Efficient Spatial Binning on the GPU

AMD Technical Report, February 2009

Christopher Oat* AMD, Inc.
Joshua Barczak† AMD, Inc.
Jeremy Shopf‡ AMD, Inc.

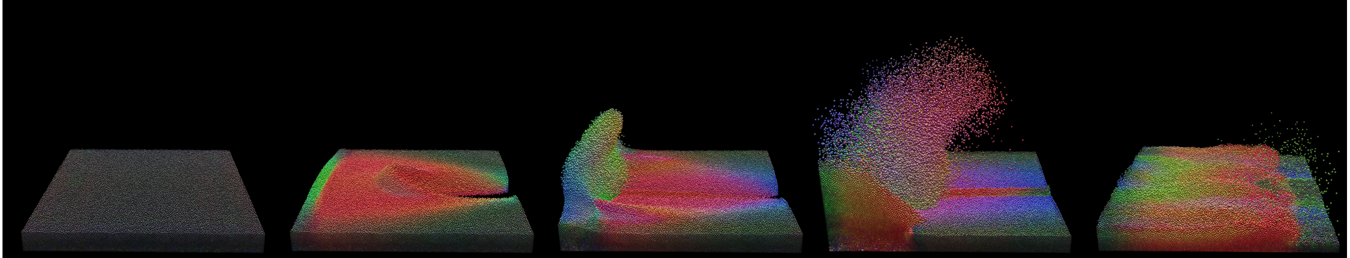


Figure 1: *Five still frames taken from a particle system in which more than 500,000 particles are simulated entirely on the GPU. Particles are shaded to indicate their velocity. The particles react to a user-applied force that pushes them towards a collision plane where they crash and roll back. The binning algorithm presented in this paper is used to construct a spatial data structure to facilitate nearest-neighbor searches for computing particle-to-particle collisions.*

Abstract

We present a new technique for sorting data into spatial bins or buckets using a graphics processing unit (GPU). Our method takes unsorted point data as input and scatters the points, in sorted order, into a set of bins. This is a key operation in the construction of spatial data structures, which are essential for applications such as particle simulation or collision detection. Our technique achieves better performance scaling than previous methods by exploiting geometry shaders to progressively trim the size of the working set. We also leverage predicated rendering functionality to allow early termination without CPU/GPU synchronization. Furthermore, unlike previous techniques, our method can guarantee sorted output without requiring sorted input. This allows our method to be used to implement a form of bucket sort using the GPU.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types; K.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

Keywords: GPGPU, spatial data structure, nearest-neighbor search, bucket sort

1 Introduction

Many applications require that an array of unsorted point data be sorted into spatial bins prior to being processed. For example, particle system simulations using the discrete element method (DEM) [Bell et al. 2005; Harada 2007] require a nearest-neighbor search to apply particle-to-particle repulsive forces. It is important to use a spatial data structure to accelerate nearest-neighbor searches, as a brute-force search on n elements will require an expensive $O(n)$ search per element. By partitioning the particles into spatial bins, the search can be limited to nearby particles, which dramatically reduces its computational cost.

In a GPU-based simulation, constructing these data structures on the GPU is necessary to maintain high performance. If these data structures are to be built by the CPU, particle positions must be transferred out of graphics memory into system memory, and the resulting data structure must be transferred in the opposite direction. In addition to consuming precious bus bandwidth, these kinds of hybrid GPU/CPU approaches require synchronization between GPU and CPU, which reduces utilization by introducing stalls.

In this work, we present a new technique for sorting point data into spatial bins using graphics hardware. Our method operates by repeatedly scattering point primitives into successive slices of a texture array. Dual-depth testing [Everitt 2001] is used to ensure that all elements are binned sequentially (and, as a side effect, causes them to be binned in sorted order). Unlike previous techniques, we can use geometry shaders to eliminate previously binned elements from the working set, which provides significantly better performance scaling. Our technique can be implemented using any Direct3D® 10 capable consumer hardware, without need for proprietary GPU compute APIs.

