CGI2014 manuscript No. 86

Fast Sorting for Exact OIT of Complex Scenes

Pyarelal Knowles · Geoff Leach · Fabio Zambetta

June 2014

Abstract Exact *order independent transparency* (OIT) techniques capture all fragments during rasterization. The fragments are then sorted per-pixel by depth and composited them in order using alpha transparency. The sorting stage is a bottleneck for high depth complexity scenes, taking 70–95% of the total time for those investigated. In this paper we show that typical shader based sorting speed is impacted by local memory latency and occupancy. We present and discuss the use of both registers and an external merge sort in *register-based block sort* to better use the memory hierarchy of the GPU for improved OIT rendering performance. This approach builds upon *backwards memory allocation*, achieving an OIT rendering speed up to 1.7× that of the best previous method and 6.3× that of the common straight forward OIT implementation. In some cases the sorting stage is reduced to no longer be the dominant OIT component.

Keywords sorting · OIT · transparency · shaders · performance · registers · register-based block sort

1 Introduction

Transparency is a common graphics effect used for rendering particles, glass, antialiasing objects that might otherwise be drawn with the alpha test such as fences, foliage, billboards etc., and to see through objects but keep their spatial relativity to internal features or other objects.

Transparency rendering is a non-trivial problem in computer graphics. Unlike opaque rendering, where hidden surfaces are discarded using the z-buffer, all visible transparent surfaces contribute colour to the final image. Moreover the contribution is order dependent as each surface partially obscures farther ones, so surfaces must be sorted for a correct result.

Many 3D applications forgo accuracy and render transparency without sorting or only partially sorting, for example sorting per-object. This is because completely sorting per-triangle is expensive and difficult, requiring splitting triangles in cases of intersection or cyclical overlapping. Order-independent transparency (OIT) allows geometry to be rasterized in arbitrary order, sorting surfaces as fragments in image space. Our focus is exact OIT, for which sorting is central, although we note there are approximate OIT techniques which use data reduction and trade accuracy for speed [17, 13].

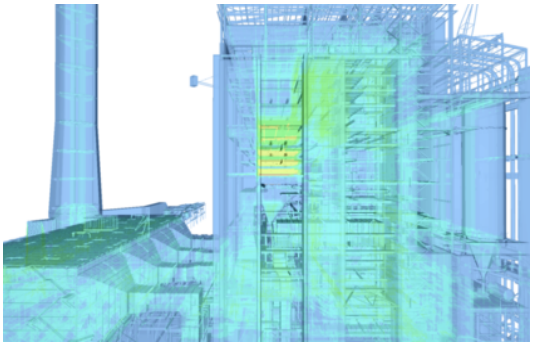


Figure 1: Power plant model, with false colouring to show *backwards memory allocation* intervals.

Complex scenes, such as the power plant in Figure 1, generally have high depth complexity, with many overlapping triangles. These scenes, which we term *deep*, are a focus as fragment sorting in OIT quickly becomes a bottleneck for them. This is partly due to the computational complexity of sorting being super-linear, unlike the other OIT operations which are linear. Thus, for deep scenes on the order of hundreds to a thousand fragments, the limiting problem in OIT performance is that of sorting many small lists — small in the context of sorting but still *deep* in relation to OIT. Current approaches perform sorting in local memory as global memory has much higher latency. Using local memory improves performance, however it has some drawbacks and further improvements are possible. Our core contributions are as follows:

RMIT University, Melbourne

- We show that OIT sorting performance is limited by local memory latency and low *occupancy* issues, discussed in Section 2.
- We use registers as another, faster, level of memory in the GPU memory hierarchy to store and sort fragments for shallow pixels.
- To apply this to deep pixels we introduce register-based block sort (RBS), an external merge sort, discussed in Section 3.
- We use *backwards memory allocation* [9] in combination with RBS to both provide a strategy pattern for various sort sizes and to improve occupancy. The speed increase from the combination is greater than the product of their individual increases.
- We implement RBS and other sorts in CUDA for a comparison to the GLSL implementations.
- Finally, we compare the performance of RBS with previous OIT compositing approaches.

Our platform uses OpenGL+GLSL and an Nvidia Titan graphics card.

2 Background

The first stage in exact OIT is *capturing* every visible fragment up to the first fully opaque one for each pixel. Because the GPU generates fragments in parallel the issue of race conditions is raised. Methods of fragment capture and construction of a data structure to store them are briefly discussed in Section 2.1.

After all fragments are captured and saved, they are sorted and composited in the *resolve* stage. Both sorting and compositing are performed in the same pass for OIT, meaning the sorted order is used only once and does not have to be saved, whereas other applications may benefit from saving the sorted fragments for reuse. The resolve stage is commonly implemented in a fragment shader with a single full-screen pass. The costly operations are (1) reading the captured fragments stored in global memory, (2) storing the data locally as is required for a fast sorting operation, (3) the sorting operation itself. The final step is an inexpensive compositing iteration through the sorted fragments.

The fragment capture and sorting operations used in OIT are fundamental building blocks which are increasingly also being used in other applications, for example translucency, ambient occlusion [1], CSG [6], metaball rendering [18] and indirect illumination [19,5,20]. The growing number of applications emphasises the importance of efficient approaches.

OIT uses the GPU in its normal graphics/rasterizer mode for capture in a fragment shader. The resolve stage is then commonly performed in another fragment shader, however an alternative is to use GPU compute capabilities via CUDA. CUDA ultimately runs on the same hardware, although its memory and programming model differs from that of shaders. For this reason we also compare our shader based approaches with CUDA implementations, and discuss different performance characteristics.

