EXAMENSARBETE INOM TEKNIK,
GRUNDNIVÅ, 15 HP
STOCKHOLM, SVERIGE 2016

Disc

Approximative Nearest Neighbor Search using Ellipsoids for Photon Mapping on GPUs

MARCUS BERGHOLM

VIKTOR KRONVALL

KTH
SKOLAN FÖR DATAVETENSKAP OCH KOMMUNIKATION

KTH Computer Science
and Communication

Disc: Approximative Nearest Neighbor Search using Ellipsoids for Photon Mapping on GPUs

Disc: Approximativ närmaste grannsökning med ellipsoider för fotonmappning på GPU:er

MARCUS BERGHOLM, VIKTOR KRONVALL

Degree Project in Computer Science, DD143X
Supervisor: Michael Schliephake
Examinator: Örjan Ekeberg

CSC, KTH 2016-05-09

Abstract

Recent development in Graphics Processing Units (GPUs) has enabled inexpensive high-performance computing for general-purpose applications. The K-Nearest Neighbors problem is widely used in applications ranging from classification to gathering of photons in the Photon Mapping algorithm. Using the euclidean distance measure when gathering photons can cause false bleeding of colors between surfaces. Ellipsoidical search boundaries for photon gathering are shown to reduce artifacts due to this false bleeding. Shifted Sorting has been found to yield high performance on GPUs while simultaneously retaining a high approximation rate. This study presents an algorithm for approximatively solving the K-Nearest Neighbors problem modified to use a distance measure creating an ellipsoidical search boundary. The ellipsoidical search boundary is used to alleviate the issue of false bleeding of colors between surfaces in Photon Mapping. The Approximative K-Nearest Neighbors algorithm presented is a modification of the Shifted Sorting algorithm. The algorithm is found to be highly parallelizable and performs to a factor of 86% queries processed per millisecond compared to a reference implementation using spherical search boundaries implied by the euclidean distance. The rate of compression from spherical to ellipsoidical search boundary is appropriately chosen in the range 3.0 to 7.0. The algorithm is found to scale well in respect to increases in both number of data points and number of query points.

Referat

Grafikprocessorer (GPU-er) har på senare tid möjliggjort högprestandaberäkningar till låga kostnader för generella applikationer. K-Nearest Neighbors problemet har vida applikationsområden, från klassifikation inom maskininlärning till insamlande av fotoner i Photon Mapping för rendering av tredimensionella scener. Användning av euklidiska avstånd vid insamling av fotoner kan leda till en felaktig bladning av färger mellan ytor. Ellipsoidiska sökområden vid fotoninsamling har visats reducera artefakter oraskade av denna typ av felaktiga färgutblandning. Shifted Sorting har visats ge hög prestanda på GPU-er utan att förlora kvalitet av approximationsgrad. Denna rapport undersöker hur den approximativa varianten av K-Nearest Neighborsalgoritmen med Shifted Sorting presterar på GPU-er med avståndsmåttet modifierat sådant att ett ellipsoidiskt sökområde bildas. Algoritmen används för att reduceras problemet av felaktig blanding av färg i Photon Mapping. Algoritmen visas vara mycket parallelliserbar och presterar till en grad av 86% behandlade sökpunkter per millisekund i jämförelse med en referensimplementation som använder sfäriska sökområden. Kompressionsgraden längs sökpunktens ytnormal väljs fördelaktligen till ett värde i intervallet $3,0$ till $7,0$. Algoritmen visas skala väl med avseende på både ökningar i antal data punkter och antal sökpunkter.

Contents

1 Introduction 1
1.1 Problem statement 2

2 Background 5
2.1 Physically based 3-d rendering 5
2.1.1 Ray Tracing 6
2.1.2 Photon Mapping 7
2.2 K-Nearest Neighbors 8
2.2.1 Ellipsoid distance measure for K-Nearest Neighbors 8
2.3 K-Nearest Neighbors using GPGPU 9
2.3.1 CUDA 9
2.3.2 K-Nearest Neighbors using CUDA 11
2.3.3 Approximative K-Nearest Neighbors 11
2.3.4 Shifted Shorting 11

3 Implementation 13

4 Result 21
4.1 Hardware 21
4.2 Benchmarks 21
4.3 Rendering 27

5 Discussion 31
5.1 Conclusion 33

References 35

Chapter 1

Introduction

The K-Nearest Neighbors Algorithm is a Nearest Neighbor Search algorithm and an optimization problem for finding the k closest points given a query point. Closeness is determined using a distance function or a distance measure.

The applications of the K-Nearest Neighbors problem are many and ranges from classification in machine learning to anomaly detection in data mining and therefore it is of importance to find a fast algorithm to solve the K-Nearest Neighbors problem.

The K-Nearest Neighbors problem may be defined using a set of data points, $\mathcal{D}$ and a set of query points $\mathcal{Q}$ as shown in figure 1.1. Let $\mathcal{D} = \{d_1, d_2, d_3, ..., d_n\}$ and $\mathcal{Q} = \{q_1, q_2, q_3, ..., q_m\}$ for each $q_i \in \mathcal{Q}$ find the k points in $\mathcal{D}$ that are the closest to the query point q_i. The k points are found using the distance measure which is commonly the euclidean distance between q_i and d_j. These points are the points within the sphere with radius equal to the distance between q_i and the k-th data point from q_i.

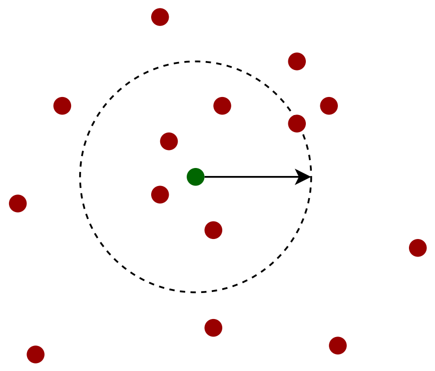


Figure 1.1: K-Nearest Neighbors with k=5 where the green point in the center is a query point and the red points are points in $\mathcal{D}$

Using the GPU for finding the k nearest neighbors yields significant speedup compared to calculating K-Nearest Neighbors on the CPU. Even a “brute-force” implementation on the GPU may be faster than a more sophisticated implementation on the CPU. (Garcia, Debreuve, and Barlaud 2008)

K-Nearest Neighbors is commonly solved using a *k-d tree* on the CPU in order to limit the search space when finding the nearest data points to a query point but Li et al. (2012) describes a modern approach for approximately solving K-Nearest Neighbors using shifted sorting that performs better on the GPU.

In Photon Mapping the K-Nearest Neighbors algorithm is widely used to calculate the illumination of a distinct point by finding the k closest photons. To calculate the distance to these points it is common to use the euclidean distance. This can cause false bleeding of colors between surfaces leading to a superfluous influence of colors from nearby surfaces when estimating radiance. A more accurate way to obtain the closest photons that limits this problem is to instead use a disc or ellipsoid compressed in the direction of the surface normal at the point. (Jarosz, Jensen, and Donner 2008)

1.1 Problem statement

The purpose of this study is to evaluate how the Approximate K-Nearest Neighbors algorithm suggested by Li et al. (2012) will preform when using an ellipsoid as search boundary. It will also check if the ellipsoid search boundary have any significant improvement on the problem of false bleeding between surfaces.

This project will focus on using K-Nearest Neighbors in Photon Mapping as described by Henrik Wann Jensen, specifically implementing the proposed ellipsoid version of K-Nearest Neighbors that reduces false bleeding of colors between surfaces. (Jarosz, Jensen, and Donner 2008)

In order to allow for fast calculation of the k nearest neighbors we propose implementing a modified version of the algorithm suggested by Li et al. (2012) that takes the ellipsoidic measure into account. For calculating such a measure we require both a position and a normal vector for every query point $q_i \in \mathcal{Q}$.

Using a different search boundary however may have consequences on how to most efficiently store the data points and parallel calculation of distances may prove difficult which in turn may incur performance penalties on rendering scenes. It is therefore important to verify if the performance characteristics still hold when modifying Shifted Sorting Approximate K-Nearest Neighbors to use ellipsoidic progression of the search boundary when gathering the nearest points.

Research Questions

- *Is is possible to implement the Approximate K-Nearest Neighbors algorithm in a highly parallel fashion on consumer-grade GPUs while using an **ellipsoid** as the search boundary?*
- *What shape of the ellipsoid is appropriate to use and how should k be chosen in order to gain a satisfiable approxmiation when rendering 3-dimensional scenes with Photon Mapping?*
- *What data structures should be used to store the points in order to retain a good approximation when using the ellipsoidic search boundary?*

Chapter 2

Background

The following chapter presents the underlying theories and background needed for developing an algorithm that may answer the research questions in section 1.1.

