# Performance Analysis for GPU-based Ray-triangle Algorithms

Juan J. Jiménez, Carlos J. Ogáyar, José M. Noguera and Félix Paulano
*Grupo de Gráficos y Geomática de Jaén, University of Jaén, Campus Las Lagunillas, Edificio A3, 23071, Jaén, Spain*

Keywords: Ray-triangle Algorithms, Parallel Processing, Raytracing, Geometric Algorithms, Graphics Processors.

Abstract: Several algorithms have been proposed during the past years to solve the ray-triangle intersection test. In this paper we collect the most prominent solutions and describe how to parallelize them on modern programmable graphics processing units (GPUs) by means of NVIDIA CUDA. This paper also provides a comprehensive performance analysis based on several optional features and optimizations (such as back-face culling and the use of pre-computed values) that allowed us to determine the influence of each factor on the performance. Finally, we analyze the architecture of the GPU and its impact on the parallel implementation of each method, as well as the approach used to achieve a high-performance fine-grained parallel computation on the ray-triangle test.

## 1 INTRODUCTION

Triangles are one of the most used primitives in Computer Graphics. Due to their simplicity, triangles are used as the basic geometric elements for high-performance applications, including real-time rendering. In consequence, the triangle mesh is the most common structure used nowadays for representing complex objects, and therefore most existing triangle-based geometric algorithms can be applied on triangle meshes in a natural way. This also applies to most ray-triangle intersection test algorithms proposed in the literature.

Ray-triangle and segment-triangle intersection tests are basic algorithms used for solving many problems in Computer Graphics. This includes applications such as ray tracing, ray casting, inclusion tests, boolean operations, object modeling, physics simulation, collision detection, etc. Therefore, guaranteeing the performance, robustness and accuracy of the intersection test algorithms is paramount in the development of the aforementioned applications.

On the other hand, parallel computing is one of the best ways to improve the efficiency of a broad class of geometric algorithms. For modern CPUs, this is achieved by using all the available cores and their SIMD architecture (Benthin, 2006), (Shevtsov et al., 2007), (Havel and Herout, 2010). Also, current GPUs can be used as a general purpose multiprocessor. This has motivated the adaptation of many Computer Graphics algorithms to the GPU, resulting in implementations that typically outperform their CPU-based counterparts. Examples include linear algebra, image processing, collision detection (Lin and Gottschalk, 1998), global illumination, etc. It is important to remark that not all algorithms can be efficiently implemented on the GPU. Nevertheless, the recent improvements of these architectures (memory, execution control, thread management, memory bandwidth, etc.) are widely expanding their field of application (Kim et al., 2007), (Rueda and Ortega, 2008).

One of the applications that can undoubtedly benefit from these GPU improvements is the ray-triangle intersection test. The parallelization of this test would in turn result in an important improvement of the ray tracing rendering technique (Glassner, 1989), (Amanatides and Choi, 1995), (Purcell et al., 2002), (Foley and Sugerman, 2005), (Aila and Laine, 2009) as well as in all the other Computer Graphics applications mentioned above.

In this paper we present a comprehensive study of ray-triangle intersection algorithms for high performance applications based on modern programmable graphics hardware. The algorithms included in this study have been implemented in NVIDIA CUDA, which provides a high level framework for implementing general purpose algorithms on today's GPUs. We also study the performance of each algorithm based on several factors that define a common criteria for evaluating the best strategy in each case.

## 2 RAY-TRIANGLE ALGORITHMS

Fast ray-triangle intersection algorithms have been an active field of research in Computer Graphics for a long time. Several algorithms have been developed in order to solve this problem in an efficient and robust manner, see (Plucker, 1865), (Badouel, 1990), (Möller and Trumbore, 1997), (Segura and Feito, 1998), (Segura and Feito, 2001), (Jiménez et al., 2010). Albeit all these algorithms can be efficiently implemented on the GPU, the peculiarities of the GPU architecture make some of them more adaptable than others. In this section we present an overview of the ray-triangle intersection algorithms considered in this study. A GPU-based parallel implementation of these algorithms will be later provided in Section 3.

Note that in this paper we are mostly interested on the aspects related to the adaptation of these methods to the parallel hardware. Therefore we will omit the full description of the algorithms, referring to the original publications instead.

