Biased global illumination via Irradiance Caching and Adaptive Path Tracing on GPUs

Vladimir Frolov[(1,2)], Alexander Kharlamov[(1,2)] and Alexey Ignatenko[(1)]
(1)The department of Computational mathematics and cybernetics, Lomonosov Moscow State University, Moscow, Russia.
{vfrolov, ignatenko}@graphics.cs.msu.ru
(2) NVIDIA, akharlamov@nvidia.com

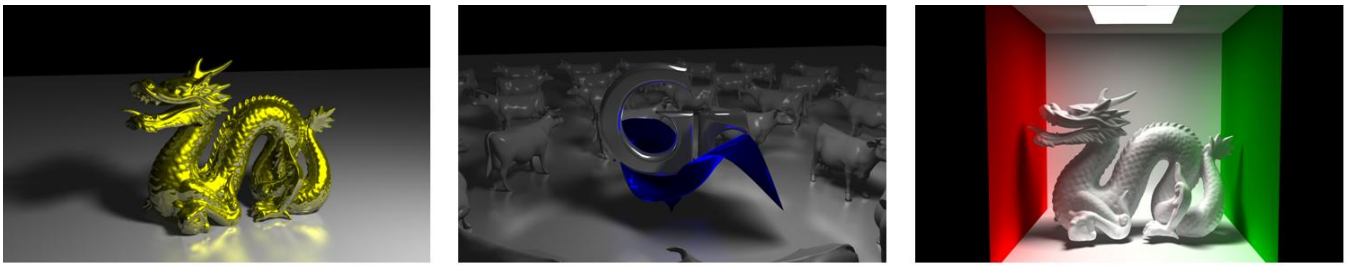

Figure 1. The presented hybrid approach uses irradiance caching to approximate smooth indirect lighting and path tracing for fuzzy effects such as soft shadows or glossy reflections. All screenshots where rendered at 1920x1200 resolution on a GTX 260 under 3 minutes.

1. ABSTRACT

This work presents an approach for biased photorealistic rendering on GPUs. The key idea is to combine irradiance caching with coherent adaptive path-tracing to maximize performance of the SIMD style execution.

Key words: *Global illumination, photorealistic rendering, GPU computing.*

2. INTRODUCTION

For the last decade Graphics Processor Units (GPUs) have made a great advance and became fully programmable processors. Since pixel shader 2.0 appeared, ray-tracing community has seen several successful implementations that used GPUs to perform ray-tracing. With the introduction of CUDA programming model this area of research experiences further growth. As more APIs are becoming available one can expect to see an increasing interest in this area. We have started our research at the point when CUDA C was the only option, however since programming model is shared between CUDA C / OpenCL and DirectX Compute APIs, we can generalize our results, and make some HW independent conclusions.

Several unbiased GPU photorealistic renderers are available now (IRay, Octane, Arion). However, we find that unbiased solutions have two significant drawbacks. Unbiased approaches usually perform computations in a brute-force manner. This means, that algorithmic complexity is higher, and in practice, unbiased approaches may use up to an order of magnitude more rays than alternative algorithms (such as irradiance cache). The second problem comes from a highly irregular nature of ray-tracing itself. Although each ray can be processed in parallel, the workload and data access pattern per each ray can be very different. This can lead to inefficient resource utilization. Although algorithmically more efficient biased approaches are more difficult to implement on GPU because of their complex nature and unbalanced work distribution. Our paper presents the research that we have performed on GPU efficiency for ray-tracing. We present a global illumination pipeline that uses irradiance cache with path tracing to quickly compute smooth indirect illumination, and soft shadows / glossy reflections.

3. RELATED WORK

3.1 GPU ray-tracing

Purcell et al. in [1] proposed to implement ray tracing pipeline in a set of fragment programs. Uniform grid has been used as an acceleration data structure due to simplicity. Data streaming was arranged so that a ray was generated within one kernel (this was implemented using a fragment program executed over a full screen quad), second kernel would perform grid traversal. If the ray hits a voxel with triangles, it is passed to a ray-triangle intersection kernel. If the intersection is not found, it is passed back to the grid traversal kernel. To manage the state of a ray (traversal, intersection, and shading) stencil test was set up respectively. The simplicity of this approach is appealing even today. It allows easier debugging, along with a more focused performance bottleneck analysis.

Foley et al. in [2] suggest two alternative approaches ray tracing kd trees. Since GPUs don't natively support stack, the authors suggest implementing one of the following techniques:

1) Modify kd tree nodes to support a reference to the parent node. This reference is used whenever a ray needs to backtrack to the parent node and to process a different sub-tree.
2) Traverse a kd tree until a non empty leaf is found. However if a ray doesn't intersect any triangles within this leaf, ray's origin is modified to skip the same leaf. The kd tree traversal is restarted and the whole process is repeated until intersection is found or the ray exits the scene.

