# Iterative Layer-Based Raytracing on CUDA

Alejandro Segovia, Xiaoming Li, Guang Gao
University of Delaware
Electrical And Computer Engineering
DuPont Hall, Newark, DE
{segovia, xli, ggao}@udel.edu

## Abstract

*A raytracer consists in an application capable of tracing rays from a point into a scene in order to determine the closest sphere intersection along the ray's direction.*

*Because of their recursive nature, raytracing algorithms are hard to implement on architectures which do not support recursion, such as the NVIDIA CUDA architecture. Even if the recursive portion of a typical raytracing algorithm can be rewritten iteratively, naively doing so could tamper with the image generation process, preventing the parallel algorithm's results from maintaining high fidelity to the sequential algorithm's and resulting, in many cases, in lower quality images.*

*In this paper we address both issues by presenting a novel approach for expressing the recursive structure of raytracer algorithms iteratively, while still maintaining high fidelity to the images generated by the sequential algorithm, and leveraging the processing power of the GPU for parallelizing the image generation process.*

*Our work focuses on designing and implementing a raytracer that renders arbitrary scenes and the reflections among the objects contained in it. However, it can be easily extended to implement other natural phenomena, such as light refraction, and to aid the iterative implementation of recursive algorithms in architectures like CUDA, which do not support recursive function calls.*

## 1. Introduction

Raytracing is a technique widely used in several computer generated imagery (CGI) applications. At its most basic level, the algorithm consists in casting (or tracing) rays from a given point in 3D space through a viewing window, called the viewport, into space, trying to find intersections with a given set of objects called *the scene*.

Because of their mathematical simplicity, objects commonly used are spheres, whereas rays are characterized by a base point and a direction. Other objects usually employed include polygons or, more specifically, triangles.

While raytracing algorithms allow magnificent visual representations of scenes and natural phenomena using straightforward and simple algorithms, they are far more compute-intensive than a rasterization process[1].

The main difference between rasterization and raytracing consists in that they work in opposite ways[1]. During a rasterization process, geometric primitives in the scene are transformed and *projected* onto a region of a 2D plane. Lighting, texturing and scaling are applied at different stages of the process, allowing the final image rendered to be correctly textured and shaded as seen from the camera.

During a raytracing process, on the other hand, rays are cast *from* the camera into the scene for every pixel in the image to render[1]. If an intersection of a ray with an object is found, color computations are performed at the intersection point. The calculated color will eventually be the one painted at the pixel that corresponds to the ray cast.

It is the great number of computations that have to be performed during a raytracing process that make it far more compute-intensive than a rasterization process.

One important characteristic of raytracing algorithms is that the final color of every pixel in the image to be generated can be calculated independently of the others, providing a very natural way to parallelize a raytracer's implementation by dividing pixel color calculations among threads.

Raytracing is known to be an *embarrassingly parallel* problem, and thus raytracing algorithms provide a good opportunity to explore new hardware architectures in pursuit of interactivity [2].

Furthermore, because of their very nature, raytracing algorithms are inherently recursive and pose an interesting design and implementation problem on many-core architectures that do not support recursion, such as CUDA [3].

The problem to be addressed by this work consists in how to refactor a recursive raytracing algorithm into an iterative algorithm, suitable for efficient execution on the CUDA architecture, and which does not alter the correct combination of colors for the generated image, thus maintaining high fidelity to the original algorithm's results.

The following section details the problems addressed by our work. Section 3 introduces a basic sequential raytracing algorithm and analyzes a parallelization approach. Sections 4 and 5 present our novel stack-based approach for expressing recursion iteratively as well as our *layered* approach to raytracing and describes its implementation on CUDA. Section 6 describes our synthetic benchmark and presents the results obtained. Finally, section 7 presents the conclusions of our work.

## 2. Problem Formulation

When targeting the CUDA architecture, a raytracing algorithm can be implemented as a single function that runs on the GPU device, which is usually referred to as a *kernel* in CUDA's nomenclature. The problem is how to express the raytracer algorithm's recursion on CUDA, which does not support recursive function calls for its kernel implementations.

Under architectures such as CUDA, recursion must be replaced by another approach that will generate the same effect. One possible approach would be to transform the recursive algorithm into an iterative one, however, naively rewriting the recursive portion of a general raytracing algorithm leads to two difficulties.