We begin by reviewing previous approaches to spatial sorting on the GPU. Next, we describe our data structure and how we construct and query it. Then, we outline three applications of our method in particle simulation, artificial intelligence, and bucket sort. We provide a performance comparison of our approach against a popular method for sorting data into a grid on the GPU and show that our method is more efficient. Finally, we provide our conclusions.

*e-mail: chris.oat@amd.com
†e-mail:josh.barczak@amd.com
‡e-mail:jeremy.shopf@amd.com

2 Related Work

Purcell *et al.* [2003] present two methods for sorting point data into grid cells on the GPU as part of their GPU-based photon mapping technique. Their first method sorts points by grid cell ID using a bitonic merge sort. This results in a sorted array in which points

in the same grid cell are listed consecutively. A binary search step constructs a lookup table that contains array offsets for quickly finding each grid cell's data in the sorted array. As an optimization to the bitonic merge sort, the authors describe a method they call *stencil routing* for storing points in grid cells.

Stencil routing is a multi-pass algorithm that scatters point data into grid cells using the vertex shader. When a point lands in a grid cell, the stencil value associated with the cell is incremented to prevent additional points from being written to the cell. This ensures that, if multiple points map to the same grid cell, they will not overwrite each other. A depth test prevents the same point from being stored in a cell multiple times. For the depth test to function correctly, stencil routing requires that its input data be in sorted order. Stencil routing must iterate over the entire data set once for each storage location within a cell (loop count is equal to maximum cell capacity).

Amada *et al.* implement a GPU particle system that constructs a nearest-neighbor map on the CPU [2004]. The authors identify the neighbor map generation and its transfer to the GPU as the main bottleneck of their system.

To overcome this bottleneck, stencil routing has been used to implement spatial data structures on the GPU, particularly for particle systems and particle-based rigid body simulations [Harada 2007; Harada et al. 2007b]. We anticipate that this will be one of the main applications of our technique. Subsequent work [Harada et al. 2007a] describes a sliced spatial data structure for point data on the GPU. This method employs a pre-pass over the point data to construct mapping functions that attempt to minimize wasted memory associated with unused cells in a uniform grid. A final stencil-routing step scatters the particles into cells within the grid and, thus, their method is related but orthogonal to our work. As we will show, binning is a more efficient way to scatter point data into grid cells; the scattering step in this algorithm would benefit from our binning technique.

Unlike stencil routing, our algorithm does not require the input data to be in sorted order. Additionally, our algorithm is more efficient because it removes points from the working set as they are binned and enables synchronization-free early termination once binning has completed. We are not aware of other techniques for scattering point data into grid cells, so we will use stencil routing as the basis for performance comparisons. We do not compare against the bitonic merge sort as Purcell *et al.* identify it as inferior to stencil routing in terms of performance.

Part of our binning method relies on a dual-depth test that was inspired by the depth peeling algorithm originally presented in [Mammen 1989] and later adapted for the GPU by Everitt [2001]. Depth peeling uses dual-depth tests to find the furthest fragment from the eye that is closer than all previous fragments at a pixel location, and was originally devised as a solution to order-independent transparency. On modern graphics hardware, the dual-depth test is performed by configuring the depth unit to perform one test and using one of the programmable stages to perform a second, complementary depth test. We use a similar dual-depth test to filter previously binned items during a given iteration of our algorithm.

3 Binning

Sorting data into bins on the GPU is challenging for a number of reasons. Current graphics APIs do not allow generalized atomic writes, so updating a linked list or placing a data element at the end of an array is non-trivial. The construction must be made as efficient as possible because real-time, dynamic applications will have to reconstruct this data structure on every update (*i.e.*, once every frame in a game). Querying the data structure must also be fast, since nearest-neighbor searches, one of our primary applications, will require multiple queries to gather all the binned elements near a particular point. We begin by describing the data structure itself, then describe how the data structure is queried, and finally explain how the data structure is updated.