Horn et al. in [3] suggest a modification of the restart algorithm. The idea is to keep a short stack in registers and resorting to restarts in fewer cases.

For bounding volume hierarchies Thrane et al. in [4] have shown the stackless traversal for efficient GPU implementations. Each leaf stores an escape index to the corresponding node as shown in Figure 2.

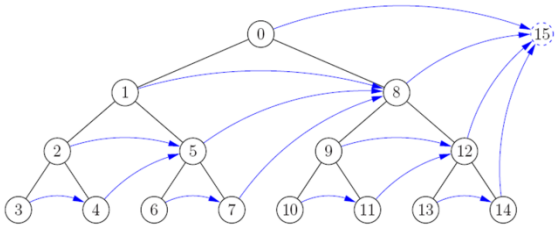

Figure 2 BVH structure for stackless traversal.

3.2 GPU Global Illumination

Wang in [5] presents an efficient approach for the global illumination using photon mapping on GPU. The key aspect of this work is to use irradiance cache with photon mapping and final gathering to quickly compute smooth indirect illumination. Direct lighting is computed using simple ray tracing and supports hard shadows from point light sources.

In [5] irradiance cache points positions were determined from the geometry discontinuities. Quad tree was used for adaptive subdivision to determine irradiance cache points positions. To evaluate the discontinuities, geometry metrics (screen space discontinuity in normals or positions) have been introduced. If the Quad tree corner point's difference exceeds certain threshold, then the subdivision of this node is required, otherwise no further subdivision is required. The similar approach was used in [6], but without final gathering. Direct illumination was computed with ray tracing and indirect - with photon mapping. This algorithm works well for caustics but produces noisy results for indirect lighting.

In [7] photon mapping was used to compute the full light equation. The drawback of this approach is strong low-frequency noise and dark edges. The low-frequency noise is a general problem of photon mapping. It can be removed with a final gathering step or filtering in object space. Dark edges appear on the borders of geometry because light is gathered only from the half of the disc but the result value divides on the area of the full disc.

Both filtering and final gathering introduce additional bias. Also we consider that final gathering has difficulties when two surfaces lie close to each other and there is a lack of photons in the scene. To eliminate the coming artifacts, the more complex and slow secondary final gathering should be used (as described in [8]). So, approaches from [5] and [6] may be a good choice for an interactive rendering but not for a photo realistic image synthesis.

McGuire and Luebke used in [9] the combination of the rasterization for direct lighting and CPU-based photon tracing with GPU-based photon splatting for indirect illumination.

4. RAY TRACING BOTTLENECK ANALYSIS

A naïve implementation of a ray tracer kernel will most likely yield poor results. To get the maximum out of any architecture a deep analysis of underlying HW is required. A good starting point with a focus on ray-tracing would be [10].

The naïve kernel that traverses spatial subdivision structure, does ray-primitive intersection and shading, shows the following signs of illness:

1) High register count.
2) Visual profiler shows 90% bottleneck in “instruction” unit.
3) Extreme amounts of local memory spilling
4) Divergent branching counter spikes.

These four issues in fact are tightly interleaved. They are causing a shifting bottleneck from instruction throughput to being memory bound.

1) Register count directly affects HW occupancy. Occupancy is the ratio between threads running on hardware to maximum possible threads amount. Occupancy can serve as a performance metric as well as a bottleneck indicator.
2) HW resources such as registers or shared memory are limited. Since all threads run in parallel, HW scheduler has to make sure there are enough resources for the launched threads.
3) Register count indicates the amount of registers that compiler allocates per thread.
4) Given an X registers per streaming multiprocessor and Y registers allocated by the compiler, the total amount of threads that can coexist on the streaming multiprocessor is X / Y. In particular NVIDIA Tesla 10 architecture has 16 * 1024 32bit registers per Streaming Multiprocessor. Thus a register count of 32 will leave room for no more than 512 threads.

With high register count exploding it is expected to observe poor HW utilization but this is not the final problem. GPUs rely on large scale threads parallelism to cover memory access latency. Poor occupancy can turn around and become a memory bottleneck.

Compiler will try and lower the register count by pushing and popping data into local memory. However local memory has the same latency as global memory. Analyzing CUDA PTX code one would local store and local load instructions happening repeatedly within a loop. This increases bandwidth pressure, and in addition to poor latency hiding can make application memory bound.

And finally divergent branching spikes are a sign of... divergent branches. When a block of threads executes it can diverge in two ways:

1) Different warps follow different code paths. This is perfectly fine, because it means no additional overhead except for condition evaluation.
2) Different threads within a warp follow different code paths. This for example can happen as soon as a single ray from a warp finds a non empty leaf and starts ray-triangle intersection. In this case HW will generate additional warp that will execute the code path. Partly threads will be masked out. The warps will be merged together as soon as the code paths merge back together.