2.1 Physically based 3-d rendering

Physically based rendering of 3-dimensional scenes is generally done by solving or approximating the rendering equation presented by Kajiya (1986). The rendering equation describes the outgoing radiance from a given point in the direction of a solid angle. Outgoing radiance is based on the irradiance to the point, the material properties of the surface and the emitted radiance at the point. This is illustrated in figure 2.1

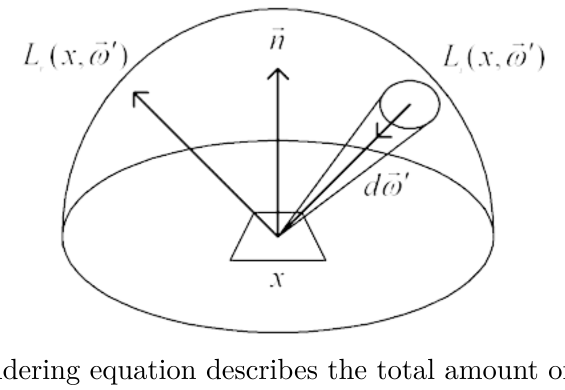


Figure 2.1: The rendering equation describes the total amount of light emitted from a point x along a particular viewing direction

The reflected radiance L_r with wavelength λ at time t from the surface point $\mathbf{x}$ in the outgoing direction ω_o is:

$$L_r(\mathbf{x}, \omega_o, \lambda, t) = \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t)(\omega_i \cdot \mathbf{n})\, d\omega_i \tag{2.1}$$

where
f_r is the bi-directional reflectance distribution function of the surface material at point $\mathbf{x}$.
L_i is the irradiance in the incoming direction ω_i to the point $\mathbf{x}$.
Ω is the hemisphere above the point $\mathbf{x}$.
$\mathbf{n}$ is the surface normal at the point $\mathbf{x}$.

The outgoing radiance is:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + L_r(\mathbf{x}, \omega_o, \lambda, t) \tag{2.2}$$

where
L_e is the emitted radiance in the outgoing direction ω_o.

2.1.1 Ray Tracing

Ray Tracing is an algorithm that is commonly used to render 3-dimensional scenes on computers. Ray Tracing handles multiple reflections and refracted light in a more straightforward fashion than rasterization algorithms. Ray Tracing was originally developed by Appel (1967) for creating line drawings where directly visible and invisible lines are shown. It was later revised by Rubin and Whitted (1980) to enable its main modern application as an algorithm for approximating the rendering equation.

Despite light rays naturally travelling from the light sources to the eye, in the ray tracing algorithm light rays are traced in the opposite direction, i.e. from the eye, or *camera*, to the light source. For every pixel on the screen a ray is shot from the camera through the pixel. If the ray intersects an object the intersection point together with a shading model is then used to calculate the color of that pixel. For specular materials a new ray is then traced in the direction reflected by the surface normal and the radiance at the intersection point of this new ray is then added when calculating the radiance of the first point.

Ray Tracing only supports direct illumination and specular reflections. It is unable to express more complex light paths such as those that modelling multiple diffuse reflections between surfaces. Without these light paths the rendered image can be perceived as harsh. The interreflections between surfaces are required in order to produce a more “soft” image.

2.1.2 Photon Mapping

Photon Mapping is an algorithm that extends ray tracing to handle indirectly illuminated scenes in order to more accurately approximate the rendering equation. This is done by adding a pre-render pass that for each light source emits photons from the light source. Photon paths are traced throughout the scene and when a photon's path intersects a surface the photon is stored in a map with the position, flux and incoming direction. (Jarosz, Jensen, and Donner 2008)

The Monte Carlo method Russian Roulette is used to determine at each intersection point if a photon should be diffusely reflected, specularly reflected or absorbed when emitting the photon. (Jarosz, Jensen, and Donner 2008)

The Photon Map is later used in the rendering pass to calculate indirect illumination and caustics. When determining the radiance of a point the K-Nearest Neighbors algorithm is used to find the closest photons and integrating over these taking the surface area into account. (Jarosz, Jensen, and Donner 2008)

The radiance at a point is calculated by approximating the term L_r from equation 2.1 as a sum of the power of photons gathered from a small area on a surface. This sum is then weighted by the area of the disc under the hemisphere the photons lie within. The approximation of radiated light can now be directly found from the Photon Map and is calculated as shown in equation 2.3. (Jensen 2001)

$$L_r(x, \omega_o) = \sum_{p=1}^{N} f_r(x, \omega_i, \omega_o) \frac{\Delta\Phi_p(x, \omega_i)}{\pi r^2} \tag{2.3}$$

Note the exclusion of time and wavelength in equation 2.3 as the intensity $\Delta\Phi_p(x, \omega_i)$ of a photon uses the color stored as the flux of that photon and as the Photon Mapping algorithm does not take time-dependent rendering into account.

Jensen and Christensen (2007) notes however, that using the Photon Map directly to calculate radiance would require a large number of photons to correctly represent light. Jensen and Christensen (2007) then developed a method to more efficiently calculate the radiance with high fidelity using only a fraction of the otherwise needed photons in the Photon Map. This method is based on irradience-caching as described by Ward and Heckbert (1992).

The radiance term of the rendering equation is split into four parts, direct illumination, specular reflections, caustics and indirect illumination. Each part is then calculated separately and integrated to obtain the total illumination radiated from the point (Jensen and Christensen 2007).

2.2 K-Nearest Neighbors

The K-Nearest Neighbors problem defined by using a set of data points, $\mathcal{D}$ and a set of query points $\mathcal{Q}$ is used to find the k nearest data points for each query point. Let $\mathcal{D} = \{d_1, d_2, d_3, ..., d_n\}$ and $\mathcal{Q} = \{q_1, q_2, q_3, ..., q_m\}$. For each $q_i \in \mathcal{Q}$ find k distinct datapoints $d_j \in \mathcal{D}$ that minimizes the distance $\delta(q_i, d_j)$

Closeness between query points and data points are calculated using a distance measure. For low-dimensional applications of the K-Nearest Neighbors algorithm the euclidean distance between the points is commonly used as this measure. The euclidean distance measure results in a spherical search boundary around each query point with the sphere containing the k nearest data points.

2.2.1 Ellipsoid distance measure for K-Nearest Neighbors

Using conventional K-Nearest Neighbors for gathering the photons when calculating the radiance estimate introduces the problem of extraneous bleeding of colors between surfaces. This is because photons on nearby surfaces may lie closer in 3-dimensional space than photons on the same surface. Jensen and Christensen (2007) recommends using a disc as the search boundary when gathering photons to reduce the impact of this false bleeding of colors as shown in figure 2.2.

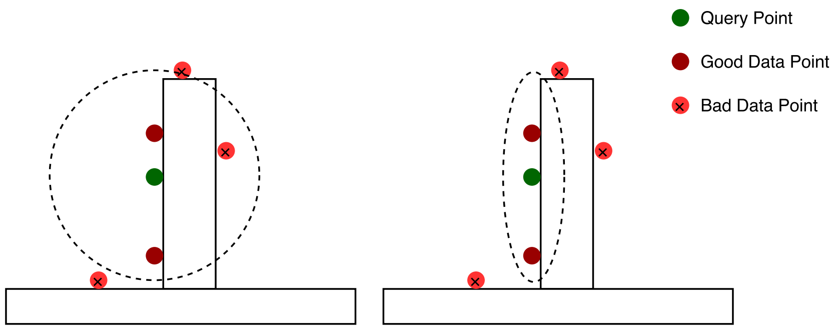


Figure 2.2: Example of using the euclidean distance(left), and ellipsoid distance(right)

Having a too high compression rate collapses the entire coordinate-space into the tangent plane for the query point and the ellipsoidical boundary is effectively a disc on this plane. Distances along the normal are now so distant that it is very unlikely that data points will be found that do not lie in this plane. This can lead to artifacts in the image caused by a different kind of false bleeding. Here photons are gathered from distant surfaces that intersect the tangent plane as there are too few photons within the plane in the vicinity of the query point. One must therefore take caution not to increase the compression rate too much. In figure 2.3 photons are gathered from the floor despite the query point lying on the surface of the sphere causing false bleeding.

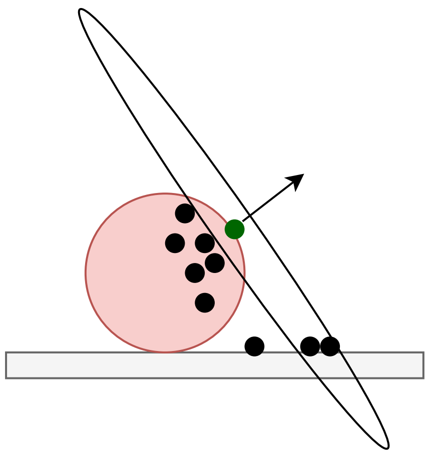


Figure 2.3: Overly compressed ellipsoid gathering false photons

2.3 K-Nearest Neighbors using GPGPU

General Purpose computing on graphical processing units (GPGPU) is a method for using graphical processing units (GPUs) to perform operations the GPU was not originally developed for.

