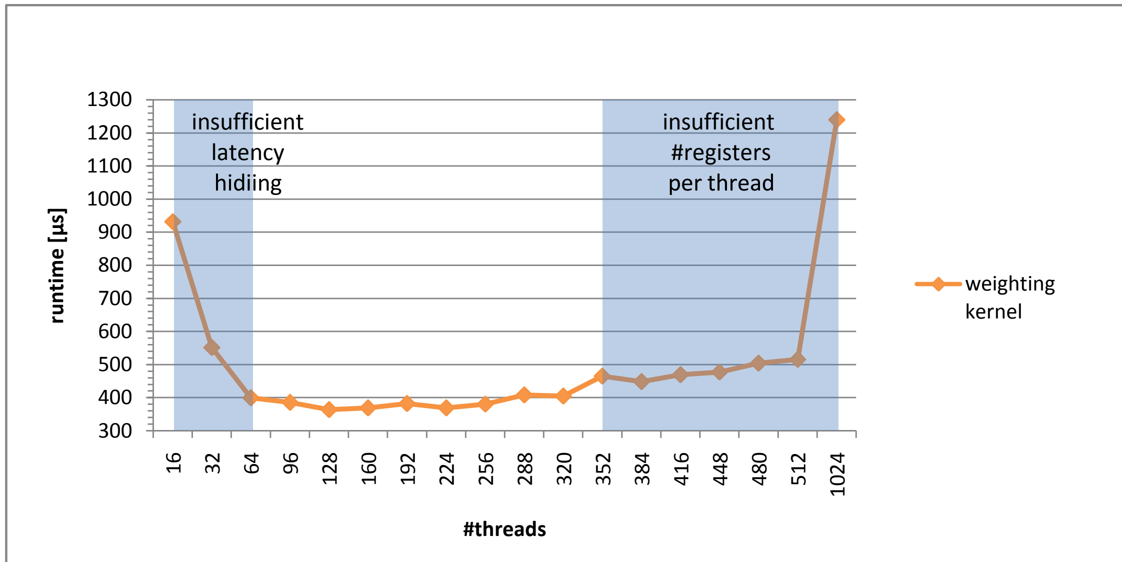

Figure 6.3.5: Runtimes for different kernel configurations: Using less than 64 threads does not provide enough flexibility to the thread scheduler to hide memory access latencies. When using too many threads, the number of registers per thread is not sufficient to incorporate all local variables. Increased data transfer to slow local memory is the result.

While the use of local memory can be avoided, the use of global memory cannot. To incorporate explicit occlusion modeling, the global arrays that contain the occlusion probabilities and the time of their last update, have to be accessed for each relevant pixel. As these data structures have to stay on the GPU permanently, to avoid data transfer to main memory, they have to be stored in global memory, where they can be read and modified by each thread. As mentioned at the end of Section 5.2, two kernels were implemented. One updates every pixel every frame, thereby saving memory space, but introducing heavy data transfer. The other one updates only relevant pixels, which decreases data transfer significantly, but requires twice as much memory space. Figure 6.3.6 gives an overview of both approaches and one that completely neglects occlusions.

Based on the global memory space available on the GPU, the best suited approach should be chosen for executing the weighting step. As in our case, only 1 GB of memory is available, a higher number of particles can be evaluated when using the computationally more intensive kernel. Details about the scalability of both solutions can be found in the following sections.

All in all, the combined CUDA and OpenGL approach evaluates 200 poses in ~ 1 ms, while the runtime is divided among the rendering, mapping and weighting as shown in Figure 6.3.7.