4.1 Proposed ray tracing pipeline

Removing the bottlenecks can be tricky so we decided to implement a simple divide and conquer strategy. We split the ray tracing pipeline into the following stages (as shown in Figure 3):

1) **Ray generation kernel**. This can be a kernel that generates eye rays or secondary rays. Rays are packed into a linear list; direction and origin are stored in a structure of arrays fashion.
2) **Tree traversal kernel.** At this stage all rays are traced through a kd tree using stack in local memory. The idea is that the tree depth is usually defined beforehand during tree construction. This allows us to conservatively estimate stack size to be no more than tree depth. That stack stores an index

and *tfar* for kd tree. The output of this kernel is a list of non-empty leafs indices. This list serves as input to the next stage. To avoid constant switches between kernels and extra overhead, we traverse the tree until a number of suitable leafs is found. They are all written to a pre-allocated buffer. The stack in local memory is lost after traversal kernel completes, so in case we can't afford to allocate enough memory to keep enough leaf nodes in it, we can resume tree traversal using "restart" logic.

3) Ray-primitive intersection kernel accepts a list of rays as input along with a list of leaf candidates per each ray. If intersection was not found within leaf boundaries, then it sends the modified ray back to tree traversal stage. If the intersection point was found, then the ray is passed along.
4) Shadow kernel generates shadow rays and checks light visibility.
5) At shading stage we compute direct illumination with shadows.
6) The goal of the material kernel is to generate secondary rays. Typically it would generate reflection and refraction rays. These rays are sent to traversal stage.
7) Store result stage performs final light equation integration.

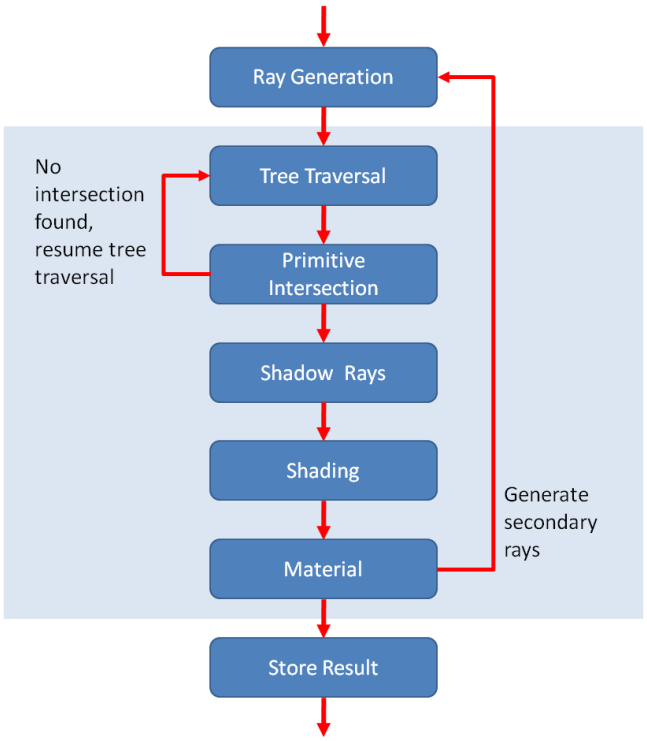

Figure 3 Ray tracing pipeline

This separation provides the following advantages:

1) Complete register usage comprehension. While kd tree traversal can fit into 16 registers and achieve perfect occupancy, ray-primitive intersection consumes 32 registers per thread. On Tesla 10 architecture we can't do better than with ½ occupancy. Shading turned out to be the most register hungry kernel due to the shading model peculiarities.
2) By reducing register pressure we have removed a significant portion of local stores and loads.
3) We have removed a significant portion of divergent threads. In fact since traversal / intersection and shading are all different kernels, the only divergence on a warp level is due to different time spent within the kernel loops.

There is still one remaining problem: varying workload per thread. Now, since we have different kernels, they serialize and intersection will not start until traversal is fully complete. This may well turn into waiting for a single thread that has the longest route through the tree. To reduce that further we implement a technique called persistent threads: each thread processes several rays instead of one. We divide the screen into blocks as shown in Figure 4 yet we launch a number of thread blocks that GPU can process in parallel. In this case each thread block has a fixed number of blocks to process. This is similar to ray pool described by Aila in [10], however the number of rays per each thread is fixed and we avoid using atomics.

Figure 4. Screen divided into blocks. Blocks of the same color are processed by the same thread block

Finally, after achieving a good performance of 50 Ms rays/sec on average we have combined the existing traversal – intersection kernels back, leaving just the shading outside.