As illustrated in Figure 2, our binning algorithm makes use of a 2D depth texture array and a single 2D color buffer to construct a data structure for storing items in bins. The color buffer is used to record the number of items in each bin (*bin load*). The depth texture array contains application-dependent key values that identify the binned items. A given 2D texel address in this array serves as a bin. A single bin is a set of texels which share the same 2D coordinates in successive texture array slices. Our binning algorithm guarantees that items are stored in bins, starting at the first slice of the array, in ascending order based on their key values. This fact may be useful for certain applications. For example, it can be exploited to perform a restricted form of bucket sort, as described in section 4.3. Sorted bins also allow applications to employ binary search when looking for a particular item in a particular bin. For applications that do not need any particular ordering, simply binning the item IDs is sufficient.

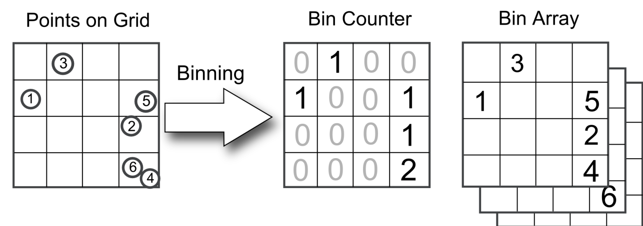


Figure 2: *Illustration of our binning data structure. Points are mapped to texel locations on a grid. The bin counter keeps track of the number of points in each grid cell. Point IDs are stored in sorted order in successive slices of the bin array.*

3.1 Queries

Fetching items from a particular bin is very straightforward. The load on a particular bin is determined by reading the corresponding texel from the bin counter, and the ith element in the bin is read by fetching from slice i in the depth texture array. Applications can use bin load and dynamic flow control to ensure that only occupied slots in a particular bin are fetched.

3.2 Building the Data Structure

To build our bin structure, we perform a series of rendering passes in which point primitives are used to represent each item to be binned. We refer to the data that is to be placed into bins as the *working set*. As items are placed into bins, they are removed from the working set. A given bin can only receive one item per iteration, so the update process may require multiple passes before the working set is eliminated. In the limit, the algorithm requires a number of rendering passes equal to the bin capacity.

We begin by clearing the bin counters to 0 to indicate that the bins are all empty. All of the slices of the bin array are cleared to 1.0. During each pass, the vertex shader determines which bin a particular item belongs in, and computes a corresponding pixel position for the point primitive. This effectively scatters the points into their corresponding bins. The point's depth value is set by mapping the key value onto $[0, 1)$. The rendering state is set such that the GPU's

depth unit will pass fragments that are *less than* the depth value stored in the depth buffer.

For each binning iteration, the corresponding slice of the bin array is used as a depth buffer. In all iterations after the first, the slice used in the previous iteration is bound as a texture for input, and the vertex shader rejects the item it is processing if its depth value is *greater than or equal to* the value stored in the previous slice. Points can be marked as "rejected" by setting their depth value to some value outside of the valid depth range (for example, a rejected vertex could have its depth value set to -1.0). The depth unit remains configured to *less than*.

Much like depth peeling, we are effectively implementing a dual-depth buffer that causes the point with the lowest value that is greater than the previously binned value to pass. Performing the *greater than or equal to* test in the vertex shader rather than the pixel shader allows us to avoid inserting clip/kill instructions into our pixel shader and allows the GPU to use its early-Z culling hardware. This dual-depth test causes the first iteration to store the lowest keyed item in each bin, the second iteration to store the second lowest, etc. To compute the bin counts, the pixel shader simply writes the iteration number into the bin count texture (1 in the first pass, 2 in the second, and so on), causing it to be updated whenever an element is binned.

3.2.1 Predicated Iteration

For situations in which the maximum bin load is low, it is possible that all the points will have been placed in bins before the binning algorithm has finished iterating. For example, if the maximum bin load is 2, then the binning algorithm can be terminated after 2 iterations. One way to detect that the working set has been eliminated is by using GPU queries to test whether any points pass the Z test (indicating that the working set is non-empty). Unfortunately, this kind of query would result in a CPU/GPU synchronization that would negatively impact performance.

The predicated rendering functionality provided by Direct3D 10 can be used to control the execution of the binning algorithm without introducing synchronization stalls. The draw calls for each iteration are predicated on the condition that the previous iteration resulted in items being scattered into bins. If no items are binned during a particular iteration, then we know that the working set has been eliminated and binning can safely terminate. Using cascaded predicated draw calls (each draw is predicated on the previous one) will result in the remaining draw calls being skipped. Thus, the GPU takes full responsibility for terminating the algorithm.