GPUs were initially only able to perform specialized operations in order to produce rasterized renders of 3-dimensional scenes. However with the advent of shaders that support floating-point operations more general applications of the GPU were made possible (Fung and Mann 2004). The benefits of using GPUs over CPUs for performing calculations are mainly that the architecture of the GPU allows for more highly parallel execution as there are many cores on the GPU that may execute independently of each other (Mittal and Vetter 2015).

GPGPU was introduced to consumers in the early 2000s as the commercial technologies OpenCL and CUDA (Tompson and Schlachter 2012). Both technologies exploit the floating point operating shader architectures using a specialized C-like programming language to enable programmers write functions that may run in parallel on the multiple cores of the GPU (Che et al. 2008).

2.3.1 CUDA

CUDA is an extension of the C++ programming language defining *kernel* functions (Che et al. 2008). A kernel function is a C++ function for executed by one sequential thread. (Nickolls et al. 2008)

CUDA uses a hierarchal model for distributing work across the GPU or GPUs. The complete set of GPUs is called a grid which is divided into thread blocks. A thread block is set of threads that run concurrently. Threads within the same block may cooperate using shared memory and synchronization intrinsics. Because each thread executes the kernel the different threads may run in parallel to each other. (Nickolls et al. 2008)

Kernel functions should be designed to not depend on other threads when executing in order to achieve high levels of parallelization. Inter-dependence of executing threads and excessive synchronization in a block limits the parallelizability of the execution.

CUDA's memory management is also hierarchal ranging from global and texture memory to registers for a specific thread as shown in figure 2.4. Global memory together with texture memory and constant memory constitute the memory accessible by all thread blocks on the grid. Each thread block has its own area of shared memory that is accessible from all threads executing in that block. Every thread in a thread block also have local memory area as well as GPU registers. (Nickolls et al. 2008, Luebke (2008))

Access to global memory in CUDA take hundreds of processor clock cycles. Shared memory and register access do not incur such a penalty for accessing the memory there. To increase memory bandwidth and reduce latency memory may be copied from global memory to shared memory. This results in a speedup if the data is accessed multiple times in a single thread block. (Nickolls et al. 2008)

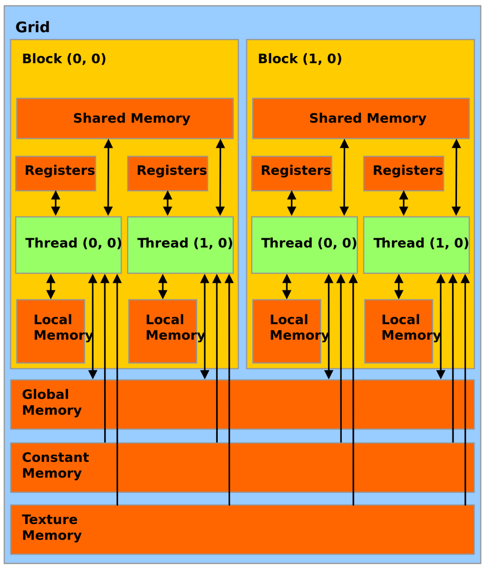


Figure 2.4: The memory model of CUDA

2.3.2 K-Nearest Neighbors using CUDA

K-Nearest Neighbors is an algorithm that is suited for computation on the GPU as the different query points' k nearest neighbors may be calculated independently of each other.

Naive *brute-force* algorithms may be sufficient to yield a significant speedup obtaining the k nearest neighbors for all query points when utilising the GPU (Garcia, Debreuve, and Barlaud 2008). However using more sophisticated methods may yield even greater speedups. Speedups are especially prominent in the step to gather the k nearest neighbors of a given query point (Arefin et al. 2012). Furthermore, more sophisticated methods may also increase the *cache-locality* by avoiding access to all of the data points utilising greater data reuse so that the memory accesses may be limited to local shared memory when computing the neighbors (Barrientos et al. 2010).

2.3.3 Approximative K-Nearest Neighbors

Some applications do not need to give the exact k nearest neighbors but an approximate result may be sufficient to produce an acceptable result. Photon Mapping is such an application. (Li et al. 2012)

By trading precision for cache-locality and performance a near-correct solution may be found by only calculating the distance between the query point and some of the data points.

2.3.4 Shifted Shorting

Li et al. (2012) describes a highly parallel algorithm for approximately obtaining the k nearest neighbors by using Morton Codes and shifted sorting. Morton Codes of the locations for the data points and query points are computed and then sorted by the Morton Codes.

This creates an implied *octtree* order of the points where points with nearby indices in the sorted array of points lies in the same octtree-node. The octtree structure may be poorly aligned to the points and therefore all points are shifted by a small amount and morton codes re-calulated. The $2k$ approximate nearest points for each shift are gathered, k in each direction of the query point, and then merged while only retaining the k nearest neighbors to the query point during each merge.

The shifted sorting algorithm obtains high parallelization because all query points approximate nearest neighbors are found simultaneously. The sorting performed in the initial step of the algorithm finds these approximate nearest neighbors to the query points by placing them next to each query point. It implies however that since the Morton Codes are pre-calculated and used for sorting all points together that different query points cannot have different frames of reference, i.e. all distances must be calculated in the same coordinate-space when obtaining the Morton Codes. As such, different ellipsoid boundaries cannot be used in this step for each of the query points when calculating distances to the data points without duplicating all data points for every query point. Such a duplication of data would in turn severely limit the parallel nature of the algorithm and is therefore not a feasible solution.

Chapter 3

Implementation

Our algorithm described in algorithm 3.1 continues the work by Li et al. (2012). In order to provide an ellipsoidical search boundary as suggested by Jensen and Christensen (2007) we create a coordinate-space for each query point. All distances in this space along the query point's surface normal are greater. By scaling the space along the surface normal within this coordinate-space we effectively create an ellipsoidical space within the standard Cartesian coordinate-space.

However, in order to allow for high levels of parallelization we need to calculate the approximate nearest neighbors for many query points in parallel. Having a separate coordinate-space for each query point would either require duplicating all data points for each query point or rescaling all distances for each query point. Both of which would quickly become unfeasible as the number of data points or query points increase.

If approximate nearest neighbors are gathered using the spherical boundary some of the approximate nearest neighbors may become distant when transforming the points into the query point's ellipsoidical coordinate-space. Therefore, we divide all query points into separate *buckets*, each representing a coordinate-space. Query points are then partitioned into these buckets according to the minimum angle between the query point's surface normal and the normal of the bucket.

Distances in the bucket coordinate-space are approximately equal to distances in the query point's coordinate-space due to having the query point's surface normal approximately equal to the bucket normal. This allows us to find approximate nearest neighbors to query points within a bucket by initially using the bucket coordinate-space when measuring distances. However, if the compression rate of the bucket-space is too high, points that are near the query point inside the query point's coordinate-space may become distant in the bucket-space. These points would be excluded when initially gathering approximate nearest neighbors. Thus, by using the data structure of buckets we hope to see an improvement in approximation rate.

The bucket normals are carefully chosen such that they are distributed as evenly as possible on the unit sphere. Noting that surfaces commonly are axis-aligned in 3-d scenes we also include the axes as bucket normals. Having too few buckets leads to the bucket spaces being very dissimilar to the query points' spaces within the bucket and the approximate nearest neighbors may find data points distant in the query point's space. Conversely, having too many buckets leads to superfluous work being done without improving the approximation. We chose to have 18 buckets, six of which are the axis aligned spaces and twelve that are normals chosen as the vertices on a regular icosahedron. The angles between the query normals and the bucket normals are thus guaranteed to be at most $\frac{\pi}{8}$ radians.

In order to reduce data consumption we do not duplicate the data points for each bucket. Instead, we transform the points from standard Cartesian space into the bucket coordinate-space when entering a bucket. After calculating the nearest neighbors for all query points in the current bucket we then transform all points back into the standard Cartesian coordinate-space.

In order to determine which query points lie within the current bucket we loop over all points noting whether the point is a data point or if it is a query point. Data points are marked as 0, query points within the current bucket are marked as 1. This is done by maximizing the dot product between the bucket normal and query normal. If the index of the maximum dot product equals this bucket then the query point is assumed to lie within this bucket. Query points not within the current bucket are marked as 2.

Before continuing we create an array storing all the original indices and values of both the query points and the data points. The array of indices and values are then sorted in ascending order based on the array previously created marking data points and query points. This leaves us with the query points outside the current bucket at the end of the array of indices and values. The sorted array of indices and values also let us find the index of points in the original input.

With the array sorted we then traverse the array marking points. This time we replace all 2's with 1's and other values with 0. The sum of this array is then found. Note that this sum represents the number of query points outside the current bucket. With this information the total number of points and the number of points within the current bucket are obtained.

All points within the bucket then transformed into the coordinate-space of the bucket. We then find the bounds for each axis within this space and scale the points to the range $[0, 0.70)$ as proposed by Li et al. (2012). Having the points in the bucket coordinate-space and rescaled, morton codes are calculated for all points within the bucket. The morton codes describe an implicit octtree where codes close to each other lie within the same subtree of the octtree.