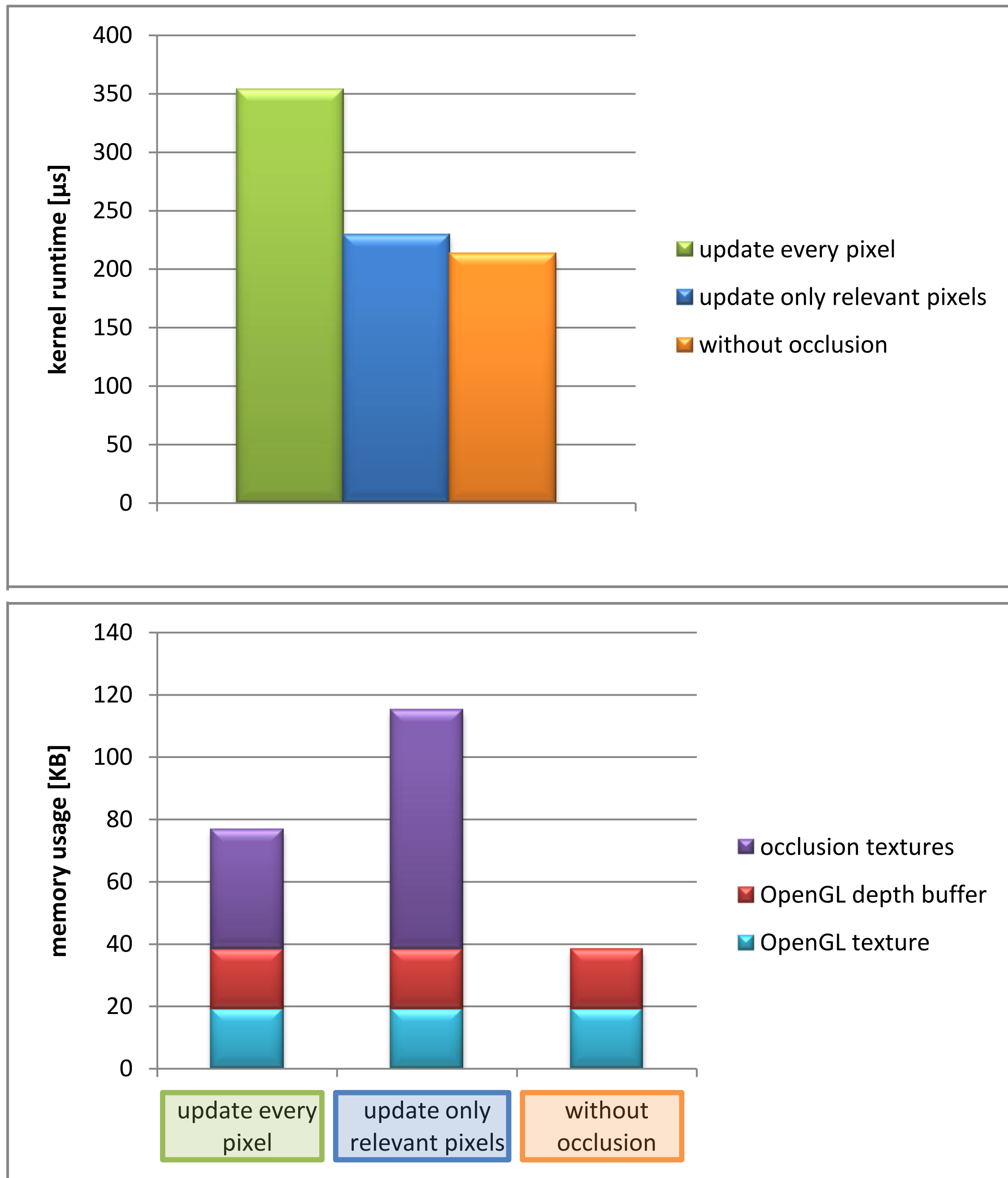

Figure 6.3.6: Tradeoff between runtime and memory usage: Using the same update time for every pixel saves 38.4 KB of memory per pose (with resolution 80 x 60), but necessitates the updating of every pixel each frame. Storing an indiviual update time per pixel increases memory usage, but saves 124 µs for weighting 200 poses. Using no occlusion at all saves a significant amount of memory.

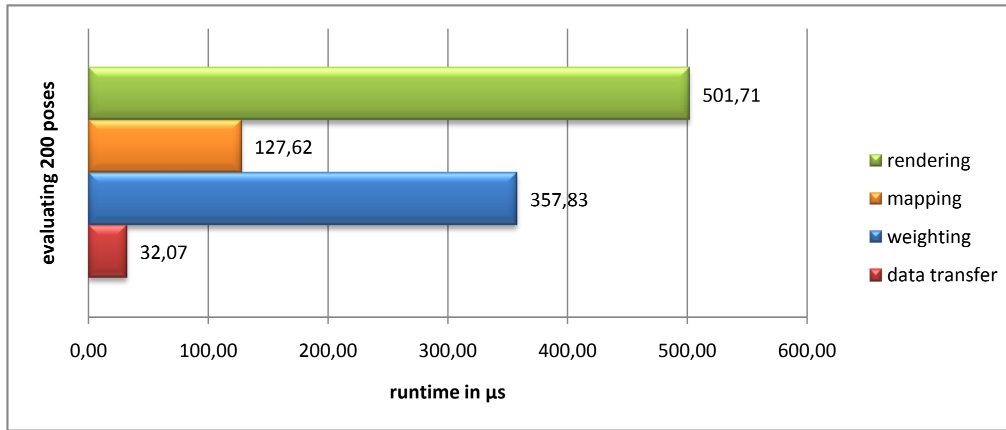

Figure 6.3.7: Runtime distribution of the evaluation step for 200 poses: The majority of the runtime is used for rendering and weighting the poses. The mapping cost between the two frameworks is noticeable, though constant, which makes it negligible when evaluating more poses.

6.4 Scalability with the number of poses

While 200 poses can be evaluated in 2.6 ms with the combined approach, it can handle 10,163 poses in less than 33 ms (see Figure 6.4.1). Compared to a maximum of 230 poses for the original algorithm, this yields a speedup of 44.19. This increase in speedup is due to the initial sublinear scaling of the combined implementation, which indicates that the GPU does not operate at full capactiy until working with more than ~ 1600 poses.