We can now pass leaf nodes between two stages through a short list in shared memory. Each thread has a few private leaf IDs stored in shared memory (it's indexable and essentially free to use instead of registers).

Combining traversal and intersection into an uber-kernel with simple persistent threads management provides us with additional benefit of lesser kernel launch overhead, easier thread management. The uber-kernel is in general slightly (5-10%) faster than its separated analogue, and doesn't have any memory overhead. However both solutions are just variations of the same software load-balancing idea.

5. SUGGESTED APPROACH

For the fast global illumination solution a combination of distributed ray tracing and irradiance caching is commonly used. We suggest a similar idea: for fast and smooth indirect lighting we use irradiance caching technique as described in [11] and we use path tracing for other effects, such as soft shadows, glossy reflections and refractions, depth of field and motion blur [8]. The main motivation behind this step is to use simple iterative algorithm and avoid complex recursive nature of distributed ray

tracing. On the other hand due to highly divergent nature of path tracing we avoid using it for full light equation evaluation and consider it only for special effects.

However, this also increases the problem with the work distribution. The black circles in Figure 5 show simple regions that require several iterations for light integration to converge. The red squares show complex regions with soft shadows and reflections. While 10-20 iterations are enough for most pixels to converge to light equation solution, some areas of pixels require 100-1000 iterations to eliminate noise. This problem is solved on the CPU by processing each pixel until sufficient quality is achieved. On the GPU, however, this presents a challenge due to of the unpredictable workload.

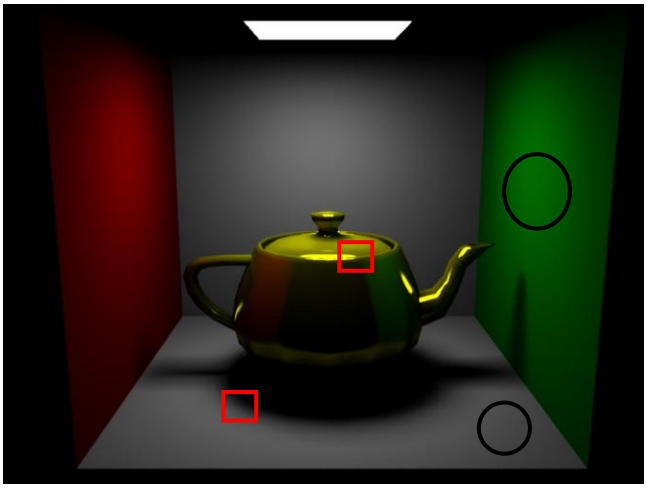

Figure 5. Teapot inside the Cornell box, direct lighting only.

5.1 Adaptive path tracing

For adaptive path tracing we split our screen into tiles as shown in Figure 6. Within the tiles we use Z-curve indexing for all 'per-ray' data (ray position, direction, and intersection info *etc*). This removes large address gaps for all pixels within a tile and enables an important bandwidth-saving optimization on NVIDIA hardware (coalesced memory reads and writes).

We define **`TMAX`** to be a number of tiles that we can process in parallel. **`TMAX`** depends on the amount of memory that we are prepared to allocate.

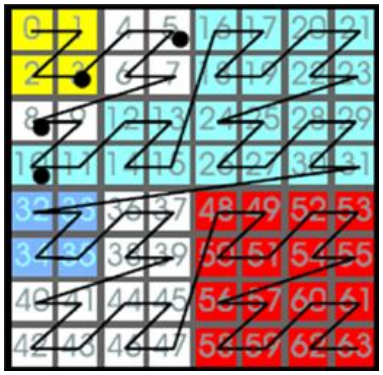

Figure 6. Z-Curve used to indexing pixels inside the tile.

We mark all tiles as active in the beginning of the rendering and add all tiles to the "active-tiles" list. In the example pseudo code below we assume tile size is 16x16:

```
var rays_per_pixel : integer;

procedure Adaptive_Path_Tracing is
  active_list: list of Tile;
  active_array: array (0..TMAX-1) of Tile;
  sz,i: inreger range 0..TMAX;
  tile : Tile;
begin

  subdivide screen to tiles;
  add all tiles to the active_list;
  rays_per_pixel := 1;

  while not active_list.empty():

    sz := min(active_list.size(), TMAX);
    active_array[0..sz] := active_list[0..sz];

    Process_Tiles_On_GPU(active_array, \
                         sz, rays_per_pixel);

    for i in 0..TMAX-1:
      tile := active_array[i];
      if not tile.finished():
        active_list.push_back(tile);
    end for

    if active_list.size() < TMAX * 0.5:
      rays_per_pixel *= 2;

  end while;

end Adaptive_Path_Tracing;
```