First, the way colors are accumulated while raytracing natural phenomena, such as object reflections or light refraction, is important and cannot be altered. Typically, for every intersection found during the raytracing process, a new ray is cast from that point into a new direction as to determine if other object intersections are found along that ray and, if they are, what color would that other object contribute to the final color seen by the camera.

This process could be repeated an arbitrary number of times, and once a satisfactory number of rays have been traced, the colors would start to be accumulated in the *opposite* order in which they were calculated, yielding the final image.

Under the classical raytracing algorithm, calculated colors would have been temporarily stored in the function call stack and then combined together using a convex combination of colors.

Because of the nature of recursion, this convex combination of colors would have been evaluated as return values are popped from the function call stack, with the first intersected sphere's color (i.e. the one which the camera is actually seeing) being the last to be combined.

Clearly this order is important and must be preserved, otherwise, the last reflected color will be more *noticeable* on the final image than the diffuse color of the sphere itself, and the produced images will not appear to be similar to the original algorithm's.

Secondly, CUDA's parallelism level depends heavily on the number of available registers, which are needed in order to execute the threads contained in thread blocks. Having a kernel use too many registers will tamper with the device's parallelism level, and in time, with the application's performance.

On current hardware, a kernel should not use more than a few registers (10 or 16, depending on the video card model) in order to achieve an *occupancy* of 100%[3]. Although raytracing algorithms are simple to implement, they might require a large number of registers for executing, especially when performing some of the complex memory management operations required to improve performance.

## 3. Raytracing Algorithms

The pseudocode depicted as Algorithm 1 corresponds to a typical raytracer's main algorithm. This algorithm is executed sequentially on each pixel in order to calculate its final color.

In this example, the function *calculateLighting* encapsulates all color calculations, in particular, shading and shadows.

**Algorithm 1** Recursive raytracing

```
intersection, sphere = closestIntersection(ray, scene)
if $intersection == null || bounces \leq -1$ then
  return BLACK
end if
color = calculateLighting(intersection, sphere, lights)
ray' = reflect(ray, intersection, sphere)
a = sphere.alpha
return (1 - a) * color + a * raytrace(ray', bounces - 1)
```

In a sequential implementation of the raytracer, Algorithm 1 would be implemented as a function to be executed on the CPU, calculating the color of every pixel in the image, one at a time.

In a parallel implementation, however, advantage can be taken of the fact that pixel color calculations are independent from one another and thus threads that process groups of pixels in parallel could be spawned.

On some architectures, it might even prove beneficial to spawn exactly one thread per pixel. This is the case for CUDA, where creating a thread for every pixel will enable our implementation to transparently scale across several generations of devices as more and more processing cores continue to be added to CUDA-enabled GPUs[3].

Care must be taken when parallelizing Algorithm 1, however. Algorithm 2 presents a first parallel version of this raytracer algorithm.

This algorithm executes in parallel, calculating the color to paint every pixel of the image being rendered. Since CUDA does not allow recursive function calls, the recursion has been naively refactored into a loop.

**Algorithm 2** Initial version of a parallel raytracer

```
pixel = BLACK
for $i = 0$ to maxbounces do
    intersection, sphere = closestIntersection(ray, scene)
    if $intersection == null$ then
        return pixel
    end if
    color = calculateLighting(intersection, sphere, lights)
    a = sphere.alpha
    pixel = (1 - a) * color + a * pixel
    ray = reflect(ray, intersection, sphere)
end for
return pixel
```

In this algorithm each pixel is assigned an initial color and, as rays are cast, hit and bounce off objects, colors are accumulated incrementally. The problem with this algorithm is that the order in which colors are combined is the exact opposite of the order used in Algorithm 1, thus yielding very different results.

This problem arises from having naively rewritten the recursion as a loop, accumulating colors as ray-sphere intersections are found. It could be avoided, however, if colors where temporarily stored in an abstract data type that would allow the program to have them accumulated in the opposite order, once all colors had been calculated. An ideal abstract data type to this means would be a stack.

The next section presents a new parallel algorithm that builds on these concepts but generates images more efficiently which are also true to the images produced by the original raytracing algorithm.