2.1 Fragment Capture

Transparency rendering is a mature concept, however its hardware acceleration is relatively new. Depth peeling [3] is among the first GPU accelerated OIT techniques. It and variations capture fragments in sorted order, but render the scene many times — proportional to the captured depth complexity. A more recent class of techniques capture fragments in just one rendering pass, made possible by the introduction of scattered writes and atomic operations to graphics hardware. Sorting fragments during capture, as is done for example in the *k-buffer* [2], Hybrid Transparency [13] and the *HA-buffer* [10], has some benefits including bounded memory, the possibility of fragment rejection and fragment reduction/merging. However, these methods are approximate or use many duplicate global memory accesses which are expensive for deep scenes and can be avoided by capturing all fragments first and sorting later.

There are two main competing approaches to capturing and saving all fragments during rasterization: *per-pixel linked lists* [22] (PPLLs) and *packed per-pixel arrays* (PPPAs) [15,11,12,14,8].

PPLLs are built with the use of a global atomic counter for node allocation. Nodes are inserted into lists via an atomic exchange of per-pixel head pointers. If insufficient memory is allocated for the nodes, data is lost or a re-render is required. PPPAs use an initial fragment counting pass and prefix sum scan to tightly pack, or *linearize* the fragment data. Despite requiring an extra pass of the geometry it has potential to improve capture performance for some scenes. There is no contention on a single atomic counter, the data location can be manipulated for better *coalescing* and there is no next pointer overhead. PPLLs are used for this paper as the focus is the resolve stage, but we mention both as OIT performance can be affected due to differences in memory layout.

After building PPLLs fragment data is read, sorted and composited in a single pass, usually by rendering a full screen quad and operating in a fragment shader. The straight forward and common approach, which is later used as a baseline for comparing tech-

niques, sorts the fragments in local memory using insertion sort, shown in Listing 1.

```
vec2 frags[MAX_FRAGS];

... populate frags from global memory

for (int j = 1; j < fragCount; ++j)
{
  vec2 key = frags[j];
  int i = j - 1;
  while (i >= 0 && frags[i].y > key.y)
  {
    frags[i+1] = frags[i];
    --i;
  }
  frags[i+1] = key;
}

... blend frags and write result
```

Listing 1: Baseline implementation with insertion sort.

Previous work [8,9] has shown significant benefits from the choice of the sorting algorithm, particularly that simpler $O(n^2)$ algorithms such as insertion sort perform better for shallower pixels and $O(nlogn)$ algorithms such as merge sort perform better for deeper pixels.

2.2 Coalescing

Sequential memory access is typically much faster than random access. Achieving sequential access in a SIMD environment is more complex as a group of threads must address memory in similar locations in parallel. Individual threads accessing separate contiguous memory can in fact damage performance.

Our observation is that PPLLs provide good memory locality, although this may seem counterintuitive. The atomic counter appears to increment sequentially for a group of SIMD threads. Since the threads write fragments (and next pointers) to the index given by the counter, writes will also be sequential across the group and hence coalesced. Although this does not hold for reading, we believe the locality for reading is not bad. Rasterization generates breadth-first layers of fragments (unlike a ray producing depth first intersections), and it is not unlikely that adjacent pixel reads will also coalesce as each thread steps through its linked list. We speculate variations in the raster pattern and small polygons inhibit this by adding spatial disorder to the links.

2.3 Registers

Arrays and variables in shader languages such as GLSL are placed in *registers* or *local memory*, both of which are on-chip as shown in Figure 2.

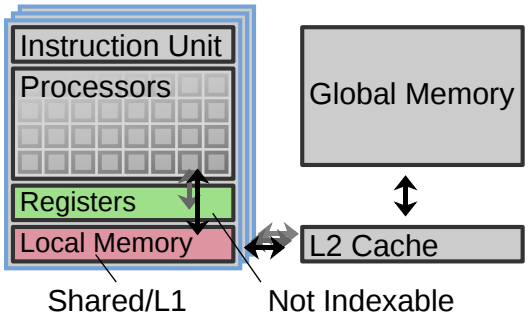


Figure 2: Broad GPU architecture.

Registers are much faster than local memory, which is much faster than global memory. Local memory used in shaders is equivalent to CUDA's *shared/L1 memory* in location. In comparison, CUDA's local memory is stored in global memory and cached in L1. This caching avoids the local memory occupancy issues of shader languages but hides the management of the memory.

Registers are significantly faster than local memory, and explicitly managing their use for performance reasons is an established concept in sorting on CPUs [16]. A drawback is the inability to dynamically index registers, as one would an array. Thus special programming techniques are needed to make use of registers as another, faster, level of memory.

Compilers can optimize an array into registers if indexing is known at compile time. Consider the two almost identical examples in Listing 2 with the intermediate SASS output from the GLSL compiler. The first loop iterates up to a dynamic value. It cannot be unrolled and registers cannot be used to store array `a`, seen by the `REP..ENDREP` loop and `lmem` local memory declaration. The second loop is bounded by a compile-time constant and is unrolled, providing compile time indexing and allowing the array to be placed in registers. Note the nested `IF` statements and `R0, R1, R2` declarations.

```
uniform int n;
int a[8];
...
for (int i = 0;
  i < n;
  ++i)
{
  a[i] = 42;
}
```