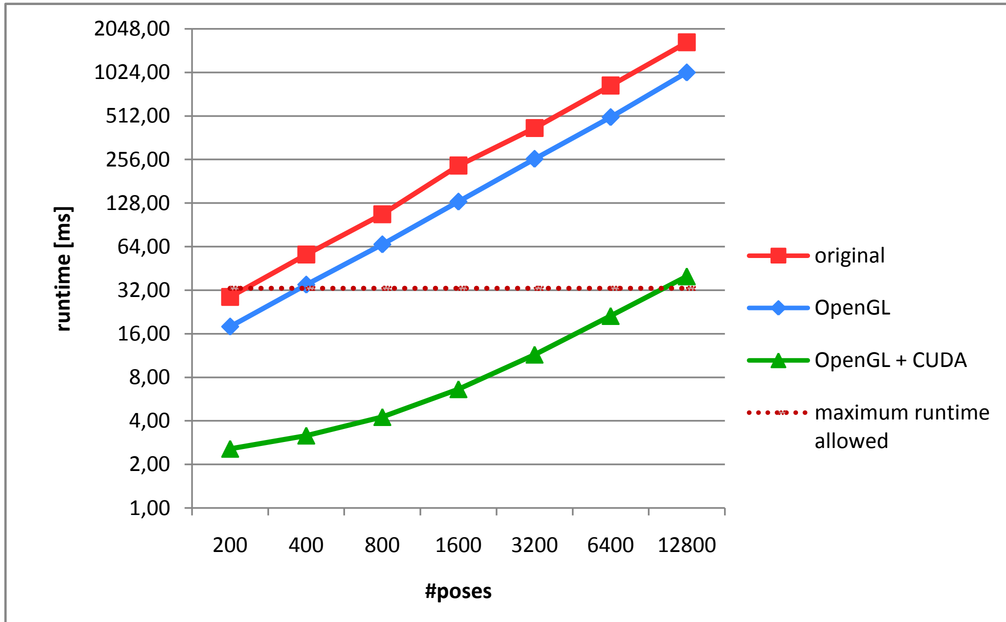

Figure 6.4.1: Scalability with the number of poses: An initial sublinear scaling can be observed for the combined approach. When evaluating 30 fps, the original algorithm allows around 200 poses to be used, while the OpenGL approach can handle ~ 400. The OpenGL + CUDA approach outmatches the two by far, presenting the opportunity to use more than 10,000 poses.

Parallelization of the evaluation step is typically the main problem to attend to when aim-

ing for a higher performance in a particle filter. However, the propagation and resampling step prove to be just as much of a problem when evaluating many particles. As can be seen in Figure 6.4.2, the resampling alone limits the possible amount of poses to ~ 2080, which is why the implemented parallelization of the resampling step is essential for further scalability. The sequential propagation step also reaches a runtime of about ~ 8.5 ms when using 12800 poses, which denotes 26% of the available 33 ms for a real-time performance.