3.2.2 Stream Reduction

Using predicated iteration provides a performance gain by eliminating redundant rendering passes after all items have been binned. However, each pass still operates on the full data set, resulting in wasted processing for the items which that already been binned. This wasted work can be avoided by reducing the size of the point stream after each binning iteration, so the GPU only processes items that are still in the working set. This is easily implemented using geometry shaders.

To implement stream reduction, the point primitives are passed to a geometry shader that discards those points which have been flagged to indicate they failed the second-depth test, as described earlier. Points that pass the test are streamed into a buffer, which is used as input in subsequent iterations. This happens concurrently with rasterization and fixed-function depth testing. The Direct3D 10 call $DrawAuto()$ can be used to submit the reduced working sets without querying how many items are in the working set, thus avoiding another source of CPU/GPU synchronization. Using this technique, points will be removed from the working set on the next iteration, after they have been binned. Note that no points can be removed from the working set during the first iteration, so stream output and second-depth testing should not be applied to it.

3.3 Handling Overflow

An overflow condition occurs when the iteration count reaches or exceeds the bin array depth before the working set is eliminated. This can occur if too many items fall into a given bin. In practice, overflow often can be prevented by using a large enough number of bins (thus dividing the data set among many bins), or by simply increasing the bin capacity to accommodate the worst-case bin load.

Testing for overflow requires an additional iteration with a query to find the number of items that remain in the working set (recall that items are not removed from the working set until after they are binned). If any points pass the Z test during this final iteration, then overflow has occurred and must be dealt with accordingly. Depending on the application, it may be possible to stop iterating once the algorithm has reached the last bin array slice, process those items that have been binned, then continue to bin the remainder by wrapping around and rendering into the first slice of the bin array. Other applications may need to allocate a larger bin structure and simply try again.

4 Applications

Many applications require spatial binning. In this section, we describe GPU implementations of three applications that benefit from spatial binning. Particle systems have been used in many games and films for many different kinds of effects. We first describe a GPU particle simulation that uses binning for accelerating particle-to-particle interaction. Next, we describe a method for path planning that uses spatial binning to detect and avoid local collisions with other agents. Finally, we show how to implement a restricted version of bucket sort using our binning algorithm.

4.1 Particle Systems

The DEM is used to simulate the behavior of particle systems both on the CPU [Bell et al. 2005] as well as the GPU [Harada 2007]. We use binning to construct a spatial data structure to facilitate nearest-neighbor searches when computing particle collision forces. Using a spatial hashing function, $\mathbb{R}^3$ is implicitly subdivided into an infinite uniform grid, which is used to map particle positions to bin addresses. Each particle searches its bin and neighboring bins for other particles. Setting the grid cell size to be approximately the diameter of a particle allows us to limit our search to immediate neighbors while effectively limiting the maximum load on any particular bin. Particle-to-particle collisions are modeled using the spring and damping forces given by Equations 3 and 4 respectively [Harada et al. 2007a]. Collisions occur (and forces are computed) when the distance between two particles is less than the particle diameter, d.

If $\mathbf{x}_i$ and $\mathbf{x}_j$ are the positions of particles i and j with $\mathbf{v}_i$ and $\mathbf{v}_j$ denoting their velocities, then the relative position of particle j to particle i is:

$$\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i \tag{1}$$

The force imparted on particle i when colliding with particle j is computed as follows:

$$\mathbf{f}_{ij} = \mathbf{f}_{ij}^{spring} + \mathbf{f}_{ij}^{damp} \tag{2}$$

$$\mathbf{f}_{ij}^{spring} = -k_s(d - |\mathbf{r}_{ij}|)\frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|} \quad (3)$$

$$\mathbf{f}_{ij}^{damp} = \eta(v_j - \mathbf{v}_i) \quad (4)$$

Here, k_s is the spring coefficient and η is the damping coefficient. The total repulsive force, f_i, on particle i due to collision from particles in its neighborhood, N (particles in i's bin and its immediate neighboring bins), is computed as:

$$\mathbf{f}_i = \sum_{k \epsilon N} \mathbf{f}_{ik} \quad (5)$$

Figure 1 shows still frames from our GPU-based particle system with more than 500,000 particles. Every frame, the particles are binned into a 128x128x128 grid that is used to compute particle collisions.