produces:

```
TEMP lmem[8];
...
MOV.S RO.x, {0, 0, 0, 0};
REP.S ;
SLT.S RO.z, RO.x, c[0].x;
SEQ.U RO.z, -RO, {0, 0, 0, 0}.x;
MOV.U.CC RC.x, -RO.z;
BRK (GT.x);
MOV.U RO.z, RO.x;
MOV.S lmem[RO.z].x, {42, 0, 0, 0};
ADD.S RO.x, RO, {1, 0, 0, 0};
ENDREP;
```

```
uniform int n;
int a[8];
...
for (int i = 0;
  i < 8 && i < n;
  ++i)
{
  a[i] = 42;
}
```

produces:

```
TEMP R0, R1, R2;
TEMP RC, HC;
...
SLT.S R2.y, {0, 0, 0, 0}.x, c[0].x;
MOV.U.CC RC.x, -R2.y;
IF NE.x;
SLT.S R2.y, {1, 0, 0, 0}.x, c[0].x;
MOV.U.CC RC.x, -R2.y;
MOV.S R0.y, {42, 0, 0, 0}.x;
IF NE.x;
SLT.S R2.y, {2, 0, 0, 0}.x, c[0].x;
MOV.U.CC RC.x, -R2.y;
MOV.S R0.z, {42, 0, 0, 0}.x;
IF NE.x;
SLT.S R2.y, {3, 0, 0, 0}.x, c[0].x;
MOV.U.CC RC.x, -R2.y;
MOV.S R0.x, {42, 0, 0, 0};
...
ENDIF;
ENDIF;
ENDIF;
```

Listing 2: Loop unrolling and register use.

The same loop unrolling and use of registers can be achieved by explicitly generating the source code for an unrolled loop, which then allows the use of individual variables instead of an array, for example `int a0, a1,` etc.

When using `lmem` the Nvidia GLSL compiler appears to pad array elements to 16 bytes while the four dimensional components of registers can be used individually. Vector component indexing may be used to better pack small elements in local memory, e.g.

```
ivec4 packedArray[8];
packedArray[i>>2][i&3] = 42;
```

although this increases number of shader instructions and we have experienced difficulty using it for OIT.

Potential performance gains from the example in Listing 2 include faster execution of unrolled code, lower latency access of registers and the increased *occupancy* from tight packing.

Unrolling loops can be difficult to do efficiently and can produce large amounts of code that can quickly reach shader program size limits. In some cases just unrolling the inner loops or simplifying the algorithm can allow use of registers and greatly reduce the number of instructions generated.

2.4 Sort Networks

A sort network is a set of hard coded compare and swap operations which sort the input in an *oblivious* manner. Sort networks apply naturally to execution in registers, whereas to get comparison sorts to do so needs modification and unrolling. However, a drawback of sort networks is a fixed input size. One solution to enable sorting variable sizes is to generate a sort network for each possible size, however this is not practical given shader instruction limits. Another is to use a larger network than is needed and pad the unused element keys with high values.

Sort networks are commonly discussed in the context of parallelization as some compare and swap operations can be executed in parallel [4]. This reduces the runtime complexity from the network's compare and swap count to its *depth*, the length of the minimum/critical path through the dependence graph. With CUDA's shared memory or compute shaders, collaboratively sorting a single pixel's fragments with multiple threads is an interesting idea, however we leave this to future work and instead discuss sort networks in relation to their implementation with registers.

2.5 External Sorting

External sorts were originally the solution to the problem of sorting data too large to fit in main memory. To reduce expense of sorting in external memory (such as disk), data is partitioned into blocks which can be sorted in faster memory. Blocks are then merged until the sort is complete. The concept abstracts to any system with a memory hierarchy, where successive levels are typically faster but smaller.

Similarly to main memory and disk external sorts, sorting algorithms have been adapted to better use levels of cache, and finally registers [21,4]. Unlike sorting small lists in OIT, these approaches commonly target sorting a single large amount of data.

2.6 Occupancy

As previously mentioned global memory has high latency. Instead of the processor blocking until completion, other threads are executed in an attempt to hide the latency and increase throughput. GPUs keep all active threads resident in registers and local memory, avoiding expensive thread context switches. *Occupancy* is a measure of how many threads can be active at once, and is limited by ratio of per-thread required to available resources. Note that every thread reserves the same

amount of resources, defined by the shader program. Low occupancy becomes a problem when a program uses enough resources that the number of active threads is too small for the GPU to adequately hide operation latency and be kept busy.

Poor occupancy is a significant performance factor in the resolve pass, which occurs particularly in the baseline OIT approach. Figure 3 shows an example where only a small number of per-pixel threads fit in local memory due to a conservative high allocation. To support more fragments, both local memory allocation and global memory access increase, reducing occupancy which amplifies the impact of slow global memory access.

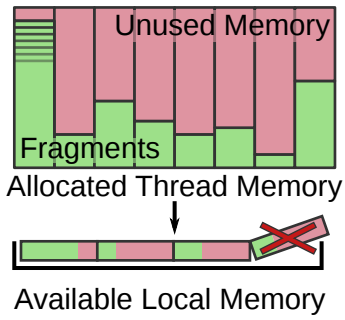


Figure 3: Local memory limited occupancy - threads declare a large amount of memory, limiting the total possible concurrently resident threads.

One strategy to address low occupancy in OIT is *backwards memory allocation* (BMA) [9]. Pixels are grouped into depth complexity intervals of powers of two, which in the case of PPLLs requires the small added cost of computing per-pixel fragment counts. Each interval is then processed in batches, using pre-compiled shaders with array sizes matching the maximum depth complexity of each interval. The stencil buffer is necessary to facilitate this, with its unique ability to mask a thread from being scheduled. A seeming alternative is to exit from shaders outside the correct interval, but in that case the threads are already active, consuming resources.