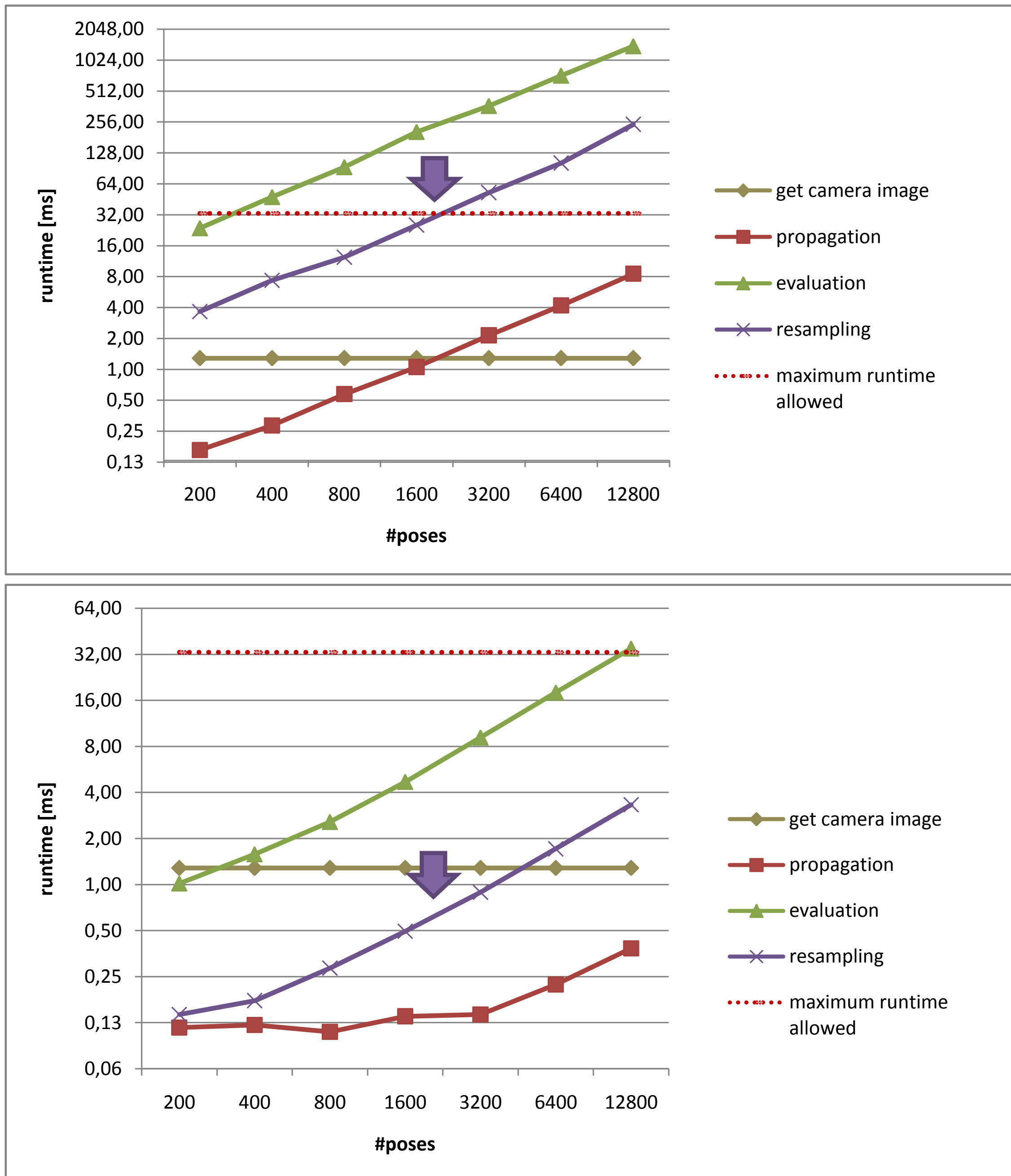

Figure 6.4.2: Scalability of the different steps in the particle filter: The resampling step becomes a bottleneck with an increasing number of poses. It takes 33 ms when using around 2080 particles in the original approach (top). The CUDA + OpenGL approach (bottom) parallelizes all major steps to avoid this bottleneck and decrease the overall runtime. Getting the camera image from the Kinect sensor takes a constant 1.29 ms and cannot be parallelized.

Figure 6.4.3 depicts the speedup that was achieved by the combined approach for every step. While the evaluation, consisting of the rendering and weighting of the poses, is the most computationally intensive step, a significant speedup could also be reached for the resampling step due to a higher internal memory bus speed on the GPU.

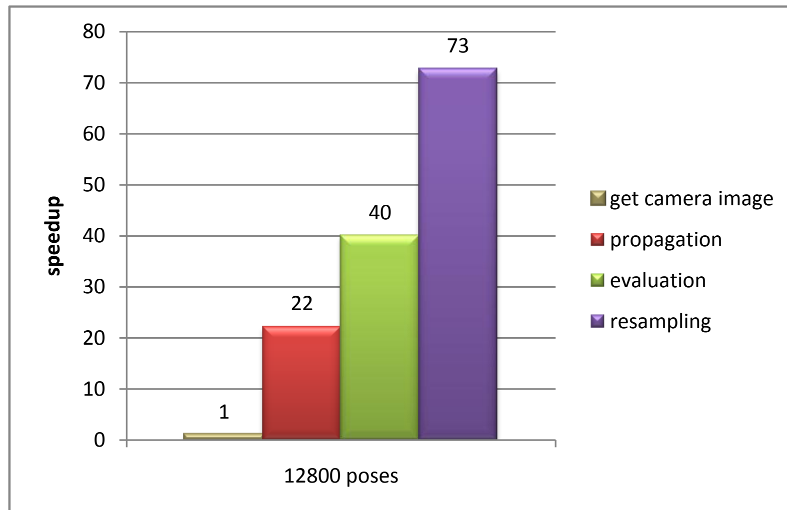

Figure 6.4.3: Speedup per step: With a speedup of 40, the parallelization of the evaluation step is the most important, as it requires 78 % of the overall runtime.

When investigating the internal runtimes of the rendering step, a significant overhead can be observed, caused by resetting the viewport and sending the matrix for every pose. Figure 6.4.4 shows that only half of the time for rendering is actually used by the draw call. Up to this point, a solution to circumvent this problem has not been implemented. A possible approach would be an implementation with CUDA, which allows transforming different vertices with different matrices within a single kernel invocation.

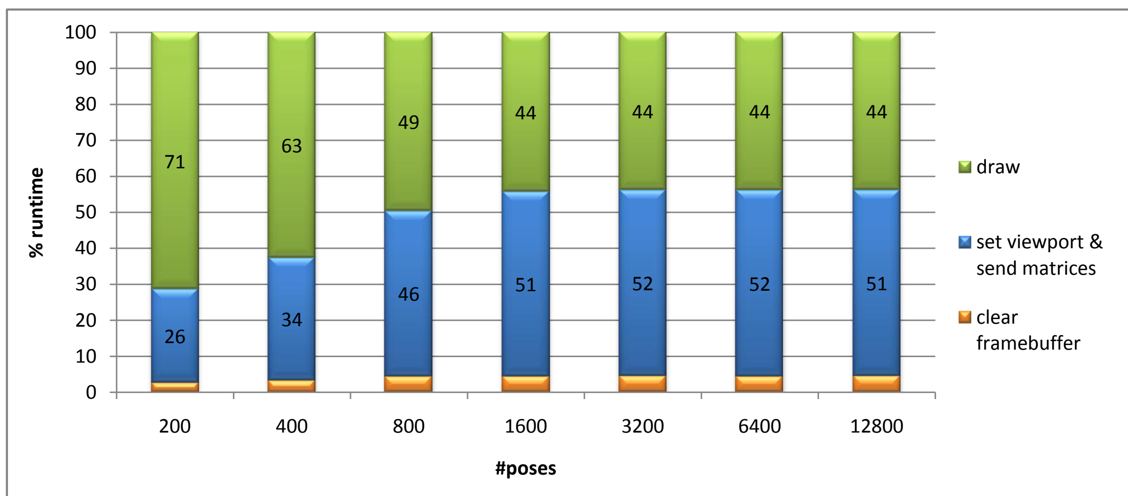

Figure 6.4.4: Internal runtimes of the rendering: The OpenGL + CUDA approach seems to spend as much time on drawing the object as on configuring the matrix and the viewport previously to the draw call.

For the weighting step, two different kernels were implemented. As can be seen in Figure 6.4.5, they both scale linearly with the number of poses. A speedup of ~ 1.8 can be

observed for the kernel that reduces internal data transfer at the cost of consuming more memory space. Thus, this kernel should be used unless the memory space provided by the respective GPU poses a limiting factor. As the GPU used in the scope of this thesis provides only 1 GB of memory, a higher number of poses could be achieved by using the computationally more expensive kernel.