The points are sorted into the morton code order together with the original indices retrieved in the first step of the algorithm. An array marking query points as 1 and data points as 0 are then created from the array of sorted points. The number of query points to the left of every data point are then obtained by performing an inclusive prefix sum on this array. The array of points are then compacted into two separate arrays. One containing the data points and the other containing an index into the array of data points for each query point. The index notes the data point to the right of the query point as shown in algorithm 3.2.

$2\lambda k$ candidate nearest neighbors are found for each query in the bucket coordinate-space. λk threads are created for each query point. The natural number λ is a constant chosen to improve the approximation accuracy for the ellipsoid boundary. Each thread responsible for finding 2 candidate nearest neighbors for this query point as shown in algorithm 3.3.

For each query point in the bucket, the $2\lambda k$ data points are now filtered down to the nearest k points. The k points are obtained by first transforming the $2\lambda k$ data points into the query point's coordinate-space and then sorted with bitonic sort according to it's distance to the query point within the query point coordinate-space.

With the initial approximate k nearest points being found for each query within this bucket we now shift all points by 0.05 along all axes as suggested by Li et al. (2012). This is to improve the accuracy of the approximation. Especially in the cases where points lie close each other but across different subtrees of the octtree created by the morton codes. Morton codes are then recalculated, points sorted in the morton code order and compaction done with this new ordering of the points.

Another $2\lambda k$ candidate nearest neighbors are then found by spawning $2\lambda k$ threads for each query in the bucket. Each thread is now responsible for obtaining one candidate nearest neighbor to the query point as shown in algorithm 3.4. The data point is then transformed into the query point's coordinate-space and the distance in this space is obtained. Binary search yields what index in the array of the k nearest points this new point should have. If this index is equal to k, the candidate is discarded. Otherwise, the value at the index for this location in an array of counters is atomically incremented. An exclusive prefix sum is performed on the array of counters. Finally the candidates are then merged into the array of the k nearest neighbors taking the indices from the counter array into account. These steps are then repeated 3 more times each time shifting by another 0.05 yielding a better approximation of the k nearest neighbors.

After all of the four shifts are finished and our k nearest neighbors found for all query points within the current bucket all points within the bucket are then transformed back into the standard Cartesian coordinate-space and we continue to the next bucket.

Algorithm 3.1: K-Nearest Neighbors with ellipsoidic distance measure

```
1: function K-NEAREST NEIGHBORS($values_0, normals, results$)
2:     $n \leftarrow$ LENGTH($values_0$)
3:     for $i = 0$ to 18 do
4:         $indexedValues \leftarrow$ store original indices and values of values
5:         $marks \leftarrow$ mark queries not in- and in bucket
6:         $indexedValues \leftarrow$ sort indexedValues in marks order
7:         $outsideBucket \leftarrow$ NOTINBUCKET($marks$)
8:         $indexedValues \leftarrow$MOVE TO BUCKET SPACE($bucket, indexedValues$)
9:         $indexedValues \leftarrow$SCALE VALUES($indexedValues$)
10:        $reverseIndices \leftarrow$ store reverse indices of indexedValues
11:        for $j = 0$ to 4 do
12:            $indexedValues \leftarrow indexedValues + 0.05 * j$
13:            $mortoncodes \leftarrow$COMPUTE MORTON CODES($indexedValues$)
14:            $indexedValues \leftarrow$ sort values in morton code order
15:            $queriesToLeft \leftarrow$MARKQUERIES($indexedValues$)
16:            $queriesToLeft \leftarrow$INCLUSIVE PREFIX SUM($queriesToLeft$)
17:            $compacted \leftarrow$COMPACT VALUES($indexedValues, reverseIndices, queriesToLeft$)
18:            $(data, queryIndices) = compacted$
19:            for $query$ in current bucket pardo
20:                $i_q \leftarrow queryIndices[query]$
21:                $2\lambda k$ $candidates \leftarrow data[i_q - \lambda k - 1, i_q + \lambda k)$
22:                if $j = 0$ then
23:                    sort $candidates$ by distance from $query$ in query ellipsoid space
24:                    Save the $k$ nearest $candidates$ into $results$
25:                else
26:                    Merge $2\lambda k$ new candidates into $results$
27:                    sort $results$ by distance from $query$ in ellipsoid space
28:                end if
29:            end for
30:        end for
31:        $indexedValues \leftarrow$RESCALE VALUES($indexedValues$)
32:        $indexedValues \leftarrow$MOVE TO UNIT SPACE($bucket, indexedValues$)
33:    end for
34: end function
```

Algorithm 3.2: Compact Values to data points only remembering indices of queries into the data point array

```
function COMPACT VALUES($indexedValues, reverseIndices, queriesToLeft$)
    for $value \in indexedValues$ pardo
        if ISQUERY($value$) then
            $originalIndex \leftarrow$ original index of value
            $queryIndex \leftarrow reverseIndices[originalIndex] - numData$
            $dataIndex \leftarrow i - queriesToLeft[i] + 1$ ▷ index of data point to right
            $queryIndices[queryIndex] \leftarrow$PACK 64-BIT($dataIndex, i$)
        else
            $dataIndex \leftarrow i - queriesToLeft[i]$
            $data[dataIndex] \leftarrow values[i]$
        end if
    end for
end function
```

Algorithm 3.3: Find candidates

```
function FIND CANDIDATES($indexedValues, data, result$)
    $i \leftarrow$ thread index
    $query \leftarrow$ block index
    $candidates \leftarrow$ allocate array of size $2\lambda k$ in shared memory
    $ellipsoidSpace \leftarrow$ allocate matrix for query point in shared memory
    if first thread in block then
        $ellipsoidSpace \leftarrow$ calculate conversion martix for query point
    end if
    for two of the $2\lambda k$ nearest neighbor candidates of this query pardo
        find two candidates in bucket space
        calculate distance for candidates in query ellipsoid space
        store the two new candidates in candidates
    end for
    BITONIC SORT($candidates, 2\lambda k, i$)
    for the first $k$ threads pardo
        store the $k$ nearest candidates in $result$
    end for
end function
```

Algorithm 3.4: Find and merge condidates

```
function FIND AND MERGE CANDIDATES($indexedValues, data, allNearestNeighbors$)
    $i \leftarrow$ thread index
    $query \leftarrow$ block index
    $ellipsoidSpace \leftarrow$ allocate matrix for query point in shared memory
    $counter \leftarrow$ allocate array of size $2k$ in shared memory
    $counterScan \leftarrow$ allocate array of size $2k$ in shared memory
    $currentNN$ allocate array of size $k$ in shared memory
    $updatedNN$ allocate array of size $k$ in shared memory
    $currentNN[i] \leftarrow allNerestNeighbors[q * k + i]$
    for every counter[i] pardo
        if $i$ is odd then
            $counter[i] \leftarrow 1$
        else
            $counter[i] \leftarrow 0$
        end if
    end for
    if first thread in block then
        $ellipsoidSpace \leftarrow$ calculate conversion martix for query point
    end if
    for one of the $2\lambda k$ new nearest neighbor candidates of this query pardo
        find candidate nearest neightbor in bucket space
        calculate distance for candidate in query ellipsoid space
        $loc \leftarrow$ binary search the location of candidate in currentNN
        if loc = k then
            Stop processing this candidate by marking it as inactive
        else
            $offset \leftarrow$ previous value in counter[loc*2]
            Atomically increment counter[loc*2]
        end if
    end for
    if first thread in block then
        $counterScan \leftarrow$EXCLUSIVE PREFIX SUM($counter$)
    end if
    for candidate in currentNN[i] pardo
        $index \leftarrow counterScan[2i + 1]$
        if $index < k$ then
            $updatedNN[index] \leftarrow currentNN[i]$
        end if
    end for
```

Algorithm 3.4: Find and merge candidates (continued)

```
40:     for every active candidate in candidates pardo
41:         $index \leftarrow counterScan[2loc] + offset$
42:         if $index < k$ then
43:             $updatedNN[index] \leftarrow candidate$
44:         end if
45:     end for
46:     if $i < k$ then
47:         $allNearestNeighbors[q * k + i] \leftarrow updatedNN[i]$
48:     end if
49: end function
```

Chapter 4

Result

The following chapter presents the results obtained by comparing our implementation as described in chapter 3 to a reference implementation based on the implementation described by Li et al. (2012). Approximation rate of our implementation is also evaluated.

4.1 Hardware

The implementation was run on a Nvida GeForce GTX 970 GPU using CUDA 7.2 with compute capabilities 5.2. The GTX 970 card has a global memory size of 4 GiB, a processing power of 3494 GFLOPS and a shared memory size of 48 KiB. The CPU of the benchmarking computer was Intel Core i5-6600 Skylake a 4-core processor with clock frequency of 3.3 GHz.

4.2 Benchmarks

Our implementation of K-Nearest Neighbors using ellipsoidic distance was compared to a reference implementation based on the algorithm described by Li et al. (2012) using the same datasets for both implementations. The datasets used are Uniformly distributed random points in 3-d space, a set of 3-dimensional clusters of gaussian normal distribution, the Stanford Bunny and data points from the Photon Map when rendering a scene similar to the Cornell Box.

