High Performance Graphics (2012)
C. Dachsbacher, J. Munkberg, and J. Pantaleoni (Editors)

kANN on the GPU with Shifted Sorting

Shengren Li Lance Simons Jagadeesh Bhaskar Pakaravoor Fatemeh Abbasinejad John D. Owens Nina Amenta

University of California at Davis

Abstract

We describe the implementation of a simple method for finding k approximate nearest neighbors (ANNs) on the GPU. While the performance of most ANN algorithms depends heavily on the distributions of the data and query points, our approach has a very regular data access pattern. It performs as well as state of the art methods on easy distributions with small values of k, and much more quickly on more difficult problem instances. Irrespective of the distribution and also roughly of the size of the set of input data points, we can find 50 ANNs for 1M queries at a rate of about 1200 queries/ms.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—Sorting and searching

1. Introduction

Nearest neighbor finding is an important step in many algorithms. High-dimensional nearest neighbors are used in many machine learning algorithms. Low-dimensional nearest neighbors are important in point-set processing in computer graphics and geometrical modeling. Two typical examples are estimating normals for a point-cloud prior to point-based rendering, and collecting the radiance of photons near a visible point in photon mapping.

Sequentially, low-dimensional nearest neighbor problems are usually solved by building a data structure, usually a k-d tree, on the data points and then using it to find the nearest neighbors for each query point. This is also a common approach on the GPU, where the k-d tree may be built either sequentially or in parallel, and then multiple queries are processed in parallel. Our algorithm instead builds on recent efficient 64-bit sorting on the GPU [MG11] and the *shifted sorting* of Liao, Lopez, and Leutenegger [LLL01]. The behavior of this algorithm is very regular with respect to the data distribution, unlike more traditional tree-search algorithms, so that it remains fast even on difficult instances where a k-d tree algorithm does poorly.

1.1. Problem definition

In the nearest neighbors problem, we are given a set P of data points, and another set Q of queries; for each $q \in Q$, we want to report its k nearest points in P, where k is a parameter that depends on the application.

There is no good upper bound on the time required to answer a nearest neighbors query; bad queries might have many almost equally close neighbors that need to be checked. So generally some approximation is allowed in the result, in the sense that the kth nearest neighbor may be at distance $(1+\varepsilon)$ times the distance to the true kth nearest neighbor. The greater the approximation factor ε, the faster the search can be, but often in practice allowing a large ε gives a good speed-up while producing results with much smaller approximation factors. Alternatively, many implementations find exact nearest neighbors, and this often works well in practice since bad query cases, which would be very slow, are rare.

1.2. Our algorithm

Our GPU algorithm builds on the simple and elegant sequential algorithm of Liao, Lopez, and Leutenegger [LLL01], which we call *shifted sorting*. Shifted sorting in turn is based on a remarkable lemma of Chan (Lemma 3.3 in [Cha97]). In 3D, their data structure consists of five sorted lists of the data points, in Hilbert code order; the lists differ from each other because the data points are shifted with respect to the coordinate system by a different amount before computing the Morton codes for each list. The sequential algorithm for performing queries is simply to locate a query point in each

© The Eurographics Association 2012.

DOI: 10.2312/EGGH/HPG12/039-047

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

list by binary search, and then to consider the k preceding and k succeeding points; from these $10k$ points, the k nearest to the query point are returned. Chan's Lemma is used to prove that *five* specific shifts lead to an algorithm with a provable approximation factor of $\varepsilon = 29$. While this is absurdly high, and increases with the dimension, Liao, Lopez, and Leutenegger [LLL01] observed quite good approximation results in high dimensions. We report similarly positive results on many distributions in $\mathbb{R}^3$, including some arising in a photon mapping application.

This algorithm is appealing for the GPU since sorting has been studied extensively in recent years. We use the recent high-performance 64-bit radix sort implementation of Merrill and Grimshaw [MG11]. We eliminate the binary searches by simply sorting the query points along with the data points. Since sort is so efficient, the majority of the computation time is thus taken by the selection of the lists of k nearest neighbors for each query. For small k this is easy, but in graphics applications often values of $k = 50$ or $k = 100$ are required, so that, for example, if we make 1 million queries on a data set of size 2 million with $k = 100$, the total size of the lists of nearest neighbors dwarfs the size of the input, and this becomes the major performance challenge.

1.3. Contributions

The main contributions we make are:

- The observation that the shifted sorting algorithm provides a more GPU-friendly basis for ANN searching than the more well-known k-d tree algorithm,
- The idea of sorting the data and queries together to eliminate searching,
- An efficient GPU algorithm for selecting the best k nearest neighbors from the $10k$ candidate points,
- A demonstration that this method performs better than a state-of-the-art k-d tree based GPU code, especially on difficult distributions of data and queries, and
- An experimental evaluation of the effective approximation error.

2. Related work

FLANN [ML09b, ML11b] is a recent nearest neighbor library, including both CPU and GPU implementations. The GPU code is designed for dimension three, and implements the standard k-d tree algorithm, using a priority queue for each query's search. In difficult cases the size of the priority queues can grow quite large. We show comparisons to FLANN in Section 4.