## 4. Independent Layers Approach

Our proposed solution consists in changing the basic structure of the raytracing algorithm in order to address the aforementioned problems.

We start by dividing the raytracing algorithm into a set of individual cooperative kernels. Each one of these kernels executes one after the other on the GPU (i.e. the *device*). Within each kernel one thread will be spawn for each pixel of the image to be rendered.

Results from a kernel's execution will remain in the device's global memory, where the next kernel launched will be able to access them.

The part of our algorithm that will execute on the CPU (i.e. the main program running on the *host*) will be responsible for launching the kernels in sequence, but will be able to access their intermediate results from global memory.

This will allow our raytracing algorithm to leverage both the programmability of the host and the high parallelism of the device, in order to implement complex effects that could not be easily expressed if the solution was implemented as a single kernel.

Furthermore, this model will allow the host to launch the kernels several times iteratively, each time with a different set of parameters, and then store temporary results for post-processing, in order to implement visual effects and natural phenomena such as reflections and refraction, which are typically expressed through recursion.

This concept is significantly different from previous work on raytracer implementations on CUDA, such as the one presented in [5]. In [5] the author does leverage both the CPU and GPU, but uses the CPU just to parse the scene configuration files and to handle events from the user. The GPU must execute the (single) kernel that renders the whole scene, without assistance from host code.

In our algorithm, since control over the image generation process is retained by the host, the main program can use the kernels as building blocks to generate different renders of the scene iteratively, keeping references to all the rendered images, and then having them combined in a final stage.

Kernels should be launched with the appropriate set of parameters in every loop depending on the visual effects being rendered. Every rendered image will remain in the device's global memory and references to them will be maintained in a stack structure in the host's main memory.

Once all images have been rendered, the stack will be unwinded and images will be combined together in the correct order as if they were *layers* of the same picture, yielding the final image.

Not only does this approach allow the iterative implementation of natural phenomena while still producing images true to the original recursive algorithm, but also addresses the problem of register pressure by separating the algorithm into independent cooperative kernels.

The idea of dividing raytracing logic among several kernels has been presented before in [4]. Their objective was to divide the logic as a mechanism to avoid branching and looping, which were constructs not available on graphics hardware at the time, as well as to maximize parallelism. Our objective, on the other hand, is to add flexibility to our algorithm and to reduce register pressure.

A new parallelized raytracing pseudocode is presented as Algorithm 3. It leverages our layered concept in order to calculate reflections.

The main loop is executed by the CPU whereas the kernels (denoted by the *Krnl* suffix) execute one at time on the device, performing their computations in parallel.

It is important to note that since a stack is a LIFO structure, layers are combined in the same order their colors would have been combined by Algorithm 1, had they been temporarily stored in a function call stack.

**Algorithm 3** Iterative, layer-based raytracing (host code)

```
for $i = 0$ to maxbounces do
    intersections = intersectKrnl(bases, directions, spheres)
    shadowmap = shadowsKrnl(intersections, spheres, lights)
    layer = lightingKrnl(intersections, spheres, lights, shadowmap)
    directions = reflectKrnl(intersections, directions, sphere)
    bases = intersections
    push(layer)
end for
image = new()
repeat
    layer = pop()
    image = lerpKrnl(layer, image)
until stackempty()
return image
```

In this version of the algorithm the host iterates *maxbounces* times, generating successive reflected images at each iteration.

For the first iteration, rays are cast in parallel from the viewer's position, through the viewport, into the scene. The resulting layer will correspond to the objects directly seen by the camera, correctly shaded, but without reflections. For the second iteration, rays will be cast (again in parallel) from the intersection points found during the previous iteration, and the direction will be the previous ray's, reflected on the sphere's surface.

The relationship between two consecutively generated images $a$ and $b$ is that $b$ corresponds to the image resulting from raytracing the scene with the rays being cast from the intersections points found when rendering image $a$ into the direction light would take when being reflected on the intersected sphere (at the intersection point).

Since a new ray will be traced for every ray previously cast, there will be a one-to-one relationship between the number of rays to be traced and the number of threads executing our algorithm.

The only exception for this rule will be for those rays that did not intersect any objects in the previous iteration, however, we can safely assume that scenes usually raytraced are dense in objects and thus it is highly unlikely for rays not to intersect any objects in the scene.