The results were compared varying the constant k, the constant λ, the compression rate, c, along the surface normal of the ellipsoidic search boundary introduced by our algorithm. Approximation rate of the algorithm was also compared to an implementation collecting the nearest neighbors exactly. The ellipsoidic search boundary of the exact implementation was assigned the same shape as the search boundary used in our implementation.

Figure 4.1 shows the performance of our implementation compared to the reference implementation by varying the number of query points from 128 to 2M in both implementations. A variant of our implementation where buckets were not used was also compared. In this implementation the query points are not sorted into buckets but all queries handled at the same time. However points are scaled into the ellipsoidic space created by each query point after gathering $2\lambda k$ approximate nearest neighbors in standard Cartesian space. Compression Rate of the ellipsoidical search boundary was selected to $c = 4.0$.

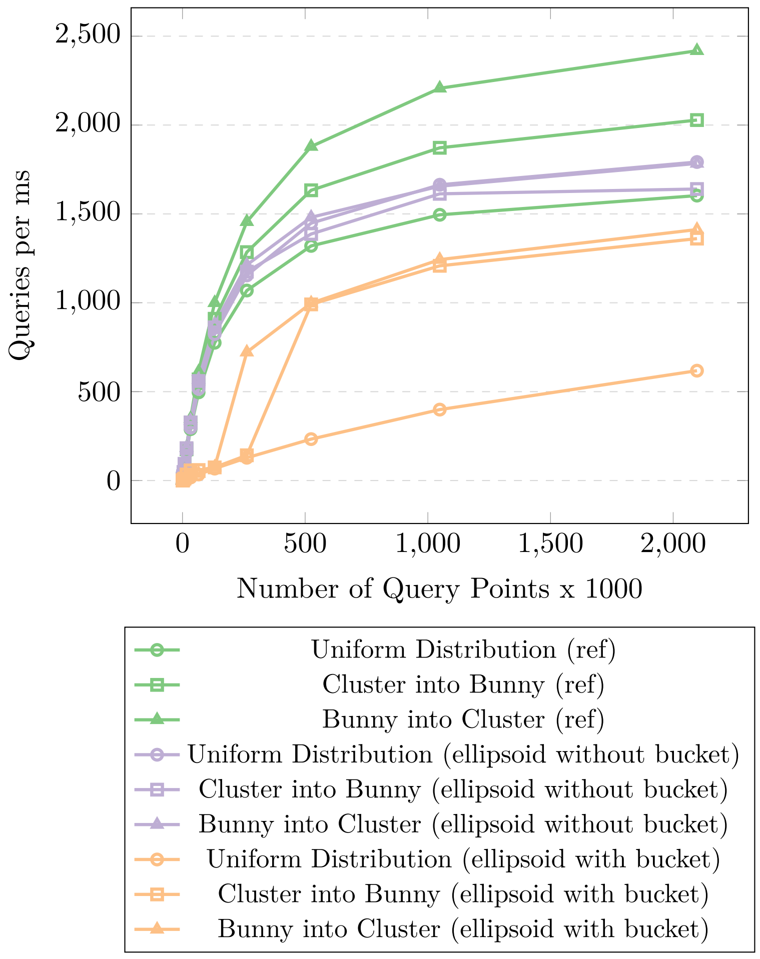


Figure 4.1: KNN on 2M data points varying number of query points from 128 to 2M, $k = 64$, $c = 4.0$ (higher is better)

The reference implementation processed on average 2016 query points per millisecond when using 2M query points. Our implementation without using buckets processed on average 1738 query points per millisecond. Our implementation using buckets processed on average 1130 query points per millisecond. This equates to a factor of

0.56 for our implementation with buckets and a factor of 0.86 for our implementation without using buckets. Similar performance characteristics as those obtained by Li et al. (2012) are achieved in all implementations. However, it is notable that our implementation performs worse on the uniformly distributed points when using buckets. On the contrary it is also notable that the our implementation without using buckets performs better than the reference implementation in some cases.

By varying the number of data points when performing the K-Nearest Neighbors Search we find a small linear decrease in number of queries processed each millisecond when increasing the number of data points in the reference implementation as shown in figure 4.2. These results are reproduced in our implementation except for the data set of uniformly distributed points in the variant using buckets. For this data set we find an exponentially decreasing number of query points per millisecond processed where the exponent is a slightly negative value.

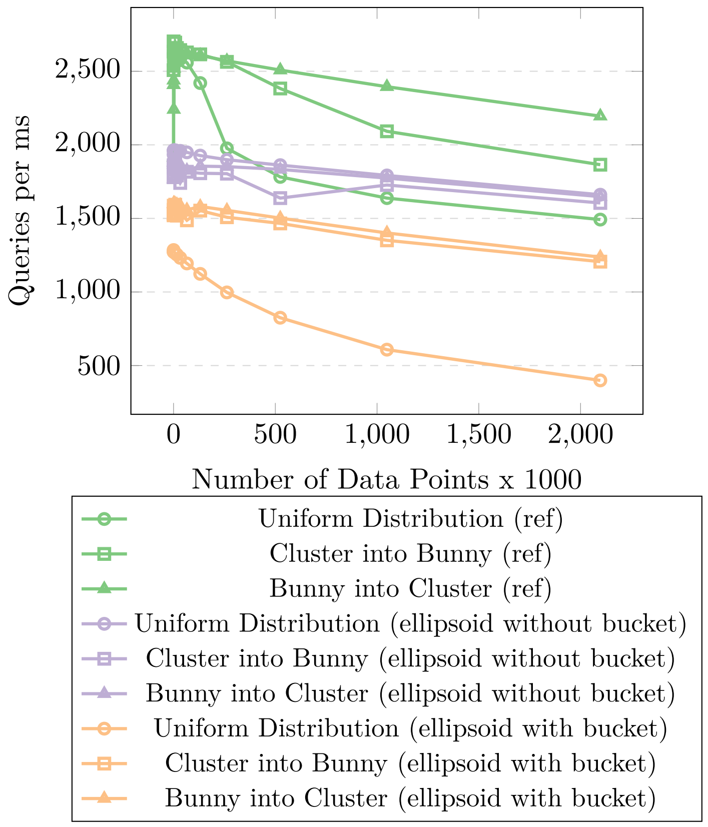


Figure 4.2: KNN on 1M query points varying number of data points from 128 to 2M, $k = 64$, $c = 4.0$ (higher is better)

Even when varying the constant k similar performance characteristics are obtained for our implementation compared to the reference implementation. Figure 4.3 shows performance of the implementations with k being increased in steps of 10 in the range $[10, 160]$. The curves follow a pattern of powers of two where there is only a slight linear decrease in performance within an order of magnitude. However increasing k to the next order of magnitude yields a significant drop in performance. This is most likely due to bitonicSort requiring the array to be of a size that is a power of 2 which in turn requires the shared memory and number of threads used per block to be padded to this size. Because of this it is therefore wise to choose k to be a power of 2.

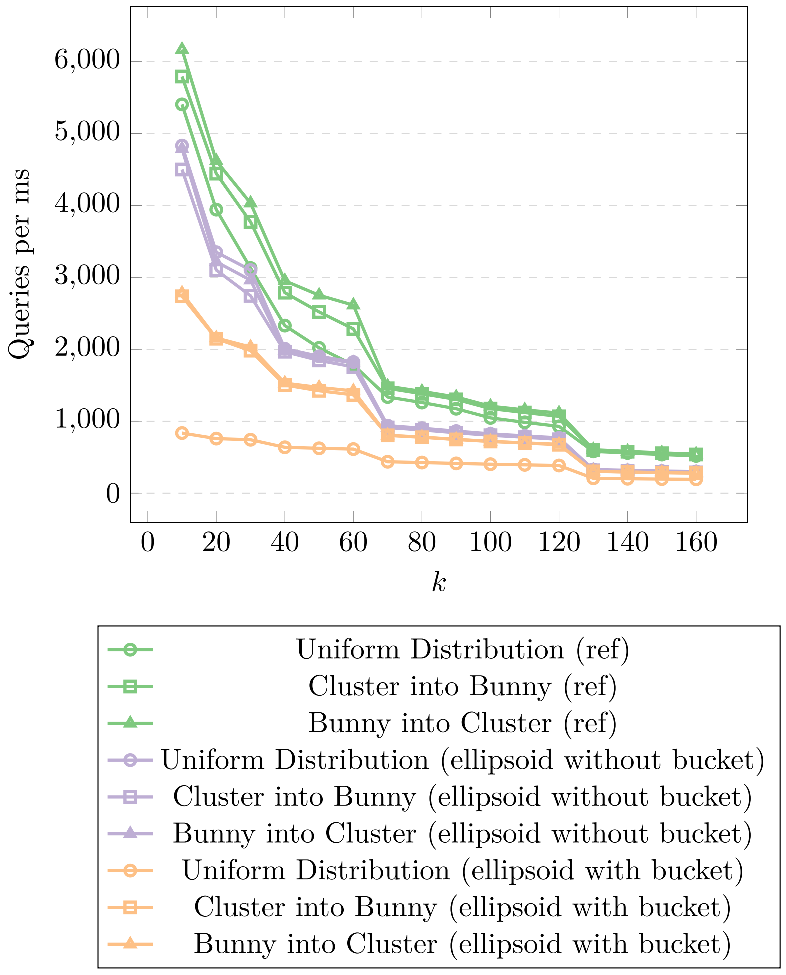


Figure 4.3: KNN on 1M query points and 1M data points by varying k, $c = 4.0$ (higher is better)

The constant λ introduced in algorithm 3.1 was evaluated with respect to performance. Increasing λ yields results much like those obtained by increasing k. Figure 4.4 shows that the product $2\lambda k$ is favorably chosen to be a power of 2.