4.2 Agent Avoidance

We also use our binning algorithm to conduct neighborhood searches for autonomous agents in a path-planning simulation (Figure 4). In this application, the simulation domain is of a known fixed size, so a uniform grid is appropriate. During path planning, each agent must conduct a search over the agents in its local neighborhood so it may alter its path to avoid collisions.

Each agent evaluates a number of fixed directions relative to the direction in which it wishes to move. Each direction is evaluated to determine the time to collision with nearby agents (Figure 3). Each direction is given a fitness function based on the angle relative to the desired direction and the time to collision. Time to collision is determined by evaluating a swept circle-circle collision test, in which the radius of each circle is equal to the radius of the bounding circle of the associated agent. The updated velocity (Equation 8) is then calculated based on the direction with the highest fitness (Equation 7) and the minimum time to collision in that direction.

$$fitness(\mathbf{d}) = w_i t(\mathbf{d}) + (\mathbf{g}_i \cdot \mathbf{d}) \cdot .5 + .5 \quad (6)$$

$$\mathbf{d}_i = \arg\max_{\mathbf{p}_i \in V} fitness(\mathbf{p}_i) \quad (7)$$

$$\mathbf{v}_i = \mathbf{d}_i \min(s_a, s_a t(\mathbf{d}_i) / \bigtriangledown ft) \quad (8)$$

w_i is a per-agent factor affecting the preference to move in the global direction or avoid nearby agents, $t(\mathbf{x})$ returns the minimum time-to-collision with all agents in direction $\mathbf{x}$, V is the set of discrete directions to evaluate, $\mathbf{g}_i$ is the preferred movement direction determined by a separate global path planner, s_a is the speed of agent a, $\bigtriangledown ft$ is time-delta since the last simulation frame, and $\mathbf{v}_i$ is the final velocity of agent i. This local avoidance method was previously described in [Shopf et al. 2008].

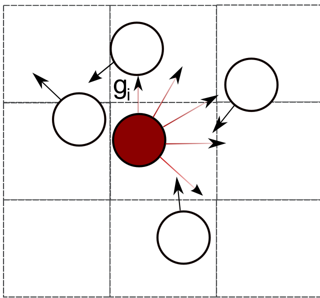


Figure 3: *Each agent evaluates a fixed number of potential movement directions based on the positions and velocities of agents in its current and adjacent bins.*

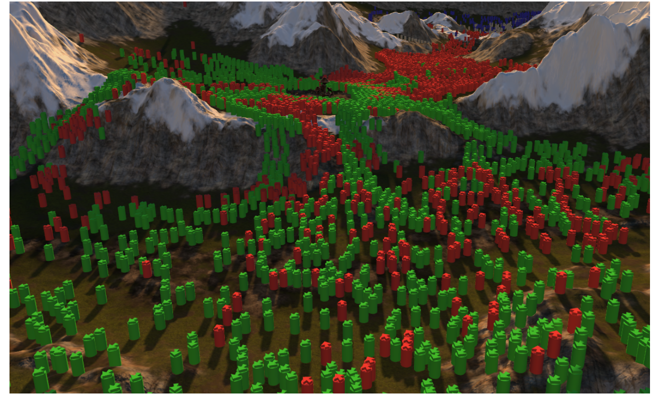


Figure 4: *Path finding with local avoidance is implemented on the GPU using binning. Agents move from goal to goal while avoiding local obstacles and each other.*

4.3 Bucket Sort

Our binning algorithm can be used to implement a restricted version of bucket sort. Because we cannot support duplicate entries in the working set (they would fail the dual-depth test and be removed during the reduction phase), this kind of bucket sort implementation is limited to random arrays of unique values. It is also required that a loose upper and lower bound on the input values be known. If the input's distribution is known, this distribution may be used to partition the input domain such that the expected bin load is the same for all bins. If the distribution is not known, then the bounds are used to partition the input domain uniformly.