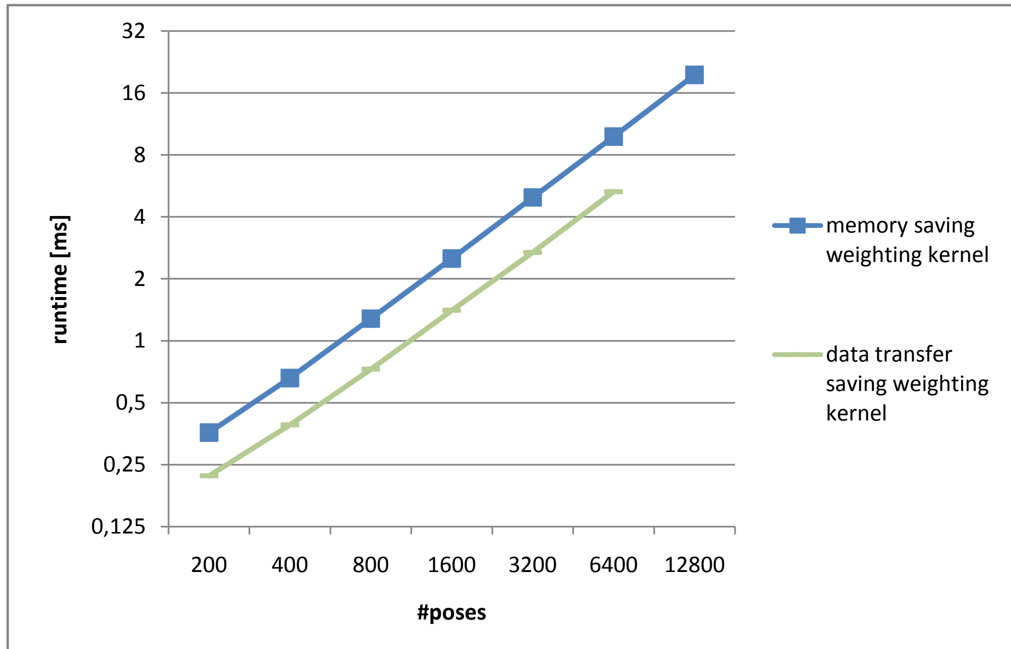

Figure 6.4.5: Scalability of the two different weighting kernels with the number of poses

Overall, the rendering and the weighting step distribute the runtime almost equally among themselves when scaling with the number of poses.

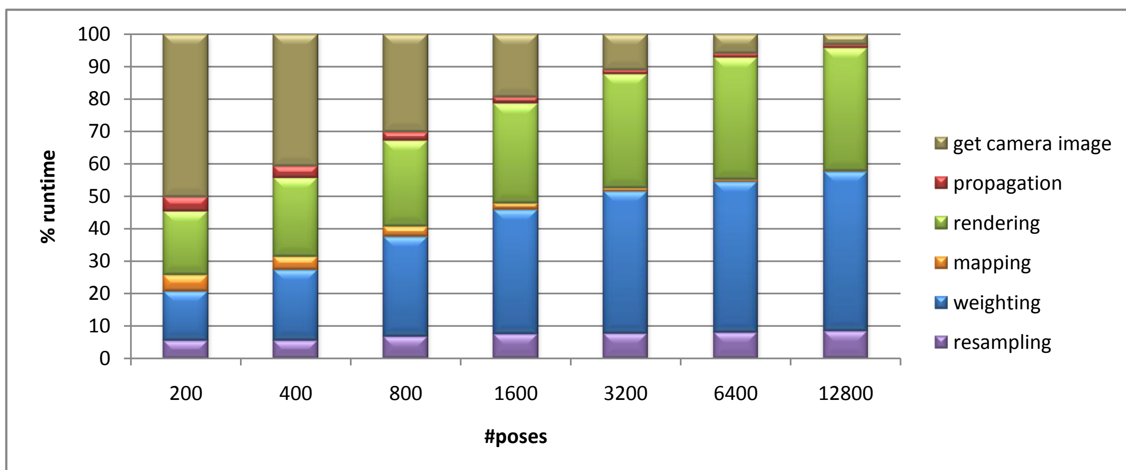

Figure 6.4.6: Runtime distribution among all major steps, splitting the evaluation into rendering, mapping and weighting: While getting the camera image contributes about 50 % of the overall runtime for 200 poses, its significance diminishes when using more poses. The majority of the runtime is distributed almost evenly among the rendering and the weighting of the particles.

6.5 Scalability with the resolution

With a sufficient number of poses, the original approach can only handle a resolution of 80 x 60. In various tracking experiments, this resolution has proven to be sufficient to robustly track the object. However, no detailed information about the precision of the estimated pose can be given, as no ground truth data is available. If desired, the resolution can be increased when using the combined approach, as it is able to evaluate ~ 270 poses on even the highest resolution of 640 x 480. How many poses can be evaluated with each algorithm for different resolutions is shown in Figure 6.5.2.