BMA is able to significantly improve occupancy and performance for shallow pixels, however it remains an issue for deep pixels. Another benefit of BMA is strategy pattern like per-interval shader optimizations with no branching overhead. In particular, a sort can be compiled to target just the fragment counts in each interval and corner cases such as zero fragments can be ignored.

CUDA has no stencil buffer, or equivalent method of masking thread execution, making a BMA type implementation for CUDA difficult. Also given the local memory model does not affect occupancy, BMA may only be of use to CUDA in cases of varying shared memory usage.

2.7 GLSL Functions

Functions are an important aspect of modular programming. For example a sorting function may take the array to be sorted as an argument, for use with different arrays. However, GLSL function arguments are *copied* in at call time and out before function exit, depending on the `in`/`out` qualifiers, to avoid potential aliasing issues [7]. This is a problem for large arguments such as arrays in which case the whole array is duplicated. Apart from the copy operation cost the local memory requirements double, also affecting occupancy and thereby performance. Using global scope variables, for arrays in particular, and generating code via macros is more predictable and stable.

3 Register-Based Block Sort

Our contribution applies to the resolve stage of OIT, without relaxing the constraint of an exact result. The resolve stage is limited both by high local memory latency and occupancy. To reduce the amount of local memory access, we introduce a sorting algorithm operating on data in registers, referred to as *register-based sort* (RS). As there are relatively few available registers, higher numbers of fragments are partitioned into blocks which are sorted using RS and then merged. We call this process, essentially an external sort type approach, *register-based block sort* (RBS). RBS has significant synergy when implemented with BMA, which is discussed in Section 4.

The first step to enable sorting in registers is unrolling an iteration through the pixel's linked list and reading fragments directly into registers, as Listing 3 demonstrates. Due to the constant arguments to `LOAD_FRAG`, the elements of the `registers` array will in fact be placed in register memory, as intended. This is only possible in the case of shallow pixels where all fragments can fit. Writing loops with constant bounds would allow the compiler to generate similar code. However, the most portable and reliable method to achieve this is by manually unrolling using explicit branching.

A straight forward way to sort values in the registers array is using sort networks, unfortunately they operate on a fixed number of elements and the number of fragments, `fragCount`, is variable. Alternatively, completely unrolling comparison sorts is not always prac-

```
vec2 registers[MAX_REGISTERS];

#define LOAD_FRAG(i) \
  if (PPLL_HAS()) \
  { \
    ++fragCount; \
    registers[i] = PPLL_LOAD(); \
    PPLL_NEXT();

int fragCount = 0;
PPLL_INIT(pixel);
LOAD_FRAG(0)
  LOAD_FRAG(1)
    LOAD_FRAG(2)
     ...
    }
  }
}
```

Listing 3: Reading PPLLs into registers.

tical as previously mentioned. Our solution lies in the middle, unrolling an insertion sort that uses swaps instead of the optimization in Listing 1 which shifts values and then performs just one swap. Unrolling just the inner loop of insertion sort, with bounds shown in Listing 4, allows registers to be used. There are much fewer instructions generated by not unrolling the outer loop and, compile time is less. However, unrolling the outer loop removes the need for the `i <= j` guard, replacing `MAX_REGISTERS-1` with the now-constant `j`, allowing each insertion to stop when complete and gives better performance.

The result is shown in Listing 5, which can also be viewed as a sort network that uses conditionals to improve performance and avoid padding. Outer conditionals allow the network to shrink to the appropriate size, for which an insertion sort network in particular works well. Inner conditionals stop inserting elements when the correct place has been found. Despite the overhead of branching this is found to be an improvement over a straight sort network, possibly due to the typical shallow weighted depth complexity distribution underfilling arrays. Finally, when combining with BMA, preprocessing directives are used to produce a sorting algorithm sorting up to just the interval size, rather than the global maximum.

```
for (int j = 1; j < fragCount; ++j)
{
  for (int i = MAX_REGISTERS-1; i > 0; --i)
    if (i <= j && COMPARE(i-1, i))
      SWAP(i-1, i);
}
```

Listing 4: Unrolling the inner loop of insertion sort.

Apart from insertion, there are other sort networks such as bitonic which have lower depth and computational complexity. However, dynamic sizing is an issue and our experiments show insertion to be faster.

As performance is significantly affected by fragment data size via occupancy, global memory transfer, sorting speed, etc., RGBA colour is packed into 32 bits with `uintBitsToFloat`. A fragment is then 64 bits, 32 colour and 32 depth which is the sort key. We find this to give sufficient colour precision and is a significant performance advantage. With fragment data any larger, indirect sorting using an index/pointer and depth is likely to be more efficient.

A minor drawback of unrolling loops is the considerable compile time. This can be alleviated to some extent by caching compiled shaders, for example via `ARB_get_program_binary` in OpenGL.

For deep pixels there are more fragments than can be placed in registers. This is where we use an external sort type approach, visualized in Figure 4. Fragments are partitioned into blocks of size `MAX_REGISTERS`, which we set to 32, copied to registers and sorted using RS as above. Sorted fragments are then written back to either local or global memory. Reading all fragments into registers before or progressively during the sort does not affect performance significantly.