Before beginning our study, some basic notations about ray representation should be introduced. In what follows we will use the parametric equation of the ray. A ray $R(t)$ with origin $O$ and normalized direction $D$ is defined as (see Figure 1):

$$R(t) = O + tD$$

The value $t$ is defined between 0 and infinity. A similar notation can be used when working with segments instead of rays. Let $Q_1$ and $Q_2$ be the two points that define the segment. Then, the segment can be expressed as follows:

$$S(t) = Q_1 + t(Q_2 - Q_1)$$

With $t$ in $[0,1]$. This allows the use of the same algorithms for both, rays and segments, with minor changes. That is, most of ray-triangle algorithms can be adapted to segment-triangle algorithms in a straightforward manner, and vice versa. Therefore and without loss of generality, we will refer only to the ray-triangle case since the segment-triangle intersection test can be considered a specific case of the first one.

Besides the ray, the triangle to be intersected is defined as a list of three vertices $(V_1, V_2, V_3)$. Additionally, the triangle normal can be either pre-calculated, which is a common practice for the storage of triangle meshes, or computed on the fly if the memory is limited.

The common result of a ray-triangle intersection algorithm is a boolean flag that indicates whether an intersection is found or not. If so, the value for $t$ is usually calculated, that is, the distance between the origin of the ray and the intersection point. The intersection point can be determined by using the ray equation in a straightforward manner. Also, the barycentric coordinates of the intersection point in the triangle surface can be calculated if further processing is required. A typical example includes the interpolation of vertex attributes such as colors, texture coordinates, normals, tangent vectors, etc. Nonetheless, not all applications require the determination of this intersection point. In these cases, the computation can be omitted to obtain a significant improvement in the performance of the intersection algorithm.

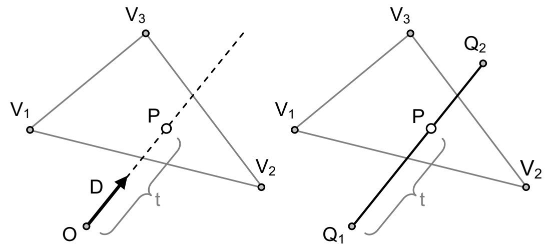

Figure 1: Left: sample configuration of a triangle and a ray intersecting at the point P. Right: a triangle and a segment intersecting at the point P.

In the literature there exists a number of solutions to the ray-triangle intersection test. Following we summarize the most prominent ones.

**Badouel's algorithm.** Badouel (Badouel, 1990) designed a solution based on determining the intersection between the ray and the plane of the triangle, the subsequent projection on a plane ($XY, YZ$ or $ZX$), and the computation of the intersection point on the triangle by using 2D barycentric coordinates.

**Möller's algorithm.** Möller (Möller and Trumbore, 1997) developed an algorithm based on the solution of an equation system formed by the ray equation and the equation of the intersection point between a ray and a triangle by using barycentric coordinates with regard to the triangle. This algorithm is an optimization of Badouel's method, and is widely considered as the fastest solution for the generic case.

**Segura's algorithm.** Segura (Segura and Feito, 1998) proposed a segment-triangle intersection algorithm which computes the sign of the volume of the tetrahedra formed by the triangle vertices and the endpoints of the segment. This method provides the boolean result of the intersection, but it does not calculate the intersection point. If this point is needed, it can be determined by means of a classic ray-plane intersection algorithm, as well as the value of $t$ (Segura and Feito, 2001).

**Jiménez 's algorithm.** This solution (Jiménez et al., 2010) is partially based on the same foundations than Segura's algorithm. Jiménez developed a segment-triangle method based on the calculation of the barycentric coordinates of the intersection point with respect to a tetrahedron formed by the triangle and one of the points of the segment. This algorithm allows the calculation of the intersection point and its barycentric coordinates in a straightforward fashion. The method can also be optimized by precalculating some values when all rays to be tested share the same origin.

**Other algorithms.** Other solutions include the use of Plücker coordinates (Plucker, 1865), (Jones, 2000), the Watertight ray-triangle intersection method proposed by Woop et al. (Woop et al., 2013), and optimizations for ray tracing like the solution adopted by Kensler et al. (Kensler and Shirley, 2006).

## 3 GPU-BASED IMPLEMENTATION

This section describes the design of the GPU-based implementation of the diverse algorithms described in the previous section. The implementation of these algorithms was based on a set of common criteria in order to highlight the impact that the characteristics of each method has on its GPU parallelization. In addition, we have studied several optimizations and their effect on the performance of each algorithm.