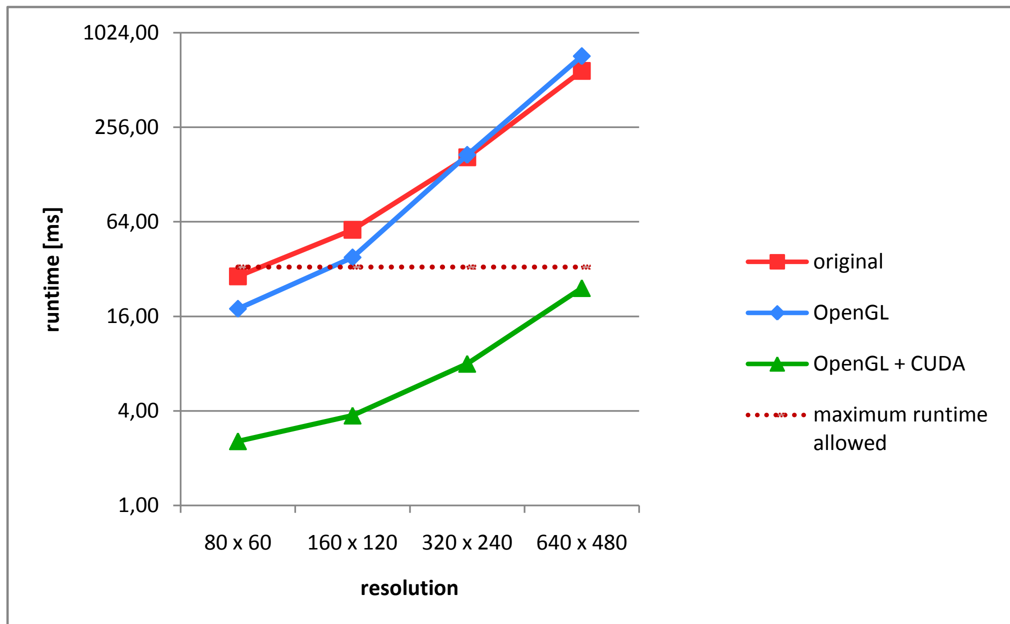

Figure 6.5.1: Scalability with the resolution when using 200 particles: The original and the pure OpenGL approach are limited to a resolution of 80 x 60 and 160 x 120, respectively. The combined approach provides the possibility to evaluate 200 poses on even the highest resolution of 640 x 480.

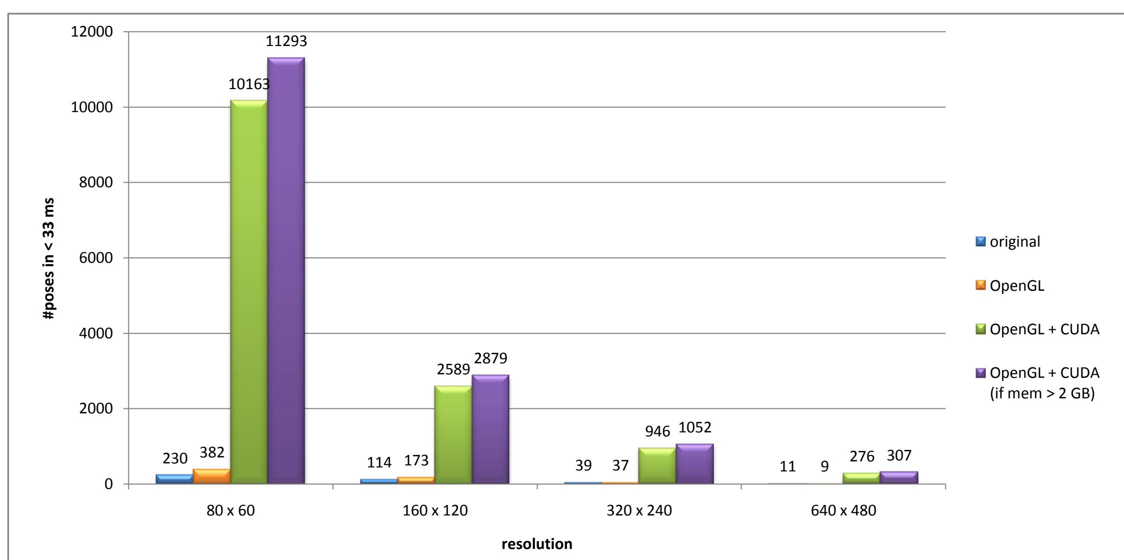

Figure 6.5.2: Possible number of poses with each approach

An interesting aspect of scaling with the resolution is the runtime distribution inside of the evaluation step. While the weighting kernel scales linearly with the resolution, the rendering time does not increase significantly. Figure 6.5.3 illustrates the distribution of the overall runtime among the separate steps. Since OpenGL is optimized to render with high resolutions, as they are frequently used in computer games and the like, this explains the sublinear scalability in this regard. The overall scalability with the resolution is linear.