The k-d tree approach was adapted to the GPU by Zhou et al. [ZHWG08] so as to avoid the priority queues, by precomputing, for each k-d tree cell, the maximum distance to the kth nearest neighbor, which allowed them to do a stackless traversal. Qiu et al. [QMN09] also compute a single (possibly not nearest) neighbor for 3D registration applications by precomputing a k-d tree on the CPU and searching it on the GPU.

More recently, Leite et al. [LTF*12] compute k nearest neighbors on the GPU with a 3D voxel grid data structure using the brute force algorithm. This is very effective when the query and data distributions are both uniform, either in space or on the surface of an object, but it degrades rapidly on difficult inputs.

Higher-dimensional k nearest neighbors is an important problem in machine learning, computer vision, and databases [BDHK06, GDB08, KZ09]. High-dimensional GPU implementations have focused on the brute-force algorithm; an exception is Pan et al. [PLM10], who use locality-sensitive hashing. Some techniques use a heap structure in order to maintain the k nearest neighbors [KH12, BGTP10] for each query.

Octree search is the basis of the theoretically optimal CPU algorithm for approximate k nearest neighbors [AMN*98] in fixed dimension, with any choice of ε. The algorithm of Liao, Lopez and Luetenegger [LLL01], on which our method is based, is related to octree search in that Morton codes can be seen as labeling the octree cells containing the points. Octrees have been constructed on the GPU and used for Poisson surface reconstruction [ZGHG11]. The BVH structure of Lauterbach et al. [LGS*09] is closely related to the construction of an octree by sorting Morton codes.

Photon mapping on the GPU is an area of great recent interest, beginning with Purcell et al. [PDC*03] and Ma and McCool [MM02], both of which used forms of spatial hashing, as did a recent speed-up [HJ10] of the progressive photon mapping algorithm of Hachisuka and Jensen [HJ09]. Spatial hashing is a more space-efficient version of the voxel-grid approach, in which only occupied voxels are hashed into a data structure, and again it does not do well on difficult ANN examples. We use a modified version of a progressive photon mapping demo using the NVIDIA Optix Ray-tracing Engine [NVI12] as a platform to test our kANN algorithm.

More recent global illumination solutions seek to reduce or eliminate k nearest neighbor searches. Wang et al. [WWZ*09] carefully cluster and average photons to form a greatly reduced set of irradiance samples. Somewhat similarly, McGuire and Luebke [ML09a] improve performance by scattering irradiance from photons to nearby pixels rather than gathering it via k nearest neighbors, and Fabianowski and Dingliana [FD09] calculate a footprint describing the influence of each photon on its neighbors.

3. Algorithm

In this section we explain two main things. The first is how the shifted sorting algorithm works, and why it is so suitable for the GPU; and the second is how the bookkeeping

© The Eurographics Association 2012.

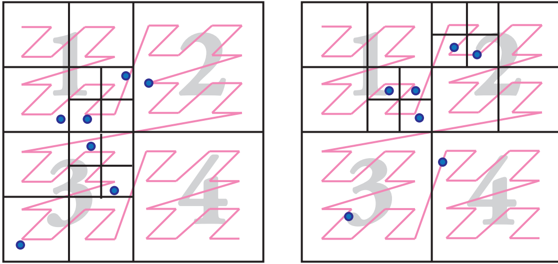

Figure 1: We use Morton codes to impose an ordering based a recursive traversal of octree cells. The most significant bits of the Morton code are the number of its largest containing top-level cell (the gray numbers), followed by the number of its cell at the next level (numbered according to the same scheme), and so on. Sorting the cells by Morton code orders them along the pink space-filling curve (left). When the input data is shifted, they appear along the curve in a different order (right).

required to collect the k approximate nearest neighbors can be done efficiently. Algorithm 1 shows the pseudo-code for our overall algorithm.

Algorithm 1: Compute k nearest neighbors

```
Input: Data points and query points, as floating point
       x, y, z
Output: For each query point, an array of k nearest data
        points, as a list of indices
Scale all points
for j = {0,1,2,3,4} do
    Shift all points, data and queries, by j × 0.05
    Compute Morton codes for all points
    mc_all[] = Sort all points, with their original
    indices, in Morton code order
    mc_data[] = Compact mc_all[] for data points
    foreach query point q pardo
        i_q = Find the index of the nearest data point to
        its left in mc_all[]
        2k candidates = mc_data[i_q − k − 1, i_q + k]
        if j = 0 then
            Sort 2k candidates by distance from q
            Save the nearest k candidates as the initial
            nearest neighbors
        else
            Merge 2k new candidates into the array of
            the current nearest neighbors (this is
            described in Algorithm 2)
            Sort the updated nearest neighbor array
        end
    end
end
```

3.1. Data scaling and Morton code computation

Our algorithm is based on sorting together the data and query points, as represented by their Morton codes. Morton-code sorting orders the points along the space-filling Z-curve. Liao et al. [LLL01] ordered points along the Hilbert curve instead; the analysis of the algorithm is the same in either case. In practice, we found that the Hilbert curve would improve our approximation quality by a small factor (no more than 3%), but at the cost of significantly more computation.