### 3.1 Common Basic Design

In our proposal, all the studied algorithms share the same structure. So in what follows we will present a common basic design for the implementation of a generic ray-triangle intersection test algorithms adapted to the CUDA parallel programming model.

In the first place, the ray-triangle algorithm itself is coded into a CUDA kernel function. The data required for this algorithm is copied from the host memory to the GPU global memory. This includes the vertices of the triangle and the ray to be tested for intersection. If the algorithm uses pre-computed data, they are also copied to global memory. Figure 2 shows the distribution of the kernels and the data into the CUDA architecture.

Since the ray-triangle algorithm is executed in parallel in multiple threads, the data set (vertices, rays and pre-computed values) is arranged in arrays. Then,

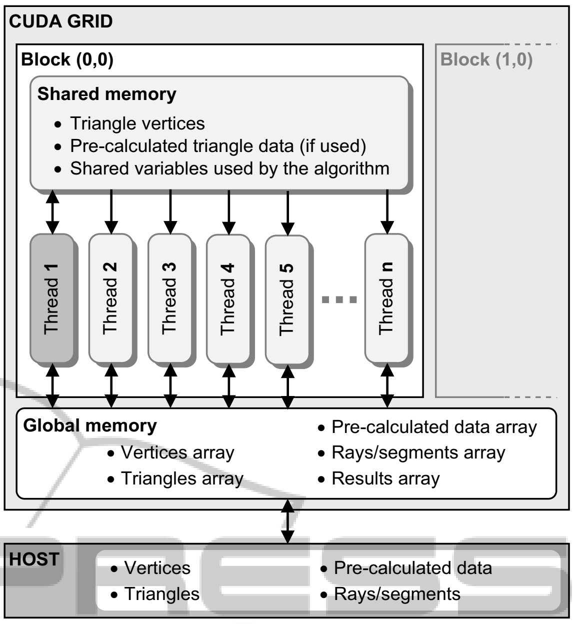

Figure 2: The integration of the kernels and the data into the CUDA architecture. Note that for the implementations proposed, only the first thread of each block writes to shared memory.

the kernel running in each thread uses its thread identifier for determining the position inside the arrays of the specific portion of data that it should load. Specifically, the data structure used to store the triangles is based on two indexed arrays. The first one stores the Cartesian coordinates of the vertices whereas the second one stores the indices to the vertices for each triangle. This structure allows the sharing of vertices between triangles, which is a common practice for storing triangle meshes, especially within the GPU.

The first approach that we implemented treated threads as isolated computing units. That is, no data was shared and every thread read the needed data directly from global memory. As expected, this naïve implementation produced poor performance because of the long-latency of the global memory access operations. This preliminary implementation was improved to make use the shared memory of each CUDA block. Since this kind of memory is built on-chip, memory accesses from the threads to the data stored in the shared memory can be performed at higher speed than to the global memory.

In the case of the ray-triangle intersection algorithms, there exist two kinds of data that are potentially candidates to be shared between the threads of a given block: the set of triangles and the set of rays. However, we should take into account that the amount of data required to describe a triangle (three points plus the pre-computed data) is greater than the amount of data needed to describe a ray or a segment (a point and a vector or two points, respectively). Moreover, the indexed array structure used to store shared vertices and triangles would force the algorithm to perform two memory accesses for retrieving a single vertex: one for obtaining the vertex index inside the triangles array, and another one for loading the actual Cartesian coordinates of the vertex inside the vertices arrays. For these reasons we consider that it is more efficient to share the data related to the triangles than the rays. As a result, in our solution the triangle data is shared among all the threads of a given grid, whilst each individual thread reads its own and unique set of rays.

Another important aspect is the granularity of the parallel execution, which is controlled by the number of rays that each kernel uses. For a fine-grained approach, each kernel execution only calculates the ray-triangle test with one ray, as shown in Figure 2. Note that the highly multithreaded architecture of the GPU encourages the use of massive, fine-grained data parallelism in CUDA (Kirk and Hwu, 2010). In our proposed implementation, a grid of $m \times n$ blocks is used for the CUDA execution. Each block handles a triangle, and the corresponding threads of the block calculate the ray-triangle test between that triangle and the set of rays. If there are more rays than threads per block, the kernel will be executed multiple times in order to test the entire set of rays.

The basic scheme of the proposed implementations works as follows.