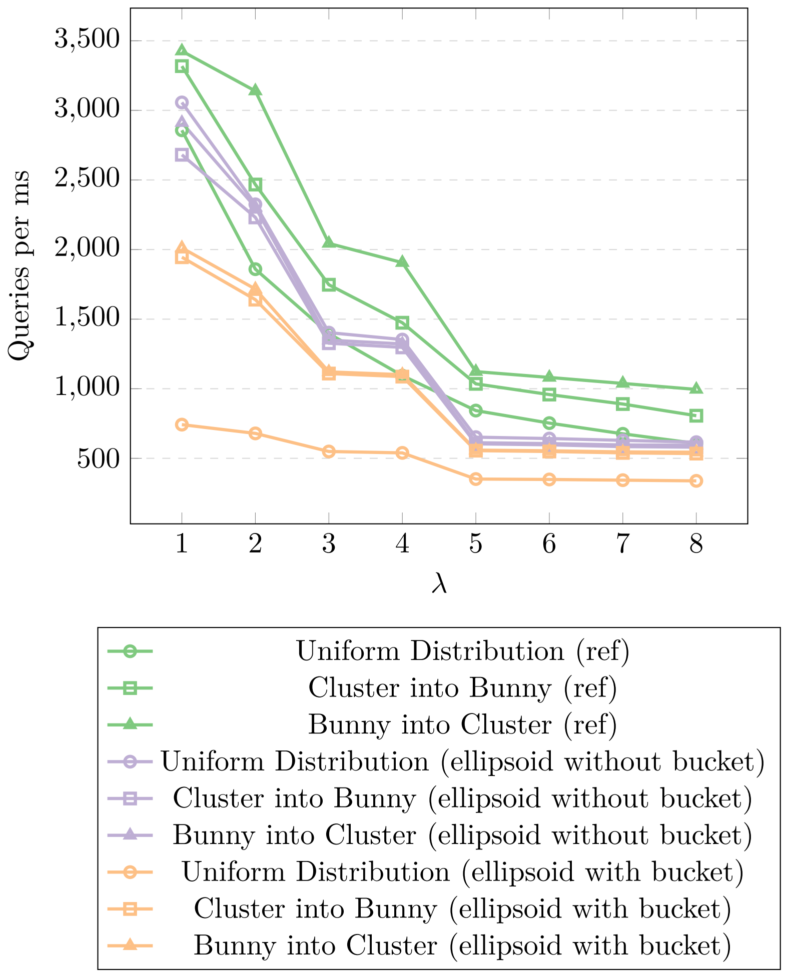


Figure 4.4: KNN on 1M query points and 1M data points by varying λ, $k = 32$, $c = 4.0$ (higher is better)

By transforming all points into ellipsoidical spaces the approximation factor is now mainly determined by the rate of compression. The approximation was calculated as the factor of dividing the exact distance from the query point to the k-th data point into the distance of the query point to the approximately obtained k-th data point. Figure 4.5 shows the approximation rate for uniform and Photon Mapping data sets with $k = 256$ with 1M data points and 128 query points. The compression rate was then varied from $c = 1.0$ to $c = 8.0$ and the approximation factor was calculated.

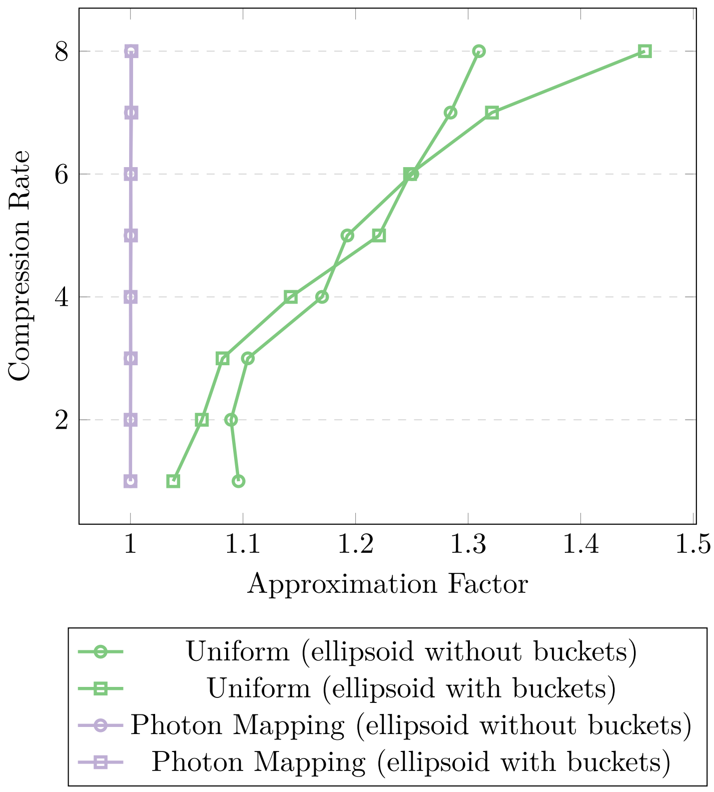


Figure 4.5: Approximation factors for different compression rates

The implementation was found to be memory-bound in both the variant using buckets and the variant without buckets. Running on the Nvidia GTX 970 card we were able to handle 1024 candidate nearest neighbors. The number of candidates used is found by calculating the product $2\lambda k$. Running on hardware with larger shared memory area would allow for higher number of candidates being processed.

4.3 Rendering

Images were rendered directly estimating radiance from the Photon Map as this more clearly visualized the problem of false bleeding. In a real-world scenario one should consider using the distributed ray tracing approach described by Jensen and Christensen (2007) where direct and indirect illumination are handled separately.

This problem of false bleeding is shown in figure 4.6 where a spherical search boundary was used when gathering photons. Note the bands of red and green color on the floor in the vicinity of the red and green walls respectively. In this case the Photon Mapper was modified to exclude reflection of photons and as such only direct illumination is handled in the radiance estimate.

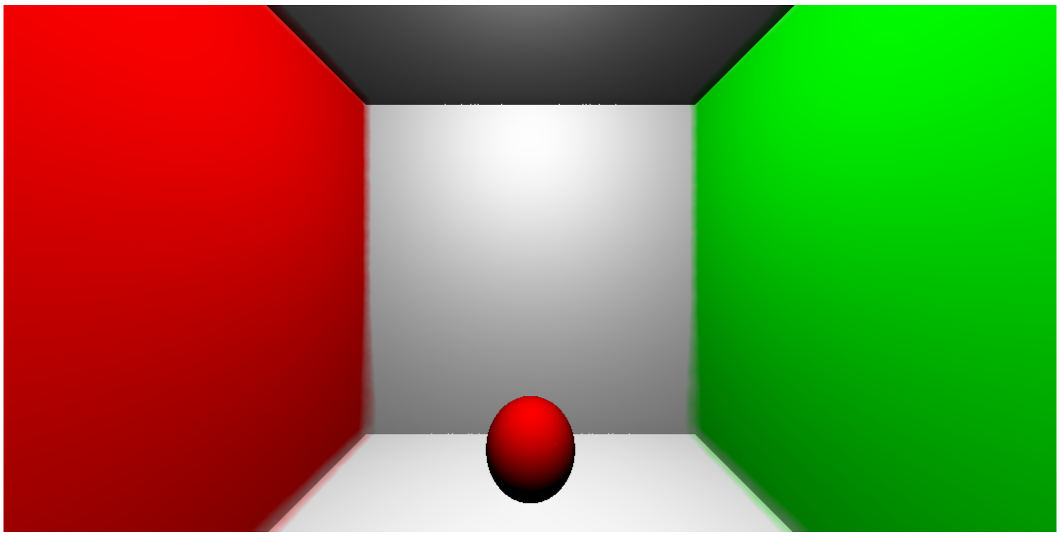


Figure 4.6: Direct illumnation with false bleeding of colors

Transforming all points into the ellipsoidical spaces created by the query points yields renders with much less false bleeding between surfaces as figure 4.7 shows. This render was performed using the variant of our implementation sorting query points into separate buckets using a compression rate $c = 4.0$ along the query points' surface normals and $k = 256$.