There is an implied octree decomposition associated with the Morton code order: points whose Morton codes share a prefix of length $d \times i$ share an octree cell at level i (where level 0 has the largest cells). Points that are near each other in space are not necessarily near each other along the space-filling curve (see Figure 1, left); this happens when a point lies near the boundary of its octree cell and a neighbor lies outside the cell. But if the points are shifted with respect to the curve, the ordering induced by the curve changes, so that nearby points in space that were far apart in the old ordering might be close together in the new one (Figure 1, right); that is, points that formerly lay in different octree cells might now be together.

Our implementation of shifted sorting begins by sorting the data and query points together. Then, each query point checks the data points to its right and to its left in the sorted order for potential approximate nearest neighbors. It then repeats this process using several different shifts, always keeping track of the nearest k neighbors seen so far. The theoretical results of Liao et al. [LLL01] and Chan [Cha02] established that in $\mathbb{R}^3$, five specific shifts (described below) ensure that k approximate nearest neighbors will be found, for any k, with an approximation factor $\varepsilon < 29$. Recall that this means that the distance from the query to the kth nearest neighbor reported is at most 30 times greater than the distance to the true kth nearest neighbor. While this is a nearly useless upper bound, we find, as they did, that in practice the results are invariably much better.

We use 64-bit Morton codes in our implementation, giving us 21 bits for each of the x, y and z coordinates. The remaining least significant bit is used to distinguish between the Morton code of a query point (1) and a data point (0). With 21 bits per dimension, we find that every point gets a unique Morton code in all of our experiments.

The shifts required by the algorithm must be relatively prime with respect to the spacing of the power-of-two grid on which the implied octree is constructed. Chan's lemma guarantees that for points in the cube $[0,1)^3$, shifting by 0.2 at each iteration will approximate all k nearest neighbors; this requires handling shifted points in the range $[0,1.8)^3$. Instead, we scale to $[0,0.75)^3$ and shift by 0.05, keeping the shifted data within $[0,1)^3$. The smaller shifts give us essentially one more bit per dimension in the Morton code, but

© The Eurographics Association 2012.

we lose the guarantee of being able to find far-off nearest neighbors, those at a distance > 0.25.

After scaling and shifting, we convert the coordinates to integers, and then compute Morton codes. This requires interleaving the bits of the twenty-one least-significant bits of each coordinate (the others are zero) into the most-significant 63 bits of the Morton code.

3.2. Sorting and candidate identification

After radix-sorting the data and queries together, every query needs to consider k data points to its left and k data points to its right. To avoid having to read and skip other query points, we compact the data, removing the queries, while remembering, for each query, the index i_q at which q would be located in the compacted data list. In the following section, each query will be compared with k points on either side of i_q in this compacted list, for a total of $2k$ potential neighbors per query.

3.3. Maintaining nearest neighbors

Since each iteration generates $2k$ nearest neighbor candidates per query, as k increases, storing all candidates from the five iterations takes up a great deal of memory. Hence, we only store the k nearest candidates for each query point as an array in the first iteration and update these arrays in the following four iterations. After the final iteration, the remaining candidates in the arrays are the result.

Maintaining the arrays of k nearest neighbor candidates, *allNN*[], is the main challenge in the implementation of this algorithm. At each iteration, we may discover new nearest neighbor candidates that need to be merged into *allNN*[] and also duplicate candidates that we have seen in previous iterations (note that within the $2k$ candidates taken from the data point array in sorted Morton code order in one iteration, there are no duplicates).

In order to replace current candidates with new, nearer candidates and detect duplicates efficiently, we store *allNN*[] in sorted order based on the distances from the candidate data points to the query point. We store these candidates as the concatenation of the distance to the query point with the index into the data point array. Using the index as the least-significant bits of the record allows us to sort by distance while being able to detect and eliminate duplicate candidates.

First iteration: Among the $2k$ data points per query point found in the first iteration, we need to keep the k nearest neighbors as the initial *allNN*[] array.

We launch a block of k threads per query. First, each thread is in charge of generating candidate records for two data points. To reduce memory traffic, we compute the distance between integer coordinates from which the Morton code was generated, instead of reading the actual floating point coordinates from global memory. All $2k$ candidate records are stored as an array in shared memory.

Then, the threads within the same block together perform a block-wise parallel sort on the array. We use bitonic sort; we found experimentally that this was more efficient in shared memory than other parallel sort alternatives such as odd-even merge sort. Note that, in order to sort $2k$ elements, bitonic sort requires k threads.

Finally, we save the smallest k sorted nearest neighbor candidate records to global memory for each query. We do this with a coalesced write, producing the result array *allNN*[].

We implement these three parts (gathering, sorting, and saving) within one CUDA kernel. Bitonic sort is the dominant factor, so the kernel runs in $O(\log^2 2k)$ parallel time across k threads per query.

Subsequent iterations: In each subsequent iteration, we may need to merge some of the new candidates that we see among the $2k$ neighbors in Morton code order into *allNN*[]. Again, we do as much of this work in shared memory as possible in order to reduce global memory traffic.

We launch a block of $2k$ threads for each query. First, the block performs a coalesced read and transfers the current nearest neighbor array *allNN*[] from global memory into *currentNN*[] in shared memory.