Given this assumption, we can conclude that, for every iteration, the computations to be performed will be correctly balanced among threads, allowing our algorithm to achieve a very high degree of parallelism when raytracing all but the simplest scenes.

Algorithm 3 relates to the work presented in [6] in the sense that we are using the GPU to perform the tasks it is best at: performing computations in parallel, while we are using the CPU to perform those tasks the GPU is worst at: orchestrating kernel calls and creating, maintaining and unwinding a dynamic stack for finally composing the layers into the rendered image.

## 5. Raytracer Implementation

We will implement an iterative layer-based parallel raytracer on CUDA in order to evaluate our new raytracing algorithm.

Another interesting visual effect that could be easily implemented by applying this approach is light refraction, triggered by a ray that impacts a translucent object.

### 5.1 Algorithm Implementation

The parallel implementation of the algorithm proposed consists in spawning one thread on the GPU for each pixel, allowing color calculations to be performed in parallel on the video hardware.

The NVIDIA CUDA architecture eases the implementation process by allowing the developer to define and implement functions that execute directly on the GPU using a set of C-language extensions.

Raytracing primitives, such as calculating intersections, calculating shadows and calculating lighting were all implemented as individual cooperative kernels.

Reflections in our raytracer were implemented using our layered concept. In order to render reflections, the host iterates, configuring kernel launches as to generate images corresponding to different ray bounces in every loop. Every rendered image is stored in the device's global memory and a corresponding stack of pointers is maintained in host memory.

Once all the images have been rendered, the host starts to unwind the stack, combining all the rendered images with one another, in order to produce the final image.

Images are combined using a convex combination of colors. The function that applies this operation is implemented as a kernel as well, so every pair of images is combined in parallel, on the GPU.

Once stack unwinding is completed, the final image is copied from device memory into host memory and rendered on the screen. Figure 1 visually depicts this concept.

In this figure, the first two images are generated by calling the same set of kernels with different raytracing parameters.

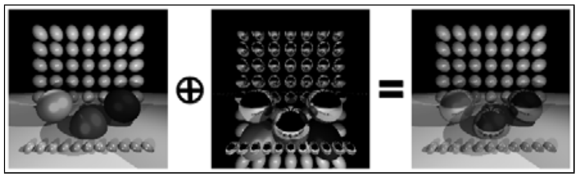

Figure 1. Layer-based raytracing example.

In the first image, all rays are cast from the camera's position, whereas their direction is into the scene. In the second image, rays are cast from all the points intersected at the first iteration, and their direction corresponds to the direction light would be reflected into when colliding with a curve mirror (the sphere). The third image shows the combination of both images.

For this sample figure, the application was configured as to bounce rays just twice, however, since the image rendering process is completely dynamic, the user may configure the application to bounce rays any number of times at runtime, with the only limitation being the device's available global memory.

Should a third level of reflection have been requested, the set of kernels would have been launched once again, this time with the ray bases set at the intersections found for the last image and the directions set to a new ray bounce on the spheres. All three images would have been composed in the end.

### 5.2 Memory Management

Spheres in our raytracer were arranged in a linear data structure. Using a linear data structure provides several benefits when compared to other alternatives.

In the first place, since the set of spheres to be processed by all threads is the same (independently the direction rays are cast into) branching is reduced dramatically, helping to further improve each kernel's performance[7][3].

In the second place, using a linear data structure helps simplify memory management logic, thus decreasing register pressure and increasing every kernel's occupancy on the device[3].

In order to find the best possible memory layout to resolve the problem of determining the closest intersection of a ray with the set of spheres, three different approaches were proposed and implemented.

The first approach consists in placing the list of spheres in global memory and having the function that calculates intersections process them directly. This approach will be referred to as the *naive* approach.

The second approach proposed consists in copying a subset of the spheres from global memory into the device's shared memory and having only that subset processed by the function that calculates intersections.

Once all the spheres in this subset have been processed, a new subset is loaded and processed.

This process is repeated until all the spheres have been used. Since shared memory is a scarce resource, restricting the subset size provides good scalability in terms of the number of spheres in the scene, which can be in the order of hundreds or even thousands.