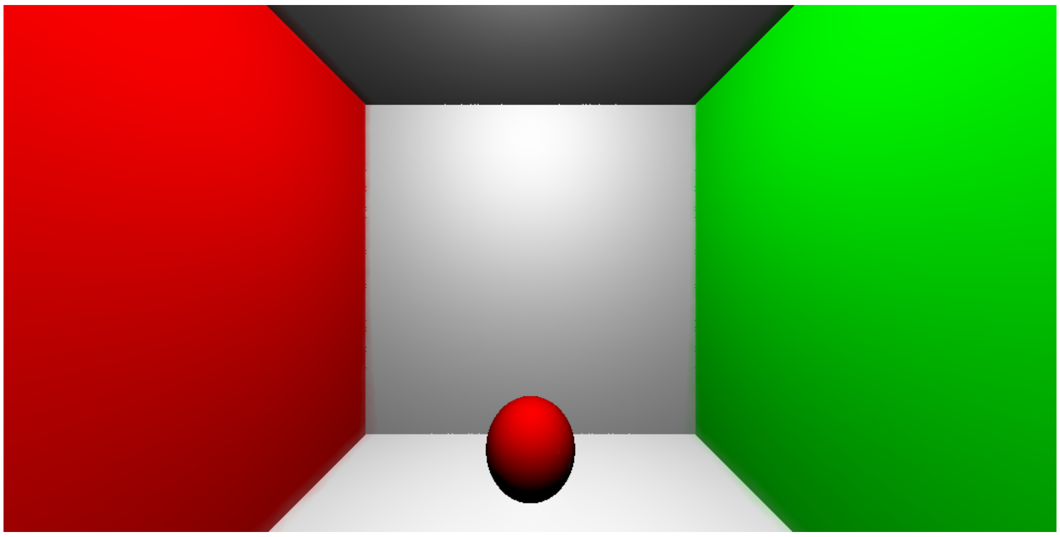
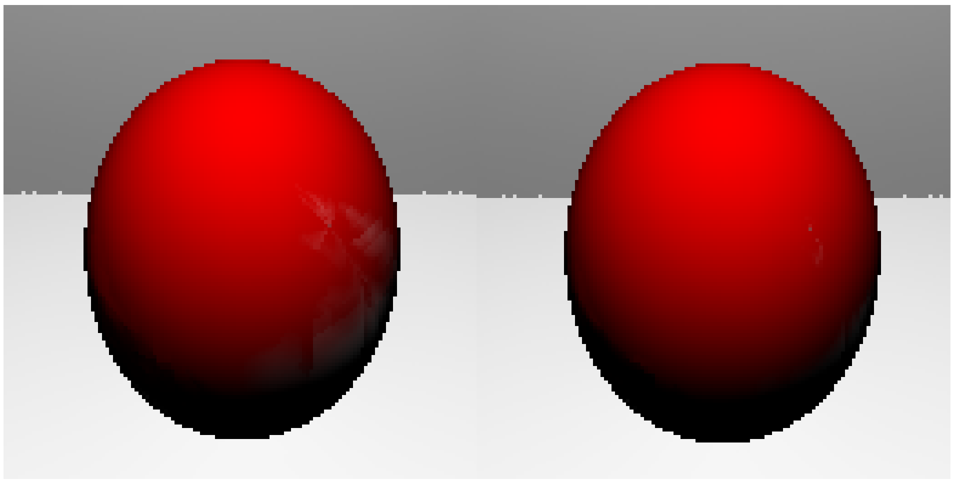


Figure 4.7: Direct illumnation with alliviated problem of false bleeding by using an ellipsoidical search boundary

Figure 4.8 shows artifacts due to a too high compression rate. The images were rendered with a compression rate of $c = 7.0$. By varying the constant λ we were able to obtain the same rate of false bleeding between walls despite lowering the value of k. Note the bleeding from the floor on the sphere in the left image.



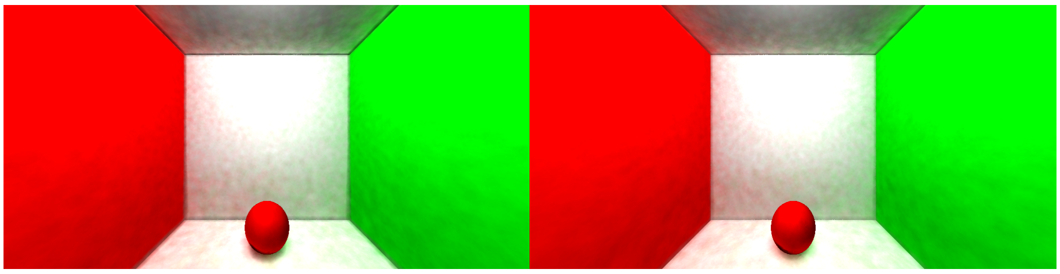
(a) $k = 256$, $\lambda = 1$

(b) $k = 128$, $\lambda = 2$

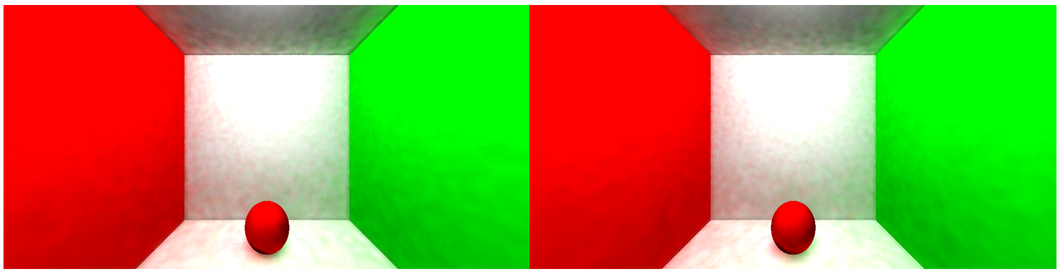
Figure 4.8: Same compression rate using different values of λ. k is chosen to give equivalent rate of false bleeding between walls and floor

After introducing diffuse interreflection between surfaces the problem of false bleeding persists when using a spherical search boundary for gathering the photons for the radiance estimate. Figure 4.9 was rendered using $k = 256$, varying the compression rate from $c = 1.0$ (spherical) to $c = 5.0$. Note the dark artifacts in the corners of the box. These are caused by the false bleeding of colors between the surfaces.

With a compression rate of $c = 2.5$ there is slight, but noticeable bleeding of colors between the walls. Increasing the compression rate to $c = 3.5$ results in a degree of false bleeding that is barely visible. Further increasing the compression rate eliminates the problem of bleeding altogether. Choosing the compression rate to $c = 5.0$ yields renders where the false bleeding is not visible at viewing distances. However, increasing the rate of compression also introduces slight artifacts visible on the sphere due to the false bleeding caused by gathering photons from the floor.



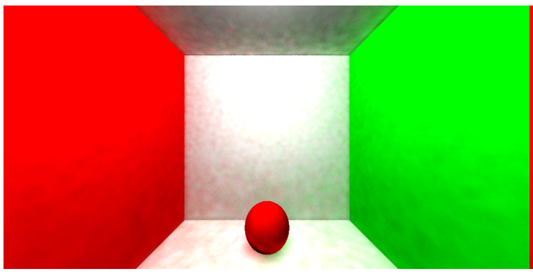
(a) $c = 1.0$ (b) $c = 2.5$

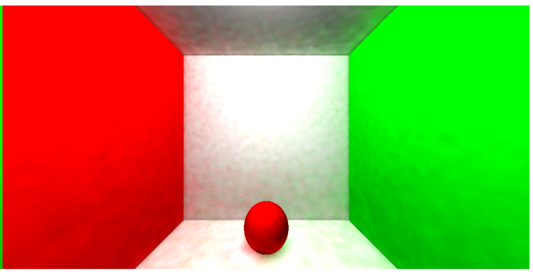


(c) $c = 3.5$ (d) $c = 5.0$

Figure 4.9: Box scene rendered by varying compression rate

Approximation factor is very similar both when using buckets and when excluding them as shown in figure 4.5 the. Figure 4.10 was rendered once using buckets and once without buckets. The visual results are indistinguishable from each other both in terms of false bleeding and other rendering artifacts. A difference image between the two renders were obtained by using the open software ImageMagick. Most areas are equal and the differing areas are likely due to randomness when emitting the photons into the scene.


(a) Using buckets


(b) Without buckets

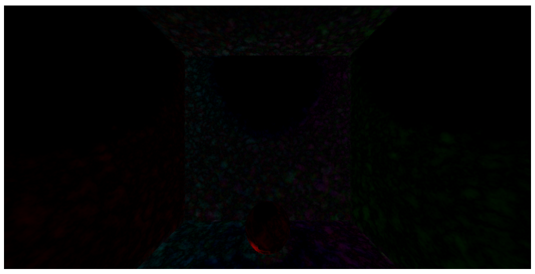

(c) Difference between the images

Figure 4.10: Comparison of renders with and without sorting query points into buckets

Chapter 5

Discussion

The purpose of this study was to evaluate how the Approximate K-Nearest Neighbors algorithm suggesed by Li et al. (2012) performs when using an ellipsoidical search boundary as suggested by Jensen and Christensen (2007). The ellipsoidical search boundary is introduced in order to increase the quality of renders by eliminating false bleeding of colors between surfaces. Performance impacts incurred by the ellipsoidical search boundary were presented in figures 4.1 to 4.4.