Then, each thread generates a nearest neighbor candidate record *candidateNN* using the same routine as in the first iteration. As *currentNN*[] is sorted, each thread does a binary search into this shared array for its *candidateNN* in parallel. The binary search returns the location *loc*, at which this record wants to be inserted, i.e., $currentNN[loc-1] \leq candidateNN < currentNN[loc]$. If $loc = k$, which means *candidateNN* cannot be one of the first k nearest neighbors, we discard it. Also, if $candidateNN = currentNN[loc-1]$, which means that *candidateNN* is a duplicate, we discard it.

Both the k current candidates and at most $2k$ new candidates may be among the new k nearest neighbor candidates, so we need to compute their ranks based on the binary search results. Our approach employs a counter array, the atomic add operation, and an exclusive prefix sum computation for the counter array.

We allocate an array *counter*[] of size $2k$ in shared memory. Its odd entries, corresponding to records already in *currentNN*[], are initialized to 1, and its even entries, corresponding to new records waiting to be merged in, are initialized to 0. Each thread trying to insert its *candidateNN* then atomically adds one to the counter array entry $counter[loc \times 2]$ and remembers the previous value of this entry in a variable *offset*. Next, the entire thread block assigned to a query performs a block-wise parallel exclusive prefix sum on *counter*[], producing *counter_scan*[]. The odd entries

© The Eurographics Association 2012.

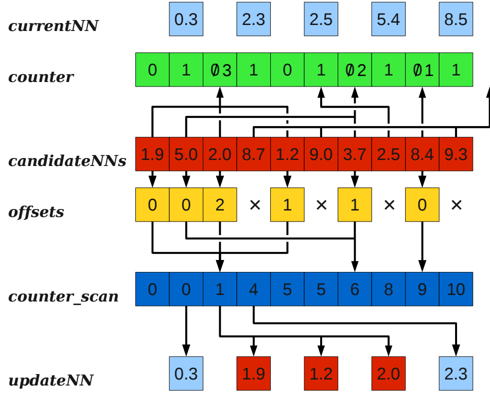

Figure 2: A graphical representation of Algorithm 2, with $k = 5$. Only the distances are shown; we assume that the index for the candidate with a distance of 2.5 matches the existing point, and it is eliminated as a duplicate.

in *counter_scan*[] are the ranks of the current candidates. The even entries are the starting ranks of the segments of new candidates waiting to be inserted into the same location. Each active thread computes the unique rank of its *candidateNN* by adding $counter_scan[loc \times 2]$ and *offset*.

Any nearest neighbor candidate record, new or old, associated with a rank greater than k is discarded. The smallest k records are saved in an array *updatedNN*[] in shared memory. Finally, *updatedNN*[] is written back to *allNN*[] in global memory in a coalesced manner.

Again, all of the above procedures are implemented in one kernel. Details of this kernel can be seen in Algorithm 2 and Figure 2, described at the block level. The binary search into a sorted array with size k takes $O(\log k)$ time. If all $2k$ candidates binary search into the same location, the atomic operations will be serialized, resulting in a worst-case time of $O(2k)$. The exclusive prefix sum for an array with size $2k$ runs in $O(\log 2k)$ time.

Since the new candidates inserted into the same location are in arbitrary order (there were no comparisons between them and their ranks are determined by the atomic operations), *allNN*[] at this moment is almost but not completely sorted. Hence, we launch another kernel to sort each array of k nearest neighbor candidates using a block-wise bitonic sort in shared memory. This proved to be faster overall than sorting each small segment separately.

4. Results

We implemented the sorting approximate nearest neighbors algorithm in CUDA on an NVIDIA GeForce GTX 480 with 1.5 GB of GPU memory.

Algorithm 2: Computing and maintaining k nearest neighbor candidates for a query in subsequent iterations

Input: Sorted array *allNN*[] of current k nearest neighbor candidate records for this query in global memory
Output: Updated array *allNN*[]
Allocate *currentNN*[] array of size k in shared memory
$currentNN[] = allNN[]$
Allocate *counter*[] array of size $2k$ in shared memory
foreach $counter[i]$ **pardo**
  if *i is odd* **then**
    $counter[i] = 1$
  else
    $counter[i] = 0$
  end
end
foreach *one of the* $2k$ *new nearest neighbor candidates of this query* **pardo**
  Compute and create *candidateNN*
  $loc =$ Binary search the location of *candidateNN* in *currentNN*[]
  if $loc = k$ *OR* $candidateNN = currentNN[loc-1]$ **then**
    Stop processing this *candidateNN*
  else
    $offset =$ Save the previous value in $counter[loc \times 2]$
    Atomically increment $counter[loc \times 2]$
  end
end
$counter_scan[] =$ Parallel in-place exclusive prefix sum on *counter*[]
Allocate *updatedNN*[] array of size k in shared memory
foreach *current candidate record* $currentNN[i]$ **pardo**
  $index = counter_scan[i \times 2 + 1]$
  if $index < k$ **then**
    $updatedNN[index] = currentNN[i]$
  end
end
foreach *one of the active new candidate records candidateNN* **pardo**
  $index = counter_scan[loc \times 2] + offset$
  if $index < k$ **then**
    $updatedNN[index] = candidateNN$
  end
end
$allNN[] = updatedNN[]$

© The Eurographics Association 2012.