During the rendering process, some tiles finish earlier than the other. They are discarded from **`active_array`** and from **`active_list`** and replaced by new tiles from **`active_list`** if the last is not empty. When the number of active tiles is less than **`TMAX/2`**, we double the number of rays per pixel.

```
procedure Process_Tiles_On_GPU (
  active_array array (0..TMAX-1) of Tile,
  sz : Integer,
  rays_per_pixel : Integer
) is
  tile_size : Integer;
  rays_num  : Integer range 0..TMAX-1;
begin

  tile_size := 16*16;
  rays_num  := tile_size*rays_per_pixel*sz;
  assert (rays_num <= TMAX);

  on the GPU:
  generate initial rays from the eye \
                  according to the rays_per_pixel;
```

```
  trace exactly rays_num rays (paths in fact);
  sample result according to the rays_per_pixel;

end Process Tiles On GPU;
```

For large resolutions, like 1920x1200 our approach allows a good balance between memory consumption, performance and GPU workload.

Each tile is represented by a Tile structure. This structure is passed back and forth between CPU and GPU after each iteration.

```
type Tile is record
  index     : integer;
  max_diff : float;
  counter   : integer;
end record;
```

The 'index' field is an offset to a group of 256 rays in a GPU memory. It is used when fetching rays and sampling the resulting color.

To evaluate when a tile has converged we use the following approach: each ray accumulates partial sum of lighting integral into $\mathbf{sum}_{odd}$ and $\mathbf{sum}_{even}$ for all odd and all even passes of the path tracing (normalized e.g. divided by total ray count). After each iteration we compute **max_diff** value – it represents the maximum difference (among all rays in a tile) between these partial sums as shown in pseudo code below:

```
for i from 0 to 255 do:
     diff(i) := ||summ_odd - summ_even||_c
max_diff := max of all diff(i);
```

Since we are using quasi Monte Carlo integration, we expect that integral should converge at some point. Though, there is no good estimate for the number of iterations, however, $\mathbf{sum}_{odd}$ and $\mathbf{sum}_{even}$ should converge to the same value **sum**. This leads us to the conclusion that as soon as $\|sum_{odd} - sum_{even}\|_C < \varepsilon$, where ε is a certain threshold that represents error, than we can stop our integration process for this pixel. When the **max_diff** $< \delta$ we can stop integration process for all pixels in the tile and discard that tile from the **active_list**.

Finally **counter** represents the number of passes that have been completed already.

5.2 Irradiance cache

Our irradiance cache implementation is very similar to the Wang's implementation in [5]. For each pixel we compute a surface position and normal. We do that on the GPU. Next, we construct a quad tree in screen space as in [5]. We used an initial size of 32 pixels both in horizontal and vertical directions. Each 32x32 quad is subdivided with a quad tree and geometry discontinuity is computed between quad tree nodes. When the discontinuity is less than a threshold, we do not perform further quad tree subdivision.

The chosen pixels correspond to the irradiance cache points in object space. At each point we generate a set of rays to sample hemisphere and compute indirect illumination. To have more coherent groups of rays we subdivide hemisphere into sectors and generate $32 \times k$ rays for each sector where $k \geq 1$. We do that on CPU in tangent space. On GPU we transform directions from tangent to object space to get correct hemisphere sampling. Next, we construct a multiple-reference octree as described in [11].

We implement interpolation algorithm, quality metric and stackless octree look-up as described in [12]. It seems that stackless approach should be efficient on GPUs. However we find that multiple-reference octree is not the best solution.

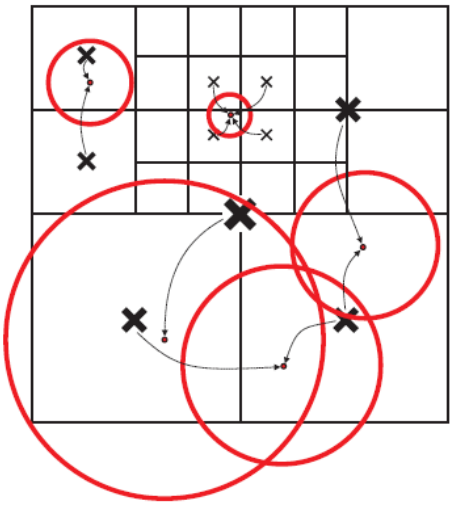

Figure 7. Multiple reference octree as described in [11].

The key advantage of the multiple reference octree is a stackless 'root-to-leaf' look-up algorithm. To find all points in the given sphere we can traverse tree from the root to a leaf and there is no need in stack or recursion. But the price for such simplicity is multiple references: each point can be referenced from multiple octree nodes. During octree construction or point insertion, we need to add each irradiance cache point P_i to all octree nodes that intersect with sphere centered at P_i, with radius equal to the search radius. The number of references in this approach can be a 5x-10x times larger than the number of points. On GPU this leads to dependent texture fetches and cache trashing. In our implementation octree look-up costs as much as a ray-tracing part. We suppose that kd-tree Wang's approach from [5] will be more efficient than the multiple-reference octree from [11], this is one of the future research strategies.