Performance characteristics in all benchmarks were found to be similar to those presented by Li et al. (2012). The faster Nvidia GeForce GTX 970 GPU did however yield an approximate speedup of 2.0 compared to the results shown by Li et al. (2012). Increasing the number of data points were shown not to highly affect the performance. Increasing the number of query points yields a higher rate of queries processed each millisecond in a logarithmic fashion. Contrarily, increasing k decreases the rate of queries processed per millisecond closely following the power of 2's such that for each order of magnitude the speed is lowered to a factor of approximately 0.6 of the previous magnitude.

The algorithm is shown to be appropriate for computations on graphical processing units (GPUs). The execution is memory-bound where the most limiting factor is the size of the available shared memory on the GPU. The Nvidia GeForce GTX 970 card allows calculations with the number of candidate nearest neighbors ($2\lambda k$) of values up to 1024. The algorithm however is not tightly coupled to the specific card and we suspect high degrees of applicability when using different GPUs. It should be noted that larger shared memory areas are recommended.

The impact on the approximation factor of using different shapes and compression rates for the ellipsoidical search boundary was evaluated as presented in figure 4.5. Rendering quality was evaluated when varying the shape as shown in section 4.3. The impact on performance and quality of renders for choosing different values of k and λ were presented in figures 4.3 and 4.4 as well as section 4.3.

The ellipsoidical search boundary incurs a performance penalty of a factor of 0.56 when using buckets and a factor of 0.86 when not partitioning query points into buckets. These factors were consistent throughout all benchmarks. This performance penalty however is to be expected as more work is required for each query. Especially the work due to the matrix multiplication when transforming points into the query point's coordinate-space which is performed for each of the $2\lambda k$ candidate nearest neighbors for each query point. The performance penalty of the ellipsoidical version using buckets was greatest for the data set of uniformly distributed points. Due to the coupling of the scene geometry and the query points' surface normals, using buckets when rendering scenes may still be applicable. Given that surfaces are either approximately aligned to the buckets or that there are few non-flat surfaces in the scene the approximation rate is likely high. However, the decrease in performance when using buckets may make the algorithm unsuitable where high speeds are required, unless a near-exact solution is required for renders. The version not using buckets however did perform at speeds close to those obtained by the reference implementation and as such is highly recommended for Photon Mapping purposes.

Approximation rate was examined for structuring the points by partitioning query points into separate buckets as shown in figure 4.5. This was also compared to an implementation where query points were not partitioned into buckets. Difference in quality of renders for these versions were presented in figure 4.10.

Figure 4.5 shows that higher degrees of compression along the surface normal when creating the ellipsoidical search boundary yields a small decrease in quality of approximation. The factor of this decrease was found to be approximately 0.04 without buckets and approximate 0.07 when using buckets. The higher degree of error in the version using buckets is likely due to the problem of over-compression described in section 2.2.1. When using points from the Photon Mapper the approximate solution was found to be very close to the exact solution with an approximation factor of at most 1.029 ($\varepsilon = 0.029$) without using buckets and at most 1.085 ($\varepsilon = 0.085$) when using buckets. Figure 4.9 shows that compression rates higher than 3.0 are sufficient to reduce the problem of false bleeding. Figure 4.5 shows that the version using buckets yield better approximations for compression rates up to 4.0. The rate of compression is thus appropriately chosen to be in the range of 3.0 to 7.0 when not using buckets and in the range of 3.0 to 4.0 when using buckets.

Further research is needed in order to find causes of difference in approximation rate between the bucket variant compared to the variant not using buckets. It should also be evaluated if it is possible to limit the copy of data for each bucket in order to improve performance of the variant using buckets.

Due to the very similar approximation factors of the two ellipsoidical versions for Photon Mapping data near-equal results were obtained when rendering the 3-d scene. This result is presented in figure 4.10.

5.1 Conclusion

This study presents an algorithm for approximately solving the K-Nearest Neighbors problem modified to use a distance measure creating an ellipsoidical search boundary. The ellipsoidical search boundary is used to alleviate the issue of false bleeding of colors between surfaces in Photon Mapping. The algorithm was implemented using CUDA to run on consumer GPU hardware allowing for highly parallelized execution. Increasing the constant k incurs a performance penalty. However, the algorithm scales well with respect to increases in number of query points and number of data points.

Near-exact results were obtained for Photon Mapping data sets where the problem of false bleeding was eliminated by the ellipsoidical search boundary. Appropriate values of compression rate when compressing the search boundary into an ellipsoid were found to be in the range 3.0 to 7.0.

The introduction of the ellipsoidical search boundary only resulted in a linear decrease in performance, measured as the number of processed query points per millisecond. This factor was found to be approximately 0.86. A version with a data structure partitioning query points into buckets were created, yielding better rates of approximation for compression rates up to 4.0.

References

Appel, Arthur. 1967. "The Notion of Quantitative Invisibility and the Machine Rendering of Solids." In *Proceedings of the 1967 22nd National Conference*, 387–93. ACM.

Arefin, Ahmed Shamsul, Carlos Riveros, Regina Berretta, and Pablo Moscato. 2012. "GPU-FS-K NN: A Software Tool for Fast and Scalable K NN Computation Using GPUs." *PloS One* 7 (8). Public Library of Science: e44000.

Barrientos, RJ, JI Gómez, C Tenllado, and M Prieto. 2010. "Heap Based K-Nearest Neighbor Search on GPUs." In *Congreso Espanol de Informática (CEDI)*, 559–66.

Che, Shuai, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. 2008. "A Performance Study of General-Purpose Applications on Graphics Processors Using {CUDA}." *Journal of Parallel and Distributed Computing* 68 (10): 1370–80. doi:http://dx.doi.org/10.1016/j.jpdc.2008.05.014.

Fung, J., and S. Mann. 2004. "Computer Vision Signal Processing on Graphics Processing Units." In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, 5:V–93–6vol.5. doi:10.1109/ICASSP.2004.1327055.

Garcia, V., E. Debreuve, and M. Barlaud. 2008. "Fast K Nearest Neighbor Search Using GPU." In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, 1–6. doi:10.1109/CVPRW.2008.4563100.

Jarosz, Wojciech, Henrik Wann Jensen, and Craig Donner. 2008. "Advanced Global Illumination Using Photon Mapping." In *ACM SIGGRAPH 2008 Classes*, 2:1–2:112. SIGGRAPH '08. New York, NY, USA: ACM. doi:10.1145/1401132.1401136.

Jensen, Henrik Wann. 2001. *Realistic Image Synthesis Using Photon Mapping*. Natick, MA, USA: A. K. Peters, Ltd.

Jensen, Henrik Wann, and Per Christensen. 2007. "High Quality Rendering Using Ray Tracing and Photon Mapping." In *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. New York, NY, USA: ACM. doi:10.1145/1281500.1281593.

Kajiya, James T. 1986. "The Rendering Equation." *SIGGRAPH Comput. Graph.*

20 (4). New York, NY, USA: ACM: 143–50. doi:10.1145/15886.15902.

Li, Shengren, Lance Simons, Jagadeesh Bhaskar Pakaravoor, Fatemeh Abbasinejad, John D. Owens, and Nina Amenta. 2012. "KANN on the GPU with Shifted Sorting." In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, edited by Carsten Dachsbacher, Jacob Munkberg, and Jacopo Pantaleoni. The Eurographics Association. doi:10.2312/EGGH/HPG12/039-047.

Luebke, D. 2008. "CUDA: Scalable Parallel Programming for High-Performance Scientific Computing." In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, 836–38. doi:10.1109/ISBI.2008.4541126.

Mittal, Sparsh, and Jeffrey S. Vetter. 2015. "A Survey of CPU-GPU Heterogeneous Computing Techniques." *ACM Comput. Surv.* 47 (4). New York, NY, USA: ACM: 69:1–69:35. doi:10.1145/2788396.

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron. 2008. "Scalable Parallel Programming with CUDA." *Queue* 6 (2). New York, NY, USA: ACM: 40–53. doi:10.1145/1365490.1365500.

Rubin, Steven M., and Turner Whitted. 1980. "A 3-Dimensional Representation for Fast Rendering of Complex Scenes." *SIGGRAPH Comput. Graph.* 14 (3). New York, NY, USA: ACM: 110–16. doi:10.1145/965105.807479.

Tompson, Jonathan, and Kristofer Schlachter. 2012. "An Introduction to the Opencl Programming Model." *Person Education.*

Ward, Greg, and Paul Heckbert. 1992. "Irradiance Gradients." In *Eurographics Rendering Workshop*, 85–98.