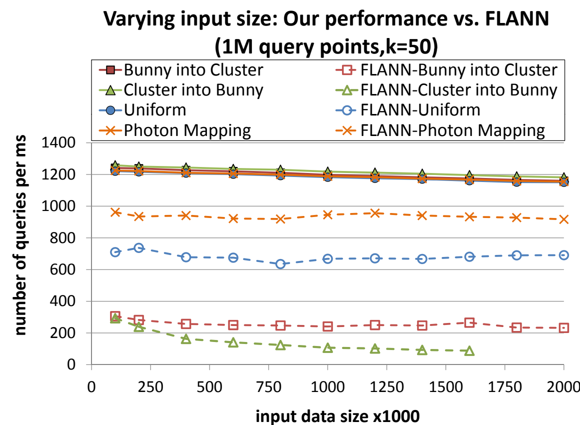

Figure 3: Number of queries per millisecond as we vary the size of the input data; 1M query points with $k = 50$. Higher results are better. We compare our results (solid lines) with that of FLANN (dotted lines) [ML11b].

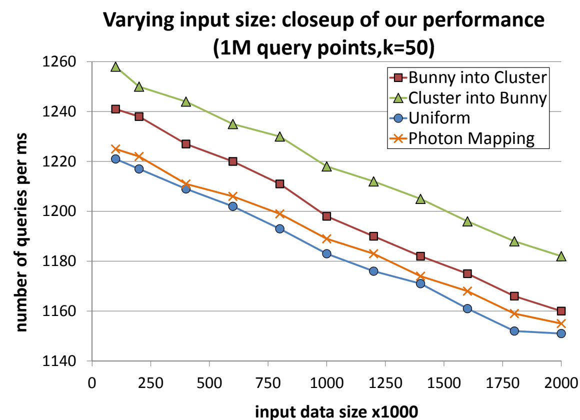

Figure 4: A closeup of only our performance seen in Figure 3. With fixed query size and k, our performance is linear with respect to data size. The only operation involving all of the data points is the radix sort.

Data Sets: We constructed synthetic test data sets for which the two distributions, of queries and data to be searched, are different; these cases are more difficult for ANN searching. We put one on a two-dimensional surface (the Stanford bunny) and the other in 25 tight Gaussian clusters in three-dimensional space. When the queries are clustered and the data is on the surface we called this Cluster into Bunny, and when the roles of the distributions were reversed, we called it Bunny into Cluster. We also considered the case in which both queries and data are uniform three-dimensional distributions.

We also generated sets of queries (eye-ray object intersections) and data (photons distributed on object surfaces) using

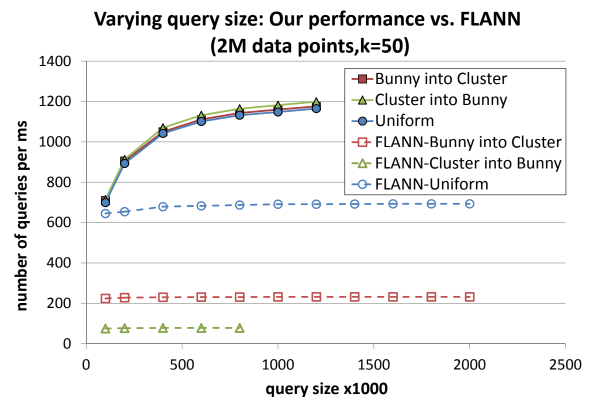

Figure 5: Number of queries per millisecond as we vary the size of the query points; querying into 2M data points with $k = 50$. Unlike FLANN [ML11b], which demonstrates consistent performance regardless of the number of queries sent, our approach exhibits significantly better performance with larger query batches.

the Optix progressive photon mapping demo [NVI12]. Figure 8 compares images created using the original demo and images created using a version modified to use our *kANN* algorithm instead of radius search, with the value of k tuned to achieve a similar level of brightness. Our experimental data sets are realistic in the sense that they produce images of comparable quality to the progressive photon mapping code.

Using these distributions, we compared our running times against the version of the FLANN [ML11b] approximate nearest neighbors library designed for low-dimensional point sets on the GPU [ML09b]. FLANN is a recent, well-known library for both high- and low-dimensional nearest neighbors; it has been incorporated into the Point Cloud Library (PCL) [PCL] and the OpenCV (Open Source Computer Vision) library [Ope]. Its CPU routines tune themselves to different data and query distributions, switching between two different data structures. Its GPU code uses a k-d tree, computed on the GPU and then searched using the standard priority search strategy [ML11a]. We compare our entire running time against only the search time of FLANN.

We find (Figure 3) that our throughput is between two and five times better, as measured by the number of queries answered per millisecond. While neither program is affected much by the overall size of the data being searched, the FLANN k-d tree is quite sensitive to the query and data distributions, with high throughput on uniform queries into uniform data, and less in the more difficult situations.