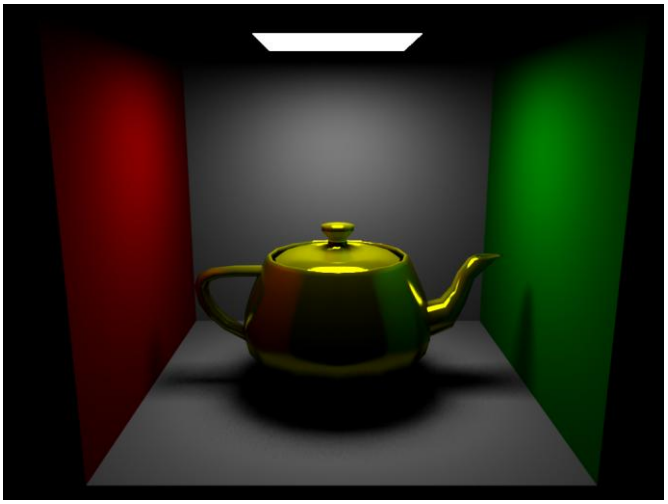

Figure 8. VRay; Core 2 Quad, 6600; 62 sec in 1024x768

Figure 9. VRay; Core 2 Quad; 6600; 153 sec in 1024x768

5.3 Complete solution

We have experimented with separate kernel architecture. Our motivation not to use uber-kernels in this case was:

1) Uber-kernels are bound by their most heavy part. For example, if we have a complex shading code, it can affect on the ray tracing performance and downgrade it;
2) It is possible that several pixels require thousands of rays and all these rays for each pixel will be traced in series. We suggest a solution to trace them in parallel with different threads;
3) Last but not least, for the complex code, like ray-tracing with different shading techniques and materials, separate kernel architecture is much more convenient than uber-kernel, especially for profiling and debugging reasons.

At each bounce of path tracing we compute direct illumination by tracing shadow rays towards each light and look-up indirect illumination from the irradiance cache. To reduce octree look-up cost we trace several shadow rays for each shadow sample. This solution allows us to do less look-ups on the regions with complex soft shadows.

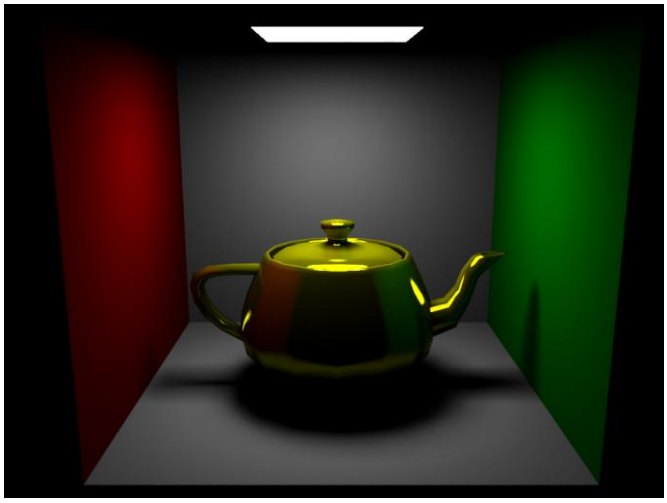

Figure 10. Our Implementation; GTX260; 15 sec in 1024x768

Figure 11. Our implementation; GTX260; 31 sec in 1024x768

6. RESULTS

Our ray tracing implementation runs with 30-50M rays per second on the 'Conference Room' scene and GTX260 GPU. It corresponds to the other works related to GPU ray tracing: [10], [13]. We use a SAH kd tree to accelerate ray-triangle intersection. We compared our renderer with VRay on the simple scene both with direct and indirect illumination. Our implementation shows good performance scaling for large geometry and higher resolutions (Fig 8-11).

VRay is a commercial renderer and we don't know exactly how it works, so it is hard to make a precise per-pixel image comparison. On the middle-level hardware our implementation performs up to 4x times faster than VRay. As we did not pursue the aim to make a per-pixel comparison, Figures 9 and 11 are slightly different but our original image contains less visible noise.

For the Dragon model irradiance cache construction takes 4 seconds. For the simple scenes, like a teapot in Cornell Box, it takes less than 0.5 sec.

All our Demos, videos, comparisons and screenshots can be found at http://ray-tracing.com (English) and http://ray-tracing.ru (Russian).