With the fragment data partially sorted in blocks, a merge is performed, outlined in Listing 6. To avoid the heavy external memory bandwidth of pairwise merging, a single k-way merge is used. This is done with a repeated linear search through the *candidate* elements of each block until there are none left. While this is faster

```
//insert 2nd element if exists
if (fragCount > 1)
{
  if (COMPARE(0, 1)) //sort 1st and 2nd
    SWAP(0, 1);
  }
  //insert 3rd if exists
  if (fragCount > 2)
  {
    if (COMPARE(1, 2))
    {
      //first swap
      SWAP(1, 2);
      //not in order, so continue
      if (COMPARE(0, 1))
        SWAP(0, 1); //second swap
      }
    }
    ... insert 4th etc.
  }
}
```

Listing 5: Fully unrolled insertion sort using registers (RS).

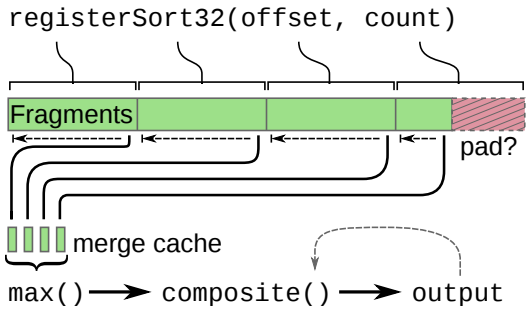


Figure 4: Sorting blocks of fragments per-pixel and compositing during an n-way merge.

for the scenes investigated, a heap may be beneficial for larger k. When implemented with BMA the maximum k, `MAX_K`, used for loop unrolling is never more than a factor of two above k.

After sorting blocks, the `registers` array is reused to cache the candidates from each block. A second set of registers, `blockIdx`, keeps track of the candidates' original locations. Reading blocks from tail to head the number of remaining elements in each block can be calculated implicitly, simplifying the corner case of a partially full final block.

As mentioned earlier, in OIT the sorted fragment data is only needed once, for blending, which is performed during the merge and the sorted result does not need to be saved.

```
int blockIdx[MAX_K];
for (int i = 0; i < MAX_K && i < k; ++i)
{
  //set blockIdx[i] to block tail

  //read initial candidates into registers[i]
}

for (int i = 0; i < fragCount; ++i)
{
  for (int j = 0; j < MAX_K && j < k; ++j)
  {
    //find fragment with max depth f
    //and its block index b
  }

  for (int j = 0; j < MAX_K && j < k; ++j)
  {
    //decrement blockIdx[b] and, if exists,
    // read data at blockIdx[b] into registers[b]
  }

  //blend f into final colour
}
```

Listing 6: Merging sorted blocks.