We also tried keeping the data size fixed at two million points, and varying the number of queries (Figure 5). We found that our performance suffered for smaller sets of queries, while for larger query sets the time required by the sorts was amortized over many more results. Unfortunately,

© The Eurographics Association 2012.

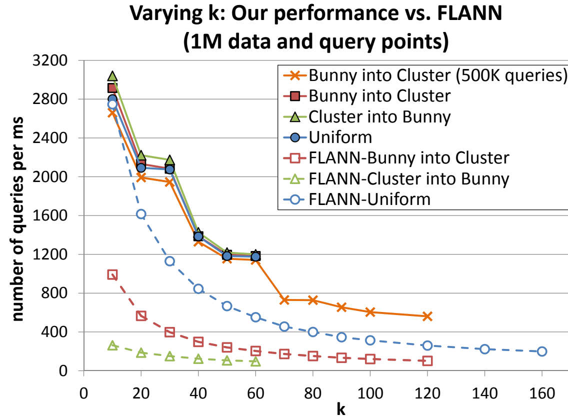

Figure 6: Number of queries per millisecond as we vary the size of k; querying 1M points into 1M input data points. Threads are allocated to queries in powers of two, with a single block servicing multiple queries. This results in nearly the same performance for $k = 20$ and $k = 30$, as 32 threads are launched per query in both cases. Similarly, performance at $k = 50$ is close to $k = 60$. We also show our performance for launching 500K queries into 1M data points (orange line) as we vary k up to 120. The smaller number of queries opens up more memory and as a result we can search for more nearest neighbors.

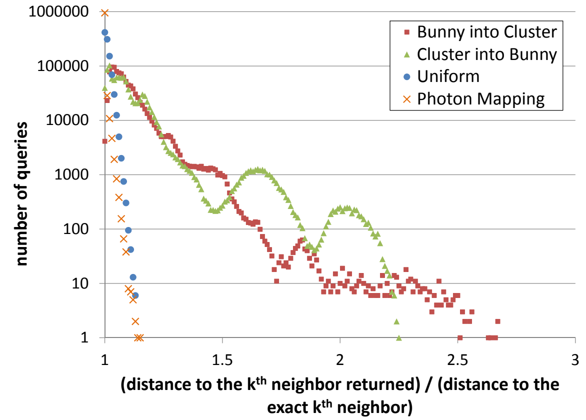

Figure 7: Our approach trades accuracy for speed. These histograms plot the ratio of the distance to the worst neighbor returned by a query to the actual kth nearest neighbor. Notice the log-scale on the y-axis. Our approach handles uniform random data in 3D with very small approximation error, including the data from our simple photon mapping scene. Cluster into Bunny has <3% of items with approximation error $\varepsilon > 1.5$, while Bunny into Cluster has 0.6%.

we ran out of memory beyond 1.2 million queries ($k = 50$, 2 million data points). FLANN failed at 800K queries on the Cluster into Bunny distribution, possibly because the poor distribution required very large priority queues in the search.

Finally, we considered throughput as a function of k, the number of nearest neighbors returned (Figure 6). For both our code and FLANN, this is the factor with the biggest influence on performance.

4.1. Approximation Factor

We achieve consistent throughput across all data sets at the cost of a small decrease in the quality of the approximation on the difficult distributions. On uniform distributions, including the distribution produced by the progressive photon mapping application, the worst query returned a kth nearest neighbor (for $k = 100$) which was at most $\varepsilon = 1.2$ times as far as the true nearest neighbor. Even on the difficult distributions, the number of queries which have approximation error ε falls off exponentially with ε, and none of the one million queries had an approximation error of more than $\varepsilon = 2.75$.

4.2. Other published results

The recent work of Leite et al. [LTF*12] uses a voxel grid, which is only efficient for uniform query and data distributions. On uniform data sets, we see that our performance is not as good at $k = 10$, but comparable at $k = 30$, and the strength of our algorithm is that we also handle very non-uniform distributions. For example, with data and query sizes of 50K they find $k = 30$ nearest neighbors for each query in 25 ms [LTF*12], for a throughput of about 2000 queries/ms; ours is about the same, even with 1 million query and data points (they also work on the GTX 480). The most natural performance comparison would be the GPU k-d tree nearest neighbor implementation [ZHWG08], which achieves real-time rendering of caustics, but it is not clear from that paper how many nearest-neighbor queries are actually performed per frame; queries far from the caustic were eliminated.

5. Discussion & Limitations

Actually returning the list of k nearest neighbors is not necessary for all applications. It would be useful to create a library function that could accumulate arbitrary functions over the k nearest neighbors without actually returning the items; this would save both space and time.

The algorithm of Liao et al. [LLL01] was intended for queries in high dimensions. As we have worked exclusively in 3D, we have only needed 64 bits in our Morton codes. In testing, we never find elements mapping to the same 64-bit Morton code. However, as the number of dimensions increases, the number of bits-per-dimension decreases; effectively, this reduces the depth of the octree.

Our algorithm is memory bound. In particular, two arrays

© The Eurographics Association 2012.