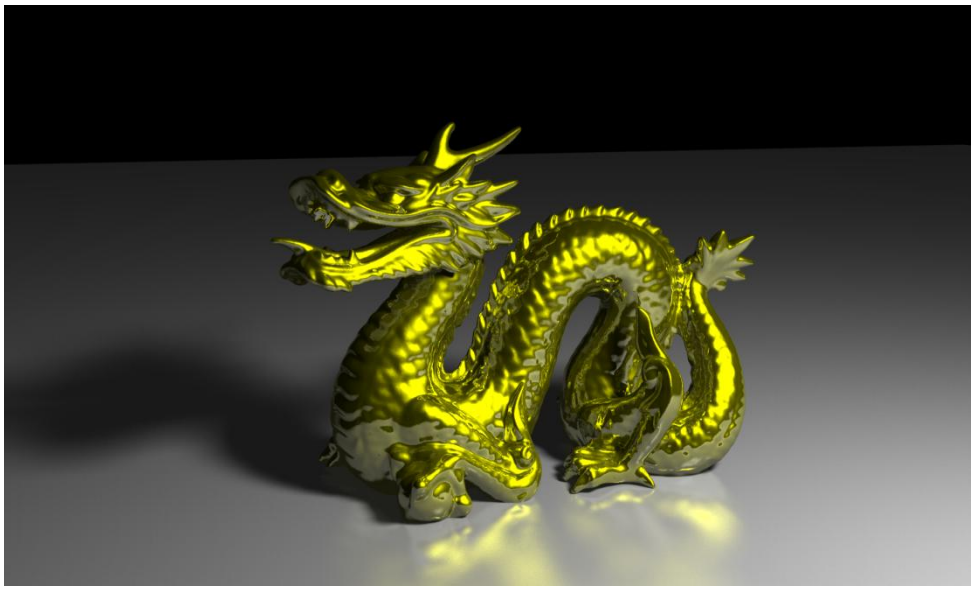

Figure 12. 1920x1200. GTX260; 144 sec.