The binning algorithm is executed using the input data as the working set. When the binning algorithm terminates, a final gathering pass is executed to collect the results. The gather pass takes a vertex buffer containing a single point for each bin. The bins are in ascending order in the vertex buffer such that the bin associated with the lowest partition of the input domain is first in the array. In the geometry shader, the associated bin's contents are fetched and are streamed out in ascending order. Because binning ensures that the data within a bin is in ascending order, no sorting need be performed in the geometry shader. The result of the gathering pass is an output buffer containing the input data sorted in ascending order.

5 Results

We evaluated our binning technique using synthetic tests that bin random sets of points based on their spatial locations. For each experiment, we used a fixed bin count and averaged the time required to bin 100 randomly generated point sets of a given size. We repeated this process for many different point set sizes and a few different grid sizes. To map a 3D grid onto a 2D texture, a *flat 3D texture* is used [Harris et al. 2003]. These experiments were conducted on a 3 GHz CPU with 2 GB of RAM and an ATI Radeon™ HD 4870 graphics card. We did not check for overflow during our experiments, but instead were careful to allocate enough bin capacity such that overflow was statistically unlikely to occur.

5.1 Binning Performance

We first examine the effect of our stream reduction and predication optimizations. The results are illustrated in Figure 5. In our experiments, we found that predication is an effective optimization, but also that stream reduction tends to be much more effective. We

also found that adding predication to stream reduction does not significantly change performance. This result is to be expected, since the predication merely skips draw calls which, because of stream reduction, would do no work to begin with.

While we found stream reduction to be generally effective, an interesting counter-example is visible in the 32x32x32 grid results in Figure 5. In this case, the number of bins is small, which implies that the average object count per bin (*bin load*) grows very quickly. As the bin load increases, fewer items are successfully binned during each pass, and the stream reduction gradually becomes less and less effective at eliminating work. Eventually, the extra bandwidth needed to repeatedly stream the active particles in and out of memory begins to outweigh the performance gained by removing items from the working set. In contrast, the naive binning algorithm does not need to repeatedly stream out its working set.

We found that this effect can be mitigated by simply delaying the start of stream reduction for a few iterations. With this modification, stream reduction is delayed until its eventual use will result in a large number of items being removed from the stream all at once, instead of removing them gradually and cycling the rest in and out of memory. This results in a significant performance improvement, compared to reducing on each iteration. This modification should not be applied blindly, as it may be harmful when the bin loads are low. We discuss a heuristic for choosing the number of iterations to be performed before reducing the working set in the next paragraph.

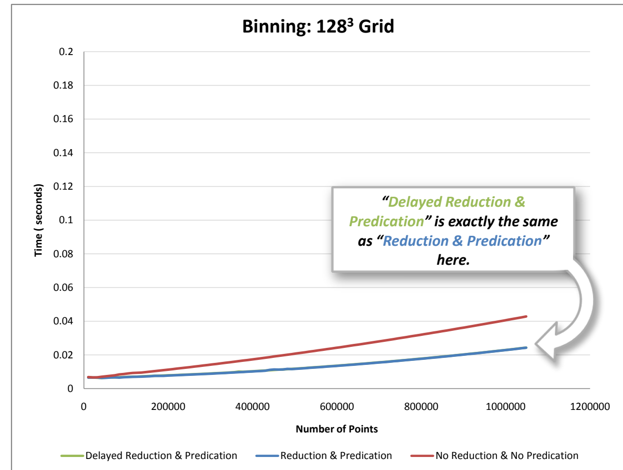
The question of whether and how to delay stream reduction must be decided on a case-by-case basis, but we can provide some general guidelines. Intuition suggests (and our results indicate) that stream reduction is most effective when the average bin load is low, and when the fraction of occupied bins (*bin spread*) is high. A high spread causes more items to be removed from the working set in each pass, and a low load ensures that the removed items represent a larger percentage of the total. In high-load, low-spread situations (many particles going into a few bins), stream reduction is at a serious disadvantage, and a delay is most likely to be helpful. Delays may also be beneficial in high-load, high-spread situations (many particles going into many bins).