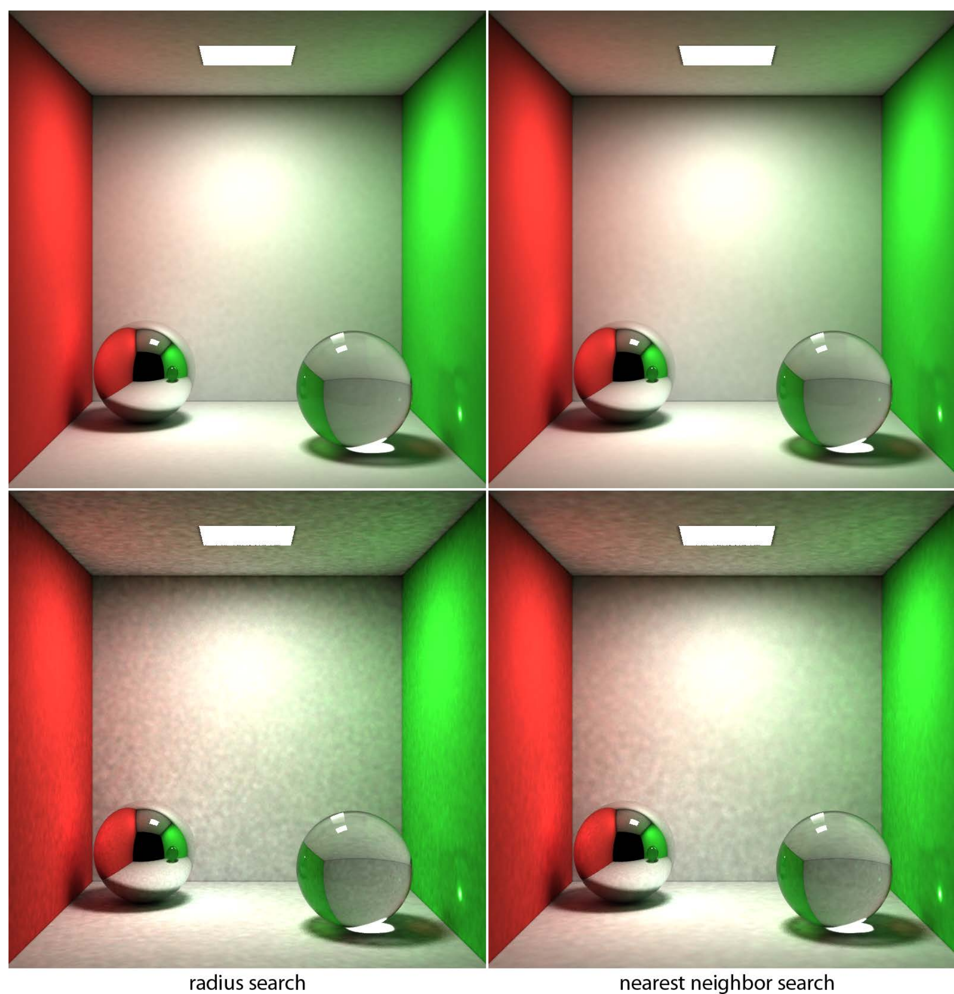

Figure 8: 100 iterations (top) and 10 iterations (bottom) of 640K photons per iteration being sent into a typical test scene. We used $k = 100$ from the *kANN* search (right), which is significantly larger than the average of 50 photons collected by each radius query from the progressive photon mapper (left), the scenes converge to nearly the same image (top). Note the difference in effective k is visible in the blotches on the far wall, but the caustic effects (particulary the effects on the right wall) show the same behavior. While we tuned k to produce similar progressive images, the combination of a relatively large k with Optix's large memory footprint meant our searches were memory-limited and overall the demo ran at about a third of the rate of the radius search version.

make up a majority of our memory usage. In between iterations, we store 64 bits of data per candidate point per query, and the final results are 32 bits per result per query; in our test case for $k = 50$ with 1M queries, these two arrays totalled 600 MB. Luckily, neither of these store the Morton codes, so an increase in Morton code size (in order to work in higher dimensions) would not incur a penalty here.

Acknowledgments

We gratefully acknowledge the support of NSF grant IS-0964357. We would also like to thank Andrew Davidson and Anjul Patney for their ideas and insight.

References

[AMN*98] Arya S., Mount D. M., Netanyahu N. S., Silverman R., Wu A. Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM 45* (Nov. 1998), 891–923. doi:10.1145/293347.293348. 2

[BDHK06] Bustos B., Deussen O., Hiller S., Keim D.:

© The Eurographics Association 2012.