1. The first step performs the initialization of the shared data. This includes loading the Cartesian coordinates of the vertices. Any existing pre-computed data is also loaded during this step. The shared data initialization is performed by the first thread of each block (Kirk and Hwu, 2010). The remaining threads of the block await until this step is finished, because they cannot continue until all the required data is available. This step ends with a barrier synchronization.
2. This step is the main part of the kernel. At this point, every thread can access the required data, as it is already cached on shared memory. Then, the actual ray-triangle intersection test is performed.
3. The last step consists of storing the results of the tests into global memory, so they can be eventually copied to host memory once the kernel finishes its execution (Kirk and Hwu, 2010). Figure 2 shows the data flow between the different memory levels. In order to increase the performance, only the result of positive tests is written into global memory. This allows the omission of a global memory access if the algorithm ends with a rejection. In order to obtain correct results, the array containing the results must be initialized with the intersection flag set to false before the kernel execution. Note that this whole step can be omitted if we intend to keep the results on GPU memory for a later use (e.g. real-time ray tracing).

As mentioned above, the data of the triangles are shared between all the threads, and only one group of access operations is performed for every block. Unfortunately, every individual thread must read the ray data directly from global memory, and this process cannot be simplified. Nevertheless, when all threads in a warp perform a load instruction, the GPU detects whether the threads access consecutive global memory locations. If this is the case, then the GPU combines or coalesces these accesses into a single access to a block of ordered data (Kirk and Hwu, 2010), which greatly increases the performance. Therefore, in our implementation the access to the array containing the rays has been designed to follow this scheme. If the granularity is greater than one, each thread performs the ray-triangle test for a subset of triangles using a loop. As a result, in order to allow the system to coalesce the accesses, the rays must be stored in global memory following an interleaved pattern. Thus, for a given iteration of the kernel, all the rays required by the threads are stored consecutively.

## 3.2 Algorithm-specific Aspects

The ray-triangle methods contempled in this study have a nearly direct translation to CUDA. The most common change was the use of the CUDA data types, especially vectors such as $float3$ or $int3$, and built-in mathematical functions. Some algorithms required additional changes to the original code, such as Badouel's method, because it uses variable indices for accessing the vertices coordinates depending on the projection used (to $XY, XZ$ or $YZ$ plane). The conversions of Möller's and Plücker's methods were obtained in a straightforward manner, because of their simplicity.

We would like to remark that we firstly implemented a unique kernel for each algorithm. These kernels were highly parametrizable because we were interested in studying their behavior after enabling or disabling different features, e.g., back-face culling, the use of pre-computed data, etc. Unfortunately we found out that the additional logic required to handle these parameters within the kernel (branching and conditional instructions) affected the performance, even when the execution path was the same for every thread. This performance penalty made it harder to evaluate the impact of each tested factor. As a result, we decided to implement different kernels in order to faithfully test the desired combination of features.

## 4 RESULTS AND DISCUSSION

In order to carry out a complete evaluation, all algorithms have been implemented in both the CPU and the GPU. The CPU-based implementations serve as a baseline reference for the study, and allow us to measure the performance implications of the characteristics of each algorithm. A more detailed analysis of CPU-based implementations of several ray-triangle algorithms is presented in (Jiménez et al., 2010), which follows the test methodology presented in (Löfstedt and Akenine-Möller, 2005).

The computer used for our experimentation was a PC equipped with an Intel Core 2 Quad Processor at 2.4GHz with 4GB RAM. The GPU used was a GeForce GTX560Ti with 1GB RAM. An increasing number of CPU cores were used for the CPU implementations with OpenMP, and the algorithms scaled as expected. The performance was measured in millions of ray-triangle intersections per second. All methods were tested with several triangle meshes featuring different complexities and topologies. The set of segments was randomly generated using the bounding box of each triangle mesh and a minimum length to ensure a better distribution of the intersection cases. The set of rays was generated in a similar manner, but without the length restriction. Single-precision floating-point numbers were used for all the implementations.

In the following subsections, we present the results of some of the most relevant tests carried out during our experimentation. We first report the performance of the basic implementation of each algorithm in the CPU and GPU. Next, we report the influence in the performance of different features for each evaluated algorithm. The tested features include back-face culling, segment intersections, the use of pre-computed data, and the cost of determining the intersection point on the triangle surface (the barycentric coordinates and the $t$ parameter). Also, we have tested an additional configuration consisting of a set of rays with a common origin. This experiment aimed at simulating a ray casting process. This wide set of experiments and configurations allowed us to draw several conclusions.