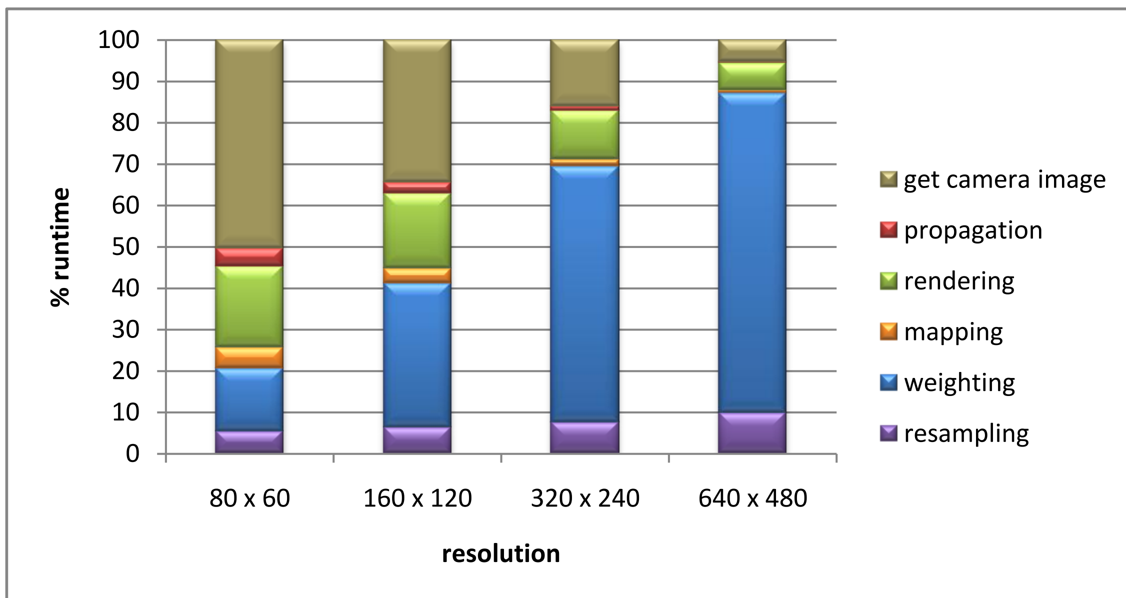

Figure 6.5.3: Runtime distribution when scaling with the resolution: The time needed for rendering does not increase significantly with the resolution, compared to the weighting step.

6.6 Scalability with the number of triangles

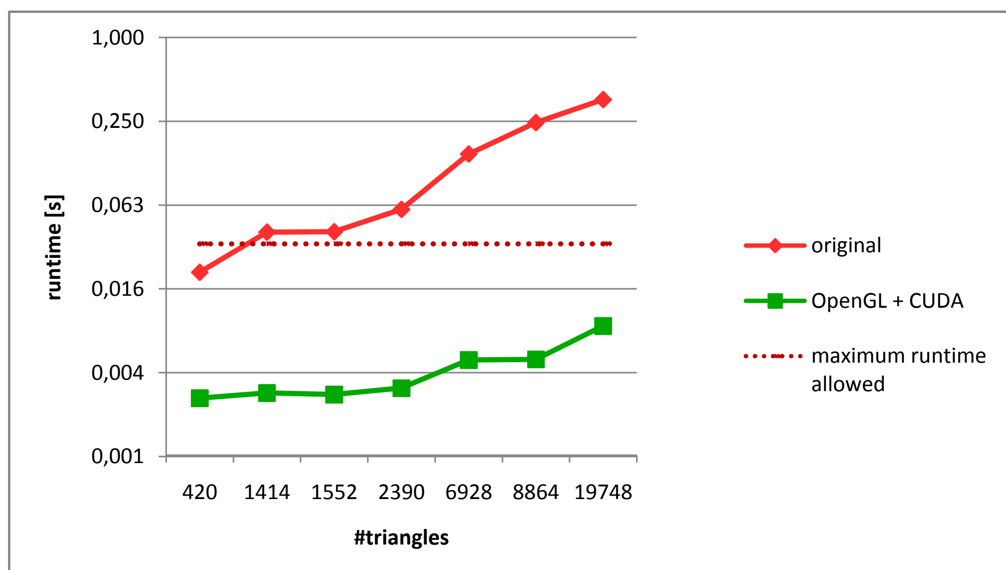

Figure 6.6.1: Scalability with the number of triangles: Using complex object models increases the runtime of the original algorithm noticeably.

The combined approach allows not only for more poses to be evaluated on a higher resolution, but can also handle very complex object models. As Figure 6.2.2 illustrates, several thousand triangles can be handled by the OpenGL rendering engine without significantly increasing the overall runtime. This enables tracking of objects with a highly complex shape.

6.7 Scalability with the number of cores

The combined approach was tested on two GPUs, a NVIDIA Quadro 4000 with 256 cores and a NVIDIA GTX 560 Ti 448 with 448 cores. The results are presented in Figure 6.7.1. The implementation appears to scale well with the number of cores. Naturally, more tests with newer GPUs would have to be conducted in order to predict the scalability with the number of cores. However, a linear scaling with the number of cores is very likely, as they can evaluate additional poses in parallel. Thus, the NVIDIA Titan Black with 2,880 cores can be expected to evaluate more than 40,000 poses in less than 33 ms.

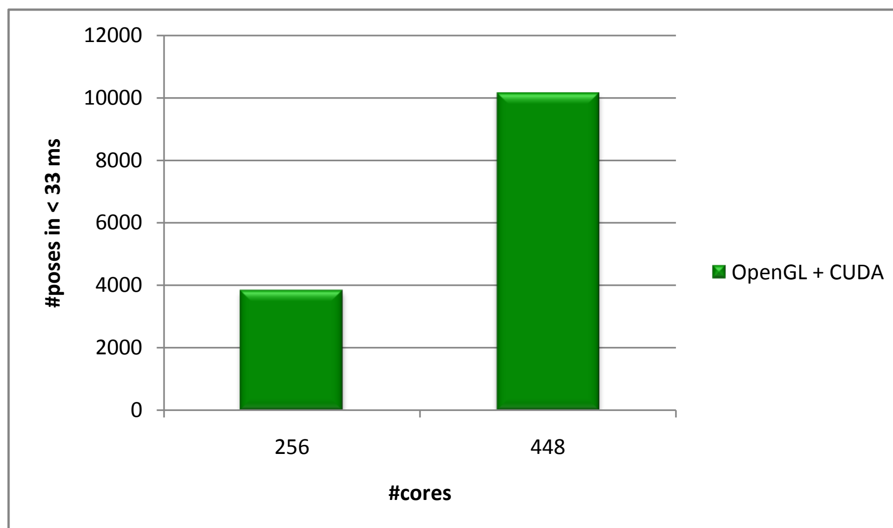

Figure 6.7.1: Scalability with the number of cores: The combined approach was tested on two different GPUs, one with 256 and one with 448 cores.