A graphics hardware accelerated algorithm for nearest neighbor search. In *Proceedings of the 6th International Conference on Computational Science*, Alexandrov V., van Albada G., Sloot P., Dongarra J., (Eds.), vol. 3994 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, May 2006, pp. 196–199. doi:10.1007/11758549_30. 2

[BGTP10] BARRIENTOS R. J., GÓMEZ J. I., TENLLADO C., PRIETO M.: Heap based k-nearest neighbor search on GPUs. In *Congreso Espanol de Informática (CEDI)* (2010), pp. 559–566. 2

[Cha97] CHAN T. M.: Approximate nearest neighbor queries revisited. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry* (New York, NY, USA, 1997), SCG '97, ACM, pp. 352–358. doi:10.1145/262839.263001. 1

[Cha02] CHAN T. M.: Closest-point problems simplified on the RAM. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2002), SODA '02, pp. 472–473. 3

[FD09] FABIANOWSKI B., DINGLIANA J.: Interactive global photon mapping. *Computer Graphics Forum 28*, 4 (2009), 1151–1159. doi:10.1111/j.1467-8659.2009.01492.x. 2

[GDB08] GARCIA V., DEBREUVE E., BARLAUD M.: Fast k nearest neighbor search using GPU. In *Proceedings of the CVPR Workshop on Computer Vision on GPU*. IEEE Computer Society, Los Alamitos, CA, USA, June 2008, pp. 1–6. doi:10.1109/CVPRW.2008.4563100. 2

[HJ09] HACHISUKA T., JENSEN H. W.: Stochastic progressive photon mapping. *ACM Transactions on Graphics 28*, 5 (Dec. 2009), 141:1–141:8. doi:10.1145/1661412.1618487. 2

[HJ10] HACHISUKA T., JENSEN H. W.: Parallel progressive photon mapping on GPUs. In *ACM SIGGRAPH ASIA 2010 Sketches* (New York, NY, USA, Dec. 2010), SA '10, ACM, p. 54:1. doi:10.1145/1899950.1900004. 2

[KH12] KATO K., HOSINO T.: Multi-GPU algorithm for k-nearest neighbor problem. *Concurrency and Computation: Practice and Experience 24*, 1 (2012), 45–53. doi:10.1002/cpe.1718. 2

[KZ09] KUANG Q., ZHAO L.: A practical GPU based KNN algorithm. In *Proceedings of the Second Symposium on International Computer Science and Computational Technology (ISCSCT '09)* (Dec. 2009), Academy Publisher, pp. 151–155. 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384. 2

[LLL01] LIAO S., LOPEZ M. A., LEUTENEGGER S. T.: High dimensional similarity search with space filling curves. In *Proceedings of the 17th International Conference on Data Engineering* (2001), pp. 615–622. 1, 2, 3, 7

[LTF*12] LEITE P., TEIXEIRA J., FARIAS T., REIS B., TEICHRIEB V., KELNER J.: Nearest neighbor searches on the GPU. *International Journal of Parallel Programming 40* (2012), 313–330. doi:10.1007/s10766-011-0184-3. 2, 7

[MG11] MERRILL D., GRIMSHAW A.: High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters 21* (June 2011), 245–272. doi:10.1142/S0129626411000187. 1, 2

[ML09a] MCGUIRE M., LUEBKE D.: Hardware-accelerated global illumination by image space photon mapping. In *High Performance Graphics 2009* (New York, NY, USA, Aug. 2009), ACM, pp. 77–89. doi:10.1145/1572769.1572783. 2

[ML09b] MUJA M., LOWE D. G.: Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)* (2009), INSTICC Press, pp. 331–340. 2, 6

[ML11a] MUJA M., LOWE D. G.: *FLANN - Fast Library for Approximate Nearest Neighbors: User Manual*. The University of British Columbia, December 2011. URL: http://people.cs.ubc.ca/~mariusm/uploads/FLANN/flann_manual-1.7.1.pdf. 6

[ML11b] MUJA M., LOWE D. G.: FLANN-fast library for approximate nearest neighbors. http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN, 2011. [Version 1.7.1]. 2, 6

[MM02] MA V. C. H., MCCOOL M. D.: Low latency photon mapping using block hashing. In *Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, Sept. 2002), Eurographics Association, pp. 89–99. URL: http://portal.acm.org/citation.cfm?id=569046.569059. 2

[NVI12] NVIDIA: Nvidia OptiX ray tracing engine. http://developer.nvidia.com/optix-interactive-examples, 2012. [Version 2.1]. 2, 6

[Ope] OPENCV: Open source computer vision library. http://opencv.willowgarage.com/wiki/Welcome. 6

[PCL] PCL: The point cloud library. http://pointclouds.org/documentation/. 6

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003* (July 2003), pp. 41–50. 2

[PLM10] PAN J., LAUTERBACH C., MANOCHA D.: Efficient nearest-neighbor computation for GPU-based motion planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Oct. 2010), pp. 2243–2248. doi:10.1109/IROS.2010.5651449. 2

[QMN09] QIU D., MAY S., NÜCHTER A.: GPU-accelerated nearest neighbor search for 3D registration. In *Computer Vision Systems*, Fritz M., Schiele B., Piater J., (Eds.), vol. 5815 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 194–203. doi:10.1007/978-3-642-04667-4_20. 2

[WWZ*09] WANG R., WANG R., ZHOU K., PAN M., BAO H.: An efficient GPU-based approach for interactive global illumination. *ACM Transactions on Graphics 28*, 3 (July 2009), 91:1–91:8. doi:10.1145/1576246.1531397. 2

[ZGHG11] ZHOU K., GONG M., HUANG X., GUO B.: Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics 17* (May 2011), 669–681. doi:10.1109/TVCG.2010.75. 2

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics 27* (Dec. 2008), 126:1–126:11. doi:10.1145/1409060.1409079. 2, 7

© The Eurographics Association 2012.