### 4.1 Performance

Table 1 shows the performance of a basic ray-triangle implementation of each method, that is, without the additional features mentioned above. For each 3D model, the Table shows the variation of the performance of the studied algorithms with an increasing number of CPU cores and the GPU. In any case the total number of intersections computed was below 50 million per second, because the peak performance of the GPU is achieved for a high number of tests. Also, the performance depended on the topology of each mesh and the spatial distribution of its triangles, although the differences were subtle.

Table 1 also shows the performance gain of each version when compared to the equivalent single-core CPU implementation. The performance gain of the methods running on the GPU mostly depended on their ratio of floating-point arithmetic instructions over conditional and flow control instructions. This result is consistent with the expected behavior of the GPU, which typically favors the former type of operations over the later. From Table 1 we observe that in general, the increment in the performance of the Badouel's algorithm was less significant than the other algorithms. This is likely caused by the operations required to project the triangles into the $XY$, $XZ$ or $YZ$ planes, which depend on the normal of each triangle. On the contrary, the most important increment in the performance was obtained by the algorithm based on the Plücker coordinates. This stems from the fact that it has few flow control instructions and focuses on mathematical operations instead.

We also noticed that in general, the CPU scalability is very predictable, although some methods needed more tests than others in order to achieve its maximum performance. Therefore, this result confirms our hypothesis that the single-core CPU tests were adequate as a baseline reference for the tests on the GPU.

It is also noteworthy to mention that several today's CPUs feature a set of SIMD instructions known as SSE. In theory, this set of instructions can increase the performance of the algorithms by a factor of 4. But despite this, we decided not to implement a SIMD version of the CPU-based algorithms because some of them do not translate well into the SSE programming paradigm. For further information, we refer to several works that show how to implement a ray-triangle algorithm using SSE (Shevtsov et al., 2007), (Havel and Herout, 2010), (Noguera et al., 2009).

Besides, the SPMD type of parallelism of the GPU makes it more efficient than the CPU using SSE extensions, which is an SIMD model. With SSE, all

Table 1: Experimental results of a simple ray-triangle test. Values are in millions of intersection tests per second. Secondary values are the performance gain factors with respect to the single-core CPU implementation.

| | | CPU x1 | CPU x2 | | CPU x4 | | GPU | |
|---|---|---|---|---|---|---|---|---|
| | | Mi/s | Mi/s | Gain | Mi/s | Gain | Mi/s | Gain |
| Bunny 69.451 triangles | Badouel | 10,86 | 21,54 | 1,98x | 30,78 | 2,83x | 1051,72 | 96,76x |
| | Möller | 20,04 | 39,58 | 1,97x | 79,10 | 3,95x | 2641,23 | 131,74x |
| | Segura | 13,75 | 27,18 | 1,98x | 43,73 | 3,18x | 1822,03 | 132,50x |
| | Jiménez | 16,94 | 33,67 | 1,99x | 67,06 | 3,96x | 2637,06 | 155,62x |
| | Plücker | 13,46 | 26,68 | 1,98x | 51,27 | 3,81x | 1944,71 | 144,38x |
| Sculpture 277.004 triangles | Badouel | 12,54 | 24,93 | 1,99x | 36,88 | 2,94x | 1064,42 | 84,82x |
| | Möller | 21,75 | 42,92 | 1,97x | 85,60 | 3,93x | 2742,03 | 126,02x |
| | Segura | 13,77 | 27,50 | 1,99x | 33,10 | 2,40x | 1831,51 | 132,95x |
| | Jiménez | 19,02 | 37,60 | 1,98x | 74,95 | 3,94x | 2677,63 | 140,75x |
| | Plücker | 13,65 | 27,18 | 1,99x | 54,28 | 3,98x | 1981,39 | 145,15x |
| Dragon 871.414 triangles | Badouel | 12,82 | 25,41 | 1,98x | 38,75 | 3,02x | 1266,63 | 98,75x |
| | Möller | 21,98 | 43,41 | 1,97x | 86,64 | 3,94x | 2744,28 | 124,81x |
| | Segura | 13,66 | 27,09 | 1,98x | 42,60 | 3,12x | 1823,45 | 133,44x |
| | Jiménez | 19,26 | 38,08 | 1,98x | 75,85 | 3,94x | 2667,47 | 138,47x |
| | Plücker | 13,56 | 26,87 | 1,98x | 53,74 | 3,96x | 1945,56 | 143,46x |
| Buddha 1.087.716 triangles | Badouel | 13,15 | 25,84 | 1,97x | 31,71 | 2,41x | 1285,58 | 97,75x |
| | Möller | 21,99 | 42,93 | 1,95x | 85,93 | 3,91x | 2746,81 | 124,88x |
| | Segura | 13,71 | 26,96 | 1,97x | 44,28 | 3,23x | 1824,86 | 133,06x |
| | Jiménez | 19,69 | 38,42 | 1,95x | 76,74 | 3,90x | 2667,16 | 135,42x |
| | Plücker | 13,55 | 26,56 | 1,96x | 53,30 | 3,93x | 1945,96 | 143,60x |