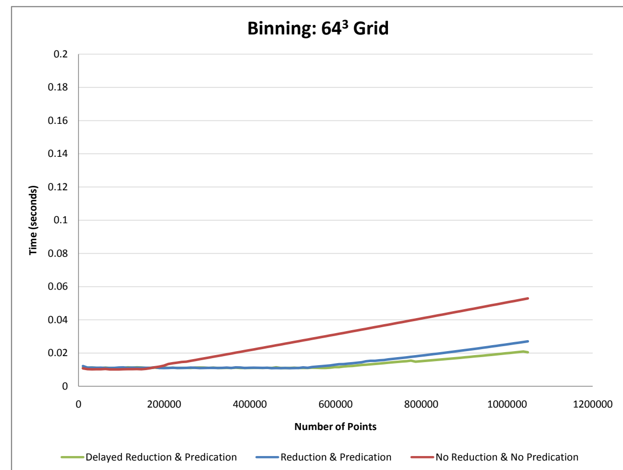
In our experiments, we obtained the best results by delaying stream reduction until the number of iterations exceeds the expected bin load (which, for a uniform distribution, is equal to the particle count divided by the bin count). We use delayed stream reduction in this fashion for all subsequent experiments.

5.2 Comparison to Stencil Routing

We also compared our binning technique to the stencil routing implementation described in [Harada 2007]. The results of these experiments are shown in Figure 6. In addition to the conventional stencil routing algorithm, we also tested an augmented version that uses our predicated iteration optimization to cull redundant rendering passes, as described earlier. This simple modification provides a significant performance improvement to the stencil routing algorithm, and provides more rigorous competition for our proposed technique. For these experiments, we used delayed stream reduction, as described above, using a delay count equal to the average bin load.

On the far left side of the plots, CPU performance is the bottleneck, and predicated stencil routing is extremely competitive. The stencil routing algorithm incurs significantly less driver overhead, because it does not need to switch render targets as often and does not need to ping-pong between Z buffers and stream output targets. In the limit, however, we find that our binning technique with stream reduction soundly defeats both variants of stencil routing.





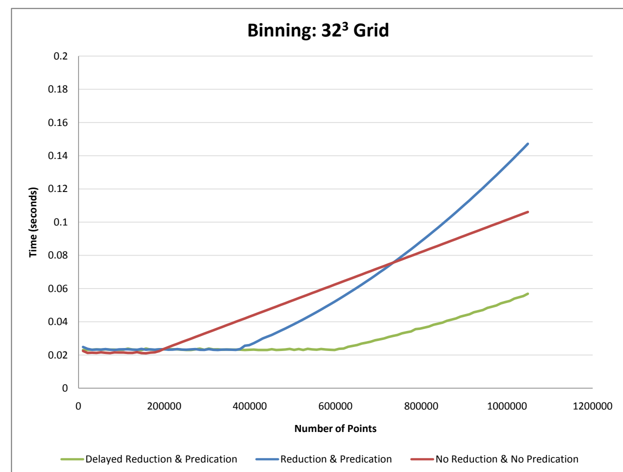
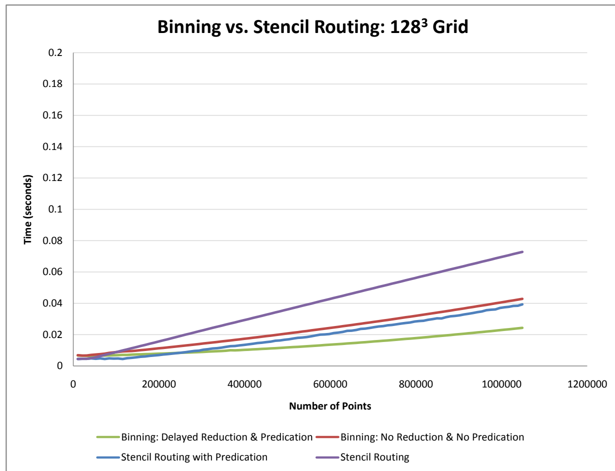
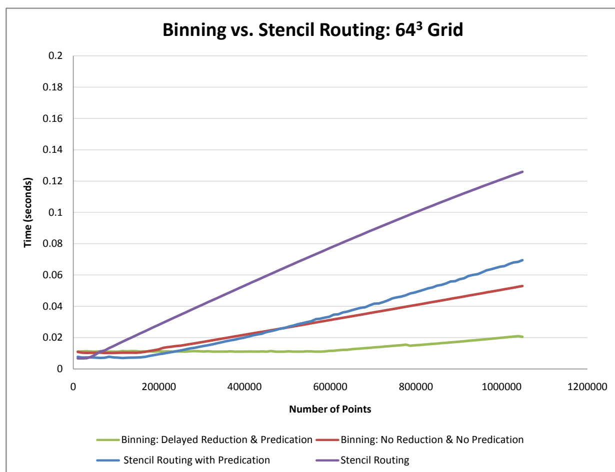