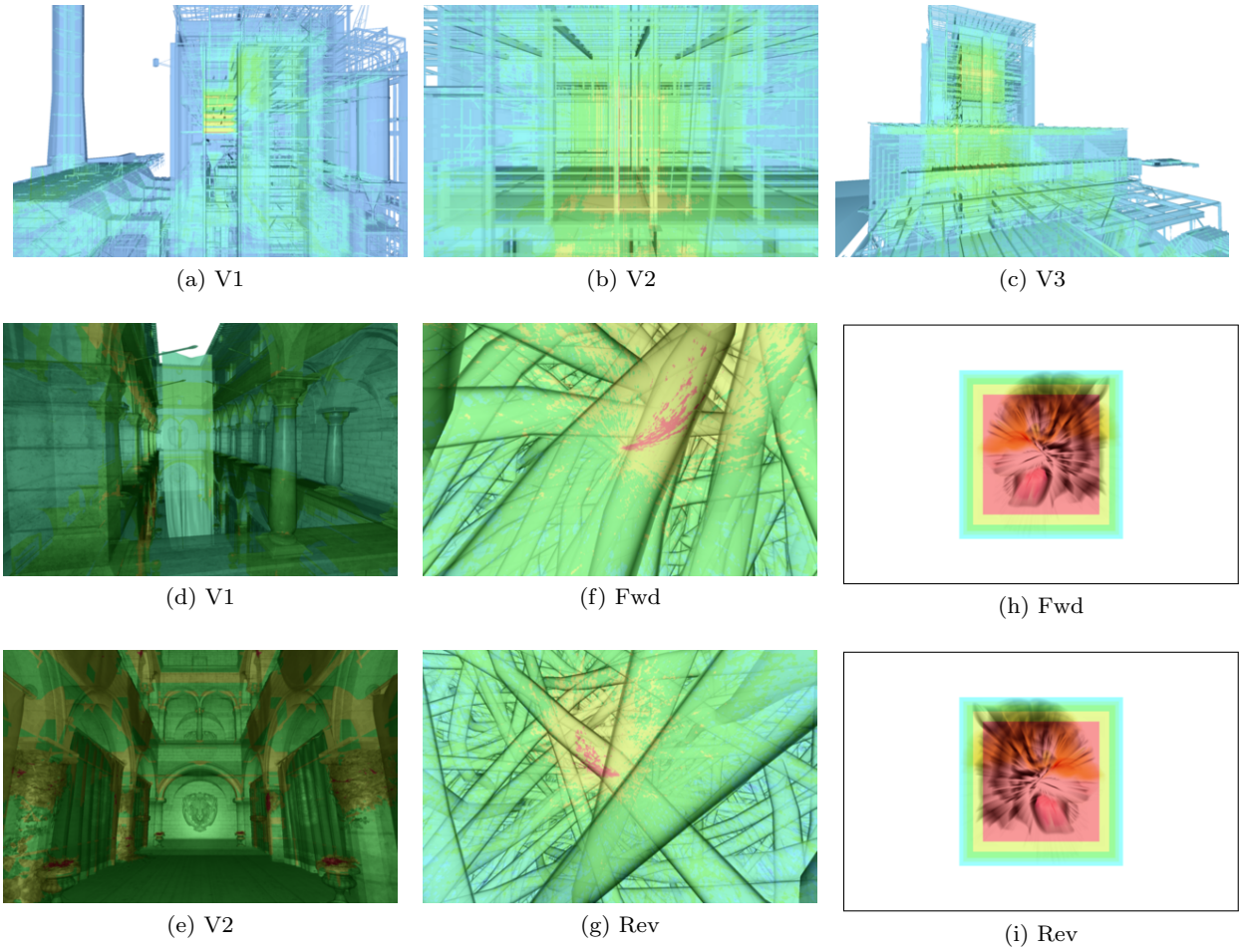
4 Results

In this section we compare and discuss performance of various exact OIT techniques, differing only in the resolve pass and keeping fragment capture using PPLLs constant. Our platform is a GTX Titan, a high end graphics card chosen for the trend of modern GPU architectures combining graphics and compute capabilities. All rendering is performed at 1920×1080 resolution.

We use a number of scenes and views, shown in Figure 5, to investigate the performance of different resolve stage techniques:

- The power plant, a model with high overall depth complexity and a depth complexity distribution characteristic typical of many scenes which is weighted towards the shallow end.
- The atrium, a moderately deep scene in which fragment capture is similar in time to the resolve pass.
- The hairball, with a more consistent depth complexity, also notable for its sorted geometry that is common for modelling packages to produce when writing a single mesh.
- A generated array of planes in sorted order, with front and back views to investigate best and worst case sorting.

For each view, we compare performance of RBS with previous OIT sorting methods. Due to a significant number of technique permutations, we use the following naming scheme. The BASE prefix refers to the use of a conservative-maximum sized local array in GLSL shaders without the use of BMA. With the suffix IS, BASE-IS is the baseline implementation using insertion sort. We add the option to dynamically choose merge sort for deep pixels (> 32), as in [9], referred to with the suffix IMS, i.e. BASE-IMS. Sorting is performed in registers in the RBS versions, which can be implemented using either local memory (L) to store the fragment blocks, or global memory directly (G) in which case BMA is unnecessary. Each of IS, IMS and RBS-L have also been implemented using CUDA, which stores its local memory data in global memory and caches in L1. To reduce occupancy issues in shaders caused by the local array size, these methods are also implemented with BMA (Section 2.6), e.g. BMA-IS, grouping pixels by depth complexity intervals and executing shader programs specific to each interval. Intervals are set to powers of two, starting at 8, i.e. 1–8, 9–16, 17–32, etc. Intervals 32–64 and up in BMA-IMS use merge sort. We set `MAX_FRAGS`, or limit BMA intervals, to the next power of two above the peak depth complexity, providing a generous near best case for the BASE implementations, as they do not adapt to shallow views of a



(a) V1 (b) V2 (c) V3

(d) V1 (f) Fwd (h) Fwd

(e) V2 (g) Rev (i) Rev

Figure 5: (a)–(c) views of the power plant, (d) and (e) views of the atrium, views inside the hairball producing (f) forward and (g) reverse sorted fragments, views of the sorted grid of textured planes giving (h) forward and (i) reverse order. False colour is used to show depth complexity relative to `MAX_FRAGS`.

deep scene. RS, without any block merging, can be used for shallow pixels below 32 fragments, sorting entirely in registers and removing the need for a local array. This occurs in view 2 of the atrium for BASE-RBS-L, however it is possible in all views when BMA is used for intervals up to 32, demonstrating the advantage of combining RBS and BMA.

A summary of the performance results is presented in Table 1. Both the total OIT rendering time (T) and just the time for the resolve stage (R) are shown. The resolve stage is of interest as this is where the compared techniques operate. Fragment capture time, given by T−R, is broadly constant for all views, although BMA requires per-pixel counts which add a small computation overhead. In addition to speedup factors compared to the baseline technique we show a "best of the rest" comparison for BMA-RBS-L, i.e. a comparison between BMA-RBS-L and the fastest non-RBS technique.

	Average		Best	
Total/Resolve	T	R	T	R
BASE-IS	1.0	1.0	1.0	1.0
BASE-IMS	1.0	1.0	2.1	2.2
BASE-RBS-L	1.7	1.8	4.9	5.4
RBS-G	1.8	2.0	5.4	6.0
CUDA-IS	1.6	1.8	4.6	5.4
CUDA-IMS	2.2	2.5	5.9	7.5
CUDA-RBS-L	2.4	2.8	7.8	9.3
BMA-IS	1.6	1.8	2.7	3.0
BMA-IMS	1.7	2.0	3.9	4.6
BMA-RBS-L	3.0	4.2	6.3	8.7
BMA-RBS-L*	1.1	1.4	1.7	2.1

Table 1: Average and best case speedup factors (×) compared to the baseline technique. BMA-RBS-L* is a comparison to the fastest non-RBS method. Both total (T) and resolve pass only (R) times are included.

As can be seen BMA-RBS-L is the fastest overall with an average total performance increase of 3.0× over the baseline across the scenes investigated, and best case of 6.3× where it is only beaten by another RBS-L implementation using CUDA. Compared to the fastest non-RBS techniques, BMA-RBS-L improves total performance by 1.15× on average, and up to 1.7× in the best case. Note that in practice only one technique can be used whereas this comparison is against the next best, which ever it may be. BMA-RBS-L more than doubles the resolve pass speed compared to BMA-IMS, from [9]. The improvement from RBS is primarily from using registers as much as possible, including the use of blocking and merging in the case of deep pixels.