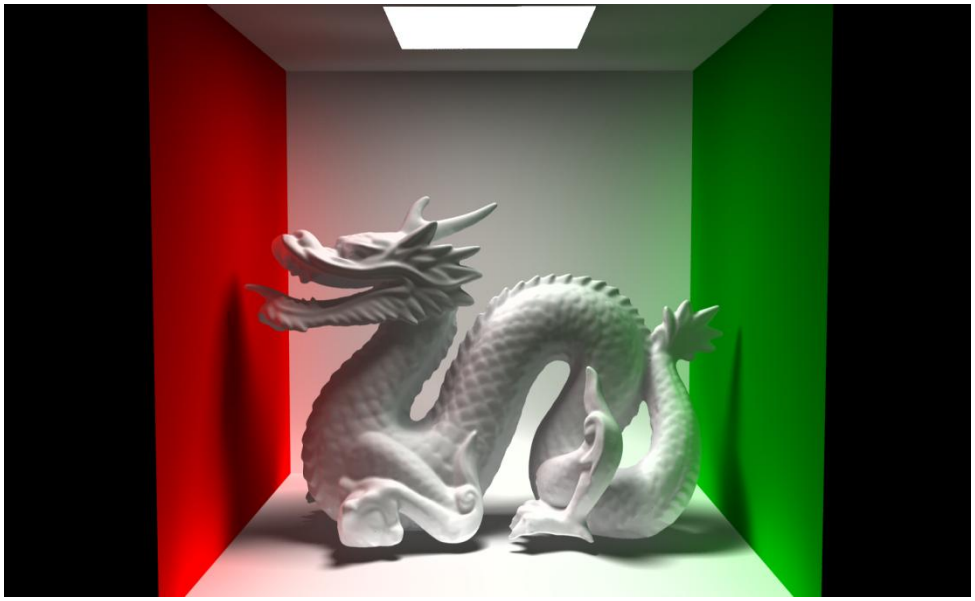

Figure 13. 1920x1200. GTX260; 181 sec

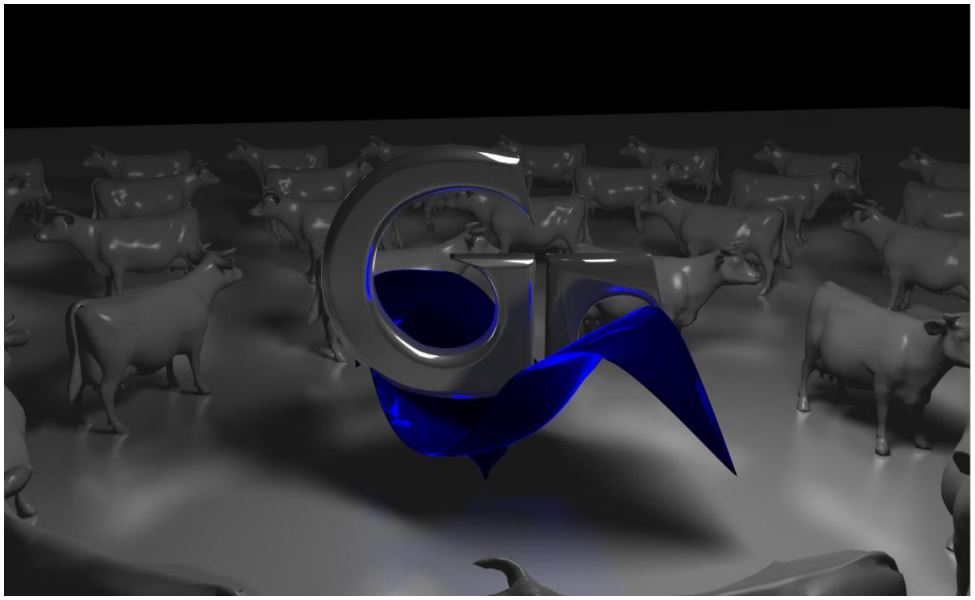

Figure 14. 1920x1200. GTX260; 159 sec

7. LITERATURE

[1] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2005. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses* (Los Angeles, California, July 31 - August 04, 2005). J. Fujii, Ed. SIGGRAPH '05. ACM, New York, NY, 268. DOI= http://doi.acm.org/10.1145/1198555.1198798

[2] Foley, T. and Sugerman, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Los Angeles, California, July 30 - 31, 2005). HWWS '05. ACM, New York, NY, 15-22. DOI= http://doi.acm.org/10.1145/1071866.1071869

[3] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. 2007. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 Symposium on interactive 3D Graphics and Games* (Seattle, Washington, April 30 - May 02, 2007). I3D '07. ACM, New York, NY, 167-174. DOI= http://doi.acm.org/10.1145/1230100.1230129

[4] THRANE N., SIMONSEN L. O.: A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis, University of Aarhus, 2005. 2, 3

[5] Wang R., Zhou K., Pan, M., and Bao, H. 2009. *An efficient GPU-based approach for interactive global illumination. ACM Trans. Graph.* 28, 3 (Jul. 2009), 1-8.

[6] Fabianovski B., Dingliana J. *Interactive Global Photon Mapping*. . In Proceedings of the EUROGRAPHICS conference, 2009. p. 1151-1159.

[7] Frolov V., Ignatenko A. *Interactive GPU Ray Tracing and Photon Mapping.* In: GraphiCon'2009.; 2009. p. 255-262. (In Russian).

[8] Jensen, H. W., Suykens F., Christensen Per H., Kato T. A *Practical Guide to Global Illumination using Photon Mapping*. SIGGRAPH 2002 Course Note #43. ACM, July 2002. (San Antonio, USA, July 21-26).

[9] McGuire, M. and Luebke, D. 2009. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana, August 01 - 03, 2009). S. N. Spencer, D. McAllister, M. Pharr, and I. Wald,Eds. HPG '09. ACM, New York, NY, 77-89. DOI= http://doi.acm.org/10.1145/1572769.1572783

[10] Aila, T. and Laine, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana, August 01 - 03, 2009). S. N.

[11] Křivánek, J., Gautron, P., Ward, G., Jensen, H. W., Christensen, P. H., and Tabellion, E. 2008. *Practical global illumination with irradiance caching*. In ACM SIGGRAPH 2008 Classes (Los Angeles, California, August 11 - 15, 2008). SIGGRAPH '08. ACM, New York, NY, 1-20. DOI= http://doi.acm.org/10.1145/1401132.1401213

[12] Pharr, M. and Humphreys, G. 2004 *Physically Based Rendering: from Theory to Implementation*. Morgan Kaufmann Publishers Inc.

[13] Garanzha K ., Loop C. *Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing*. In Proceedings of the EUROGRAPHICS conference, vol. 29 (2010), Number 2.

8. ABOUT THE AUTHORS

Vladimir Frolov graduated with MS degree from Computational Mathematics and Cybernetics department of Lomonosov Moscow State University. Vladimir works at NVIDIA as an intern. His area of research is photorealistic rendering, fluid simulation and gpu programming. His contact email is vfrolov@graphics.cs.msu.ru.

Alexander Kharlamov is a PhD student at Computational Mathematics and Cybernetics department of Lomonosov Moscow State University. Alexander works at NVIDIA as a developer technology engineer. His area of research is photorealistic rendering, physical simulation and gpu programming. His contact email is alharlamov@nvidia.com.

Alexey Ignatenko is a PhD researcher at Computational and Cybernetics department of Moscow State University. His contact e-mail is ignatenko@graphics.cs.msu.ru.