Figure 5: *Performance comparison between different binning optimizations on different uniform grid sizes. Delayed reduction has no effect on performance for the 128x128x128 grid (top) because our heuristic chooses not to delay in this case.*





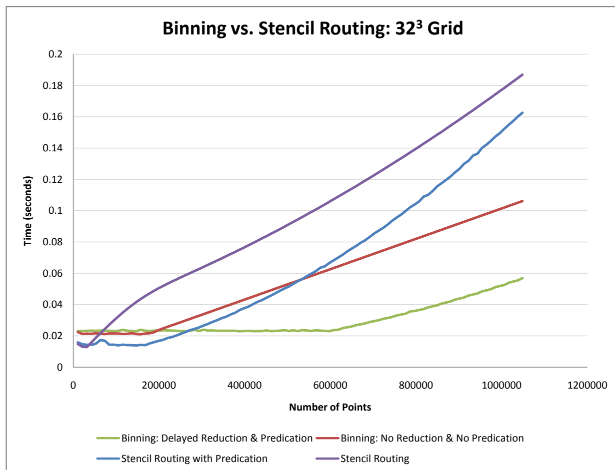


Figure 6: *Comparison between binning and stencil routing on different uniform grid sizes.*

6 Conclusion

We have presented a new method for sorting point data into spatial bins using graphics hardware with a standard graphics API. Our technique is more efficient than stencil routing because it reduces the working set as it iterates and stops iterating once it is done. We show how binning can be implemented without introducing CPU/GPU synchronization.

Acknowledgements

The authors thank the members of AMD's Game Computing Applications group for their thoughtful discussion and encouragement.

References

AMADA, T., IMURA, M., YOSHIHIRO YASUMURO, Y. M., AND CHIHARA, K. 2004. Particle-based fluid simulation on gpu. In *ACM Workshop on General-Purpose Computing on Graphics Processors*, ACM, New York, NY, USA.

BELL, N., YU, Y., AND MUCHA, P. J. 2005. Particle-based simulation of granular materials. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, New York, NY, USA, 77–86.

EVERITT, C., 2001. Interactive order-independent transparency. NVIDIA White Paper, May.

HARADA, T., KOSHIZUKA, S., AND KAWAGUCHI, Y. 2007. Sliced data structure for particle-based simulations on gpus. In *GRAPHITE '07: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, ACM, New York, NY, USA, 55–62.

HARADA, T., KOSHIZUKA, S., AND KAWAGUCHI, Y. 2007. Smoothed particle hydrodynamics on gpus. 63–70.

HARADA, T. 2007. Real-time rigid body simulation on gpus. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, Upper Saddle River, NJ, USA, ch. 29.

HARRIS, M. J., BAXTER, W. V., SCHEUERMANN, T., AND LASTRA, A. 2003. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 92–101.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics Applications 9*, 4, 43–55.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Association, 41–50.

SHOPF, J., BARCZAK, J., OAT, C., AND TATARCHUK, N. 2008. March of the froblins: Simulation and rendering massive crowds of intelligent and detailed creatures on gpu. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, ACM, New York, NY, USA, 52–101.