Individually, BMA provides an average 1.6× speedup and BASE-RBS-L, without BMA, provides an average 1.7× speedup. Together though, RBS-L with BMA provide an average 3.0× speedup which is greater than the 2.6× product of their individual increases.

Detailed rendering times for each view are given in Table 2, with the fastest techniques for each view highlighted green. Table 3 shows the speedup factor over the baseline technique and in addition, the bottom row shows the speedup of BMA-RBS-L over the fastest non-RBS technique, highlighted blue. BMA-RBS-L improves OIT rendering speed for all views and achieves the highest speed in most.

Using just registers and global memory, RBS-G performs well for all but the hairball scene, and particularly well for the atrium and planes scenes, suggesting a hybrid which uses registers, local memory and global memory may be able to integrate the improvements.

The hairball and synthetic planes scenes test already sorted and reverse sorted fragment order, for which insertion sort sorts in n and n^2 operations respectively, explaining the large difference between Fwd and Rev views for the IS techniques. Although RS in RBS also uses insertion sort for the blocks, it differs by km and km^2 operations where m is the block size, `MAX_REGISTERS`, and is made possible by the k-way merge, which always uses kn operations. As $m^2 \ll n^2$ the performance difference between sorted directions for RBS is relatively small.

In the power plant and atrium scenes the resolve stage has been reduced significantly as the dominant component of the total time. Particularly in the atrium and view 3 of the power plant, where it is now similar to or less than capture time.

5 Conclusion

We have discussed a number of performance affecting characteristics of GPU based sorting algorithms in the context of OIT. We have shown how this can be leveraged by being aware of the memory hierarchy and manipulating low level operations from a high level shading language, to provide fast register-based sorting which is particularly beneficial for deep scenes. By unrolling loops and using a sort with sort network characteristics, fragments can be placed and sorted in registers for shallow pixels. This method can be extended by sorting blocks of fragments with registers and merging (RBS). Investigation into support for even larger and deeper scenes many include using a heap for large k-way merges and hierarchical merge passes when k becomes too big for a single merge, but is left for future work.

We have shown RBS alone provides an average 1.7× total performance increase over the baseline, the standard approach of insertion sort in local memory, for the scenes investigated. RBS works in synergy with BMA to provide faster fragment sorting than the product of their individual increases. Overall, BMA-RBS-L provides an average 3.0× and up to a 6.3× total OIT rendering performance increase compared to the baseline.

The resolve stage is now no longer the dominant factor in the mid-range cases such as the atrium scene and is on par with the capture stage even for deep scenes such as the power plant. In general, interactive or even realtime speeds are becoming a possibility for deep scenes that were previously impractical to render, both for exact OIT and other applications which also use sorted fragment data.

References

1. Fabian Bauer, Martin Knuth, and Jan Bender. Screen-space ambient occlusion using a-buffer techniques. In *International Conference on Computer-Aided Design and Computer Graphics*, Hong Kong, China, November 2013. IEEE.
2. Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba, and Cláudio T. Silva. Multi-fragment effects on the gpu using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 97–104, New York, NY, USA, 2007. ACM.
3. Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.
4. Timothy Furtak, José Nelson Amaral, and Robert Niewiadomski. Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 348–357, New York, NY, USA, 2007. ACM.
5. Jan Hermes, Niklas Henrich, Thorsten Grosch, and Stefan Mller 0002. Global illumination using parallel global ray-bundles. In Reinhard Koch, Andreas Kolb, and Christof Rezk-Salama, editors, *VMV*, pages 65–72. Eurographics Association, 2010.
6. Daniel Kauker, Michael Krone, Alexandros Panagiotidis, Guido Reina, and Thomas Ertl. Rendering molecular

Scene	power plant						atrium				hairball				planes			
View	V1		V2		V3		V1		V2		Fwd		Rev		Fwd		Rev	
MAX_FRAGS	512		256		128		64		32		256		256		128		128	
Peak DC	424		250		118		45		27		200		206		128		128	
Total Frags (M)	28		53		19		17		16		113		82		40		40	
Total/Resolve	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R
BASE-IS	530	511	490	463	63	45	17.7	11.6	11.6	6.2	436	400	782	750	77	67	448	439
BASE-IMS	442	420	430	402	105	86	19.7	13.7	12.7	7.2	877	839	681	650	197	187	213	202
BASE-RBS-L	422	399	285	257	62	44	14.4	8.4	8.9	3.5	495	459	423	392	77	66	92	81
RBS-G	—	—	269	240	47	29	16.7	10.7	8.9	3.5	768	731	963	933	42	32	83	73
CUDA-IS	116	94	311	281	47	28	24.9	18.5	23.1	17.4	796	757	452	420	40	30	298	287
CUDA-IMS	89	68	183	154	57	37	24.7	18.3	22.8	17.0	539	501	456	424	74	64	81	71
CUDA-RBS-L	86	65	226	197	52	32	27.2	20.9	25.4	19.7	642	604	626	593	56	46	58	47
BMA-IS	193	172	278	244	45	24	14.0	5.9	12.7	5.0	228	181	476	437	63	50	366	352
BMA-IMS	136	110	211	177	68	47	14.5	6.0	12.8	5.1	513	467	352	313	166	152	179	166
BMA-RBS-L	83	59	109	74	39	18	11.7	3.2	9.9	2.5	231	184	255	215	62	49	75	61

Table 2: Rendering time (ms) for compared OIT techniques. Times within 5% of the fastest are shown in green.