This approach will be referred to as the *incremental* approach, since spheres are processed incrementally in fixed size batches.

The third approach proposed consists in having a subset of the spheres processed from shared memory and its complement processed directly from global memory. This approach will be referred to as the *hybrid* approach.

## 6. Synthetic Benchmark

In order to prove our proposed solution, a synthetic benchmark was designed and implemented.

The benchmark compares our new parallel raytracer algorithm against a reference sequential algorithm. The objective of the comparison was to measure the speedup obtained from our layer-based parallel implementation on CUDA, while still producing images of high fidelity to the sequential implementation.

The tests consist in rendering into a 512x512 pixel canvas a predefined scene of 51 spheres, tracing reflections among its objects 3 times.

The scene was to be rendered 30 times by each implementation under different hardware and software configurations. For each execution, the raytracing algorithm would be measured, averaged and the corresponding speedups would be calculated.

At the software level, configuration of the reference sequential implementation of the raytracing algorithm was unique, whereas the parallel implementation was to be benchmarked under three different configurations, one for each memory model presented in the previous section: naive, incremental and hybrid.

The CUDA version employed for the parallel raytracer was 2.2 and measurements were performed by means of the *CUDA Event* facilities. CUDA Events provide an exact kernel timing mechanism that can be used to measure the total time a function was executing on the GPU[3].

At the hardware level, five different configurations were benchmarked: two workstation configurations and three mobile configurations.

The following subsections group the results obtained by the different hardware configurations and, then, by the different software configurations. The speedup values for our parallel raytracer implementation were calculated by comparing against the reference sequential implementation. The reference implementation was assigned a baseline speedup of *1*.

## 6.1 Workstation Benchmarks

The following two benchmarks were executed on a workstation configuration consisting of a 2.0GHz Intel Pentium D processor, with 2GB of RAM and two NVIDIA GeForce video cards: a GeForce 8800GTX and a GeForce 280.

The video card used for benchmarking was selected by software when invoking the parallel raytracer. The Operating System under which these benchmarks were performed was Ubuntu Linux 8.10.

Figure 2 corresponds to the results from executing the benchmark on the GeForce 8800GTX video card. The video display was connected to this video card during the benchmarking process.

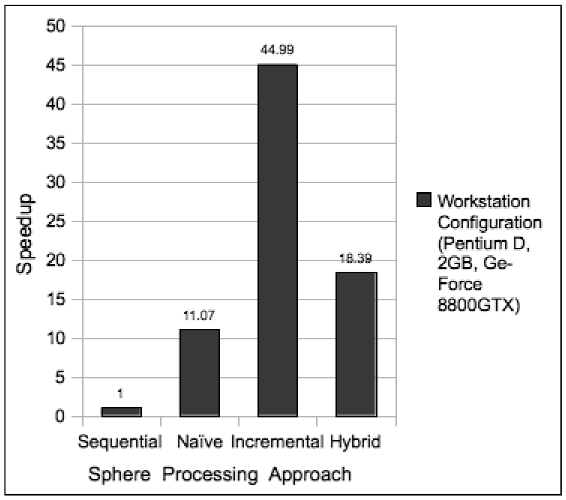

**Figure 2. Intel Pentium D 2.0GHz, 2GB RAM, NVIDIA GeForce 8800GTX.**

As Figure 2 shows, when comparing the parallel raytracer against the sequential implementation, a significant speedup is obtained for all three sphere memory layouts. The incremental approach yields the best results.

Figure 3 shows the results of executing the benchmark having switched to the NVIDIA GeForce 280 video card. This video card was not connected to the video display during the benchmarking process.

Under this benchmark, the naive approach proved to achieve a higher speedup than the incremental and hybrid approaches, even if it does not take advantage of the device's shared memory.