7. Conclusion

The goal of this thesis was to enable the evaluation of more particles by parallelizing the object tracking algorithm developed by Wüthrich et al. ([WPK+13]). As the majority of computing time was needed for rendering the different poses, a GPU-parallelization promised the best results. Two implementations were developed, of which one is significantly limited by data transfer times, while the other yields a speedup of up to 41.4 by implementing not only the rendering, but also the weighting, propagation and resampling steps on the GPU.
A combination of OpenGL and CUDA was used for this parallelization, as each of them offers a different functionality. While OpenGL is optimized for rendering, CUDA is easy to use for general purpose calculations like the weighting computation. Allowing the use of up to 12,000 particles on an off-the-shelf GPU, the opportunity for a more precise and faster object tracking is provided.
As new graphics cards arrive on the market every year, this implementation will be able to evaluate even more particles in the future. Assuming that the algorithm scales linearly with the number of cores in a GPU, an estimated number of $\sim$ 40,000 particles can be expected from using a GTX Titan Black ([Cor14]), which is a high-end gaming graphics card already available today.
Apart from the amount of particles, the algorithm can handle more complex object models without introducing a noticeable performance penalty. This allows for tracking of objects with complex shapes.

In the future, this algorithm could possibly be further improved by implementing the rendering in CUDA and finding a better workload balancing for the existing weighting kernel. Tracking multiple objects is also possible with the approach, which could be worked on in the future by developing a new sampling strategy. Additionally, the variance for the gaussian propagation distribution can be adjusted to take better advantage of the amount of particles that are available.
Furthermore, the algorithm should be tested with a high-speed camera that can deliver depth images at a rate of 60 Hz or higher. Opposed to related algorithms, this GPU-implementation is capable of taking advantage of such a high frame rate, as around 4000 particles can be evaluated in under 15 ms. This should lead to a significant improvement in tracking the object, as feedback is provided more often, which reduces the uncertainty of the pose estimation.

Bibliography

[AMAD11] Pedram Azad, David Münch, Tamim Asfour, and Rüdiger Dillmann. 6-dof model-based tracking of arbitrarily shaped 3d objects. In *ICRA*, pages 5204–5209, 2011.

[CC13] Changhyun Choi and Henrik I. Christensen. Rgb-d object tracking: A particle filter approach on gpu. In *IROS*, pages 1084–1091, 2013.

[cli] How a graphics card works.

[cor] NVIDIA corporation. Cuda occupancy calculator.

[Cor09] NVIDIA Corporation. Whitepaper: Nvidias next generation cuda compute architecture: Fermi, 2009.

[Cor12] Intel Corporation. *Intel Xeon Phi Coprocessor*. 2200 Mission College Blvd., Santa Clara, CA 95054-1549, USA, 2012.

[Cor13] NVIDIA Corporation. *CUDA Programming Guide*. 2701 San Tomas Expressway Santa Clara, CA 95050, USA, 2013.

[Cor14] NVIDIA Corporation. Geforce gtx titan black, 2014.

[DeV11] Will DeVore. Perspective and orthographic projection, 2011.

[Gro12] Joe Groff. An intro to modern opengl, 2012.

[Har13] Mark Harris. How to access global memory efficiently in cuda c/c++ kernels, 2013.

[IB98] Michael Isard and Andrew Blake. Condensation - conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29:5–28, 1998.

[Kal60] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[LO09] Oscar Mateo Lozano and Kazuhiro Otsuka. Real-time visual tracker by stream processing. *Signal Processing Systems*, 57(2):285–295, 2009.

[LPK08] Claus Lenz, Giorgio Panin, and Alois Knoll. A gpu-accelerated particle filter with pixel-level likelihood. In *VMV*, pages 235–241, 2008.

[MPASF04] Antonio S. Montemayor, Juan José Pantrigo, Ángel Sánchez, and Felipe Fernández. Particle filter on gpus for real-time tracking. In *ACM SIGGRAPH 2004 Posters*, SIGGRAPH '04, pages 94–, New York, NY, USA, 2004. ACM.

[Mün10] David Münch. 6-dof particle filter-based tracking of arbitrarily shaped objects, 2010.

[Ope14] Tutorials for modern opengl (3.3+), 2014.

[Tat13] Alan Tatourian. Nvidia gpu architecture and cuda programming environment, 2013.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[Use13] User:Alfonse. Opengl wiki, 2013.

[WM00] Eric A. Wan and Rudolph Van Der Merwe. The unscented kalman filter for nonlinear estimation. pages 153–158, 2000.

[WPK+13] Manuel Wüthrich, Peter Pastor, Mrinal Kalakrishnan, Jeannette Bohg, and Stefan Schaal. Probabilistic object tracking using a range camera. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013.