Scene	power plant						atrium				hairball				planes			
View	V1		V2		V3		V1		V2		Fwd		Rev		Fwd		Rev	
Total/Resolve	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R
BASE-IS	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
BASE-IMS	1.2	1.2	1.1	1.2	0.6	0.5	0.9	0.8	0.9	0.9	0.5	0.5	1.1	1.2	0.4	0.4	2.1	2.2
BASE-RBS-L	1.3	1.3	1.7	1.8	1.0	1.0	1.2	1.4	1.3	1.8	0.9	0.9	1.9	1.9	1.0	1.0	4.9	5.4
RBS-G	—	—	1.8	1.9	1.3	1.6	1.1	1.1	1.3	1.8	0.6	0.5	0.8	0.8	1.8	2.1	5.4	6.0
CUDA-IS	4.6	5.4	1.6	1.6	1.3	1.6	0.7	0.6	0.5	0.4	0.5	0.5	1.7	1.8	1.9	2.2	1.5	1.5
CUDA-IMS	5.9	7.5	2.7	3.0	1.1	1.2	0.7	0.6	0.5	0.4	0.8	0.8	1.7	1.8	1.0	1.1	5.5	6.2
CUDA-RBS-L	6.1	7.9	2.2	2.3	1.2	1.4	0.7	0.6	0.5	0.3	0.7	0.7	1.2	1.3	1.4	1.5	7.8	9.3
BMA-IS	2.7	3.0	1.8	1.9	1.4	1.9	1.3	2.0	0.9	1.2	1.9	2.2	1.6	1.7	1.2	1.3	1.2	1.2
BMA-IMS	3.9	4.6	2.3	2.6	0.9	1.0	1.2	1.9	0.9	1.2	0.9	0.9	2.2	2.4	0.5	0.4	2.5	2.6
BMA-RBS-L	6.3	8.7	4.5	6.3	1.6	2.5	1.5	3.6	1.2	2.5	1.9	2.2	3.1	3.5	1.2	1.4	6.0	7.1
BMA-RBS-L*	1.1	1.2	1.7	2.1	1.1	1.3	1.2	1.8	1.2	2.0	1.0	1.0	1.4	1.5	0.6	0.6	1.1	1.2

Table 3: Speedup factors (×) compared to the baseline technique. BMA-RBS-L* is a comparison to the fastest non-RBS method, shown in blue.

surfaces using order-independent transparency. In Eurographics Association, editor, *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, volume 13, pages 33–40, 2013.

7. John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL Shading Language, Version 4.40*. The Khronos Group, 1.1 edition, January 2014. http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf.
8. Pyarelal Knowles, Geoff Leach, and Fabio Zambetta. Efficient layered fragment buffer techniques. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, pages 279–292. CRC Press, July 2012. http://www.openglinsights.com/.
9. Pyarelal Knowles, Geoff Leach, and Fabio Zambetta. Backwards Memory Allocation and Improved OIT. In *Proceedings of Pacific Graphics 2013 (short papers)*, pages 59–64, October 2013.
10. Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. HA-Buffer: Coherent Hashing for single-pass A-buffer. Rapport de recherche RR-8282, INRIA, April 2013.
11. Jarosław Konrad Lipowski. Multi-layered framebuffer condensation: the l-buffer concept. In *Proceedings of the 2010 international conference on Computer vision and graphics: Part II*, ICCVG'10, pages 89–97, Berlin, Heidelberg, 2010. Springer-Verlag.
12. Jarosław Konrad Lipowski. d-buffer: letting a third dimension back in... http://jkl.name/~jkl/rnd/, 2011. Accessed 20 Feb 2014.
13. Marilena Maule, João Comba, Rafael Torchelsen, and Rui Bastos. Hybrid transparency. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 103–118, New York, NY, USA, 2013. ACM.
14. Marilena Maule, Joao L. D. Comba, Rafael Torchelsen, and Rui Bastos. Memory-efficient order-independent transparency with dynamic fragment buffer. In *Proceedings of the 2012 25th SIBGRAPI Conference on Graphics, Patterns and Images*, SIBGRAPI '12, pages 134–141, Washington, DC, USA, 2012. IEEE Computer Society.
15. Craig Peeper. Prefix sum pass to linearize a-buffer storage, patent application, microsoft corp., December 2008.
16. Abhiram G. Ranade, Sonal Kothari, and Raghavendra Udupa. Register efficient mergesorting. In *Proceedings of the 7th International Conference on High Performance Computing*, HiPC '00, pages 96–103, London, UK, UK, 2000. Springer-Verlag.
17. Marco Salvi, Jefferson Montgomery, and Aaron Lefohn. Adaptive transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 119–126, New York, NY, USA, 2011. ACM.
18. Lszl Szcsi and Dvid Ills. Real-time metaball ray casting with fragment lists. In Carlos Andjar and Enrico Puppo, editors, *Eurographics (Short Papers)*, pages 93–96. Eurographics Association, 2012.

19. Nicolas Thibieroz and Holger Grün. OIT and GI using DX11 linked lists. In *GDC 2010 Conference Session*. Presentation from the Advanced Direct3D Tutorial Day, 2010.
20. Yusuke Tokuyoshi, Takashi Sekine, Tiago da Silva, and Takashi Kanai. Adaptive Ray-bundle Tracing with Memory Usage Prediction: Efficient Global Illumination in Large Scenes. *Computer Graphics Forum*, 32(7):315–324, 2013.
21. Rajiv Wickremesinghe, Lars Arge, Jeffrey S. Chase, and Jeffrey Scott Vitter. Efficient sorting using registers and caches. *J. Exp. Algorithmics*, 7:9–, December 2002.
22. Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. *Comput. Graph. Forum*, 29(4):1297–1304, 2010.