In all three cases, the speedup obtained when comparing against the reference implementation was very significant,

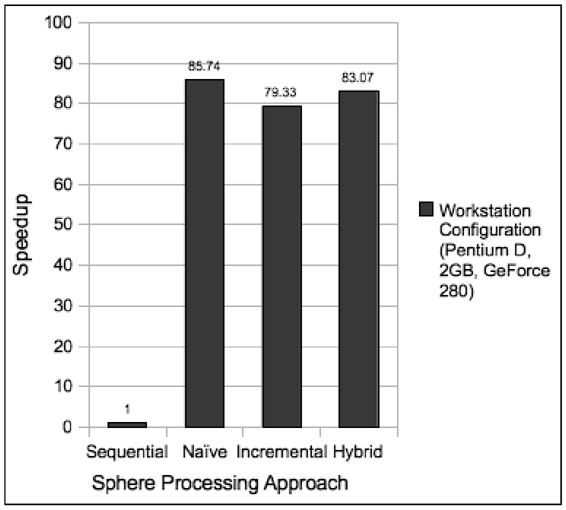

**Figure 3. Intel Pentium D 2.0GHz, 2GB RAM, NVIDIA GeForce 280.**

with the naive approach averaging an speedup of more than 85X.

## 6.2 Mobile Benchmarks

The following three benchmarks were executed under the mobile configuration, consisting of a 2.66GHz Intel Core 2 Duo processor with 4GB of RAM and two NVIDIA GeForce video cards: a GeForce 9400M and a GeForce 9600GT.

The video card used for the benchmark was selected by software when invoking the parallel raytracer. The Operating System under which these benchmarks were performed was Mac OS X 10.5.8.

Figure 4 depicts the results obtained from executing the benchmark using the NVIDIA GeForce 9400M. This video card was connected to the video display during the benchmarking process.

Although results obtained were not as impressive as the ones obtained for the workstation configuration, it is common for mobile video cards to provide less processing power than for workstation or desktop videos cards.

Once again, in all of its three configurations, the parallel implementation of the raytracer algorithm provided a significant speedup when compared against the sequential implementation. The incremental approach was the best sphere processing solution, yielding an speedup of more than 3X.

Figure 5 presents the results for the benchmark using the NVIDIA GeForce 9400M video card while not connected to the video display.

Even if the NVIDIA GeForce 9400M video card was not connected to the video display, the results obtained were

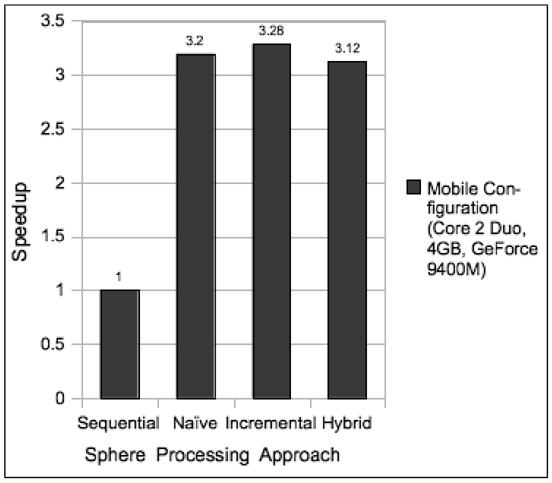

**Figure 4. Intel Core 2 Duo 2.66GHz, 4GB RAM, NVIDIA GeForce 9400M.**

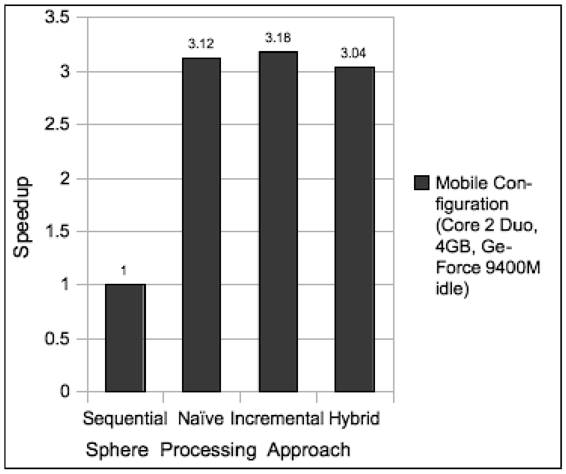

**Figure 5. Intel Core 2 Duo 2.66GHz, 4GB RAM, NVIDIA GeForce 9400M in an idle state.**

very similar to the ones previously obtained.

Finally, Figure 6 presents the benchmark results for the mobile configuration using the NVIDIA GeForce 9600GT video card. This video card was connected to the video display during the benchmarking process.

Results for this benchmark were mixed, with the naive and hybrid approaches not being able to match the results obtained for the 9400M video card. The incremental approach, however, exceeded the results previously obtained, yielding a speedup of more than 6X.

In all cases and for all hardware and software configurations, our parallel raytracer implementation proved to be significantly faster than the reference sequential implementation, with speedups obtained ranging from a minimum of 3X, for a mobile configuration, to a maximum of more than 85X, for a workstation configuration.

## 7. Conclusion

In our work we studied different raytracer parallelization approaches and proposed a new efficient and scalable algorithm that leverages a host managed, device-based stack as a mechanism for expressing the recursive nature of raytracing iteratively.

Through our synthetic benchmark we were able to verify that our algorithm yields very good results that maintain high fidelity to the images produced by the original raytracing algorithm while, at the same time, provide a significant speedup.

Furthermore, we were able to demonstrate that our algorithm is very scalable in terms of GPU hardware, being able to leverage low-end mobile video cards as well as high-end workstation GPUs by dividing our problem in a way that could be distributed to all the streaming processors available automatically.

We believe our approach for expressing recursion iteratively, with the assistance of the host managed, device-based stack, could be leveraged and extended well beyond the field of raytracing as a more general mechanism for refactoring recursive algorithms into new iterative forms where strong data dependency restrictions apply.

In terms of memory layout, we were able to identity our incremental sphere processing approach as the best solution for processing the scene spheres when these are stored in a linear data structure such as a list.

It would be very interesting to implement other memory layouts (such as kd-trees) and measure the impact on application performance from expressing the logic needed to traverse them iteratively while, at the same time, trying to leverage all the streaming processors available on the GPU.

Our incremental approach to sphere processing, on the other hand, is simple enough while, at the same time, it allows the raytracing algorithm implementation to excel at executing in parallel on the device, ranging from a minimum speedup of 3X on mobile hardware configurations to a dramatical 79X speedup on high-end hardware configurations, allowing reasonably interactive raytracing to be performed on commodity hardware.

## 8. Acknowledgement

Alejandro Segovia would like to thank Jakob Siegel for his patience and Juergen Ributzka for his encouragement. He would also like to thank Gabriel F. Gambetta and, specially, María Soledad Villar, without whose unconditional support, this work would not have been possible.

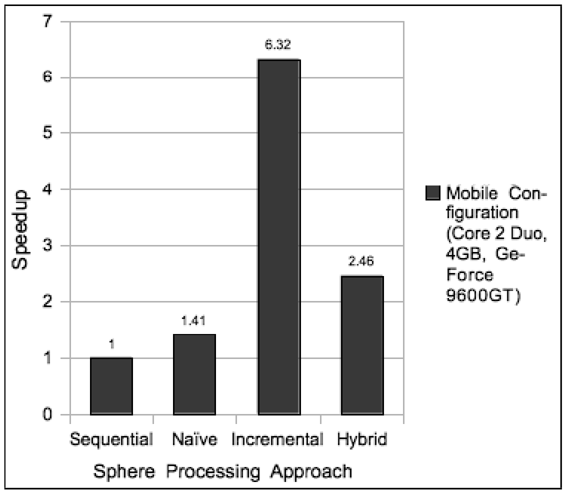

**Figure 6. Intel Core 2 Duo 2.66GHz, 4GB RAM, NVIDIA GeForce 9600GT.**

## References

[1] van der Ploeg, A.J. 2008. Interactive Ray Tracing.

[2] 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

[3] C. NVIDIA. *Compute Unified Device Architecture Programming Guide*. NVIDIA: Santa Clara, CA, 2009.

[4] 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

[5] Allgyer, M. 2008. Real-time Ray Tracing using CUDA. Master's Project Report.

[6] Carr, N. A., Hall, J. D., and Hart, J. C. 2002. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Saarbrucken, Germany, September 01 - 02, 2002). SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware. Eurographics Association, Aire-la-Ville, Switzerland, 37-46.

[7] Carrillo, S., Siegel, J. and Li, X. 2009. A control-structure splitting optimization for GPGPU. In *Proceedings of the 6th ACM conference on Computing frontiers*. ACM New York, NY, USA, 147-150.