processing units should execute the same instruction at the same time. In the SPMD system of the GPU, all available cores execute the same kernel on multiple parts of the data. However, the difference resides in the fact that the processors do not have to be executing the same instruction at the same time, and the execution order of the different groups of cores can be scheduled to avoid stalls caused by long-latency operations (e.g., global memory accesses and branch instructions).

Another drawback of the SSE programming paradigm is caused by the meticulous packaging of the data required before processing them with an SIMD instruction. This packaging of the data is a responsibility left to the programmer. That is, the scalability and granularity control must be handled by the CPU programmers, and thus, the design of the algorithms become more complicated. By contrast, the SIMT nature of the GPU allows a transparent way to handle the scalability of the algorithms.

## 4.2 Optional Features

We have carried out some additional experiments with alternative versions of each algorithm. These tests were designed to highlight the influence of different optional features (such as back-face culling, usage of pre-computed values or the calculation of the intersection point attributes) in the performance. Our motivation to independently study the impact of these features came from the fact that most optimizations reported in the literature for ray-triangle algorithms heavily rely on these features, especially the usage of pre-computed data.

Table 2 reports the performance gain obtained by including or omitting each feature. Clearly, we observe that enabling these features increases the performance in the single-core CPU scenario, which is the expected result. However, we also see that some of these popular optimizations can be counterproductive on the GPU (that is, the performance gain is almost zero or negative). Following, each feature is discussed in detail.

**Calculation of the intersection point attributes.** The calculation of the barycentric coordinates of the intersection point between the ray and the triangle did not noticeably affect the performance, whichever the version. Therefore, this feature can be safely used in any implementation without losing efficiency.

**Back-face culling.** This process discards triangles with the same orientation than the ray, that is, when the cosine of the angle between the triangle normal and the ray is positive. We found out that this optimization had a positive impact on the performance of most algorithms, especially on the CPU. This increment depended on the number of operations that each algorithm needed to perform before the back-face test can be carried out. On the GPU the performance gain was smaller, although it is always positive and therefore convenient to use.

**Segment-triangle algorithms.** Some applications require intersecting segments instead of rays. We found out that working with segments is not

Table 2: Experimental results of ray-triangle tests using several features. Values are in millions of intersection tests per second. The percentage gain of each version is related to the performance of the same algorithm running on the same hardware but without features. The triangle mesh used was Buddha. The total number of ray-triangle intersections for each test was 1 billion.

| | **Intersection point attributes** | | | | **Back-face culling** | | | |
|---|---|---|---|---|---|---|---|---|
| | CPU x1 | | GPU | | CPU x1 | | GPU | |
| | Mi/s | Gain | Mi/s | Gain | Mi/s | Gain | Mi/s | Gain |
| Badouel | 13,17 | 0,2% | 1289,63 | 0,3% | 18,00 | 36,9% | 1485,26 | 15,5% |
| Möller | 22,13 | 0,6% | 2751,68 | 0,2% | 25,86 | 17,6% | 2888,26 | 5,1% |
| Segura | 13,75 | 0,3% | 1831,48 | 0,4% | 19,93 | 45,4% | 1987,34 | 8,9% |
| Jiménez | 19,49 | -1,0% | 2666,84 | 0,0% | 26,29 | 33,5% | 2668,81 | 0,1% |
| Plücker | 13,96 | 3,0% | 1946,01 | 0,0% | 14,60 | 7,7% | 1946,14 | 0,0% |

| | **Segment-triangle intersections** | | | | **Pre-computed data** | | | |
|---|---|---|---|---|---|---|---|---|
| | CPU x1 | | GPU | | CPU x1 | | GPU | |
| | Mi/s | Gain | Mi/s | Gain | Mi/s | Gain | Mi/s | Gain |
| Badouel | 13,87 | 5,5% | 1275,23 | -0,8% | 16,22 | 23,3% | 1282,12 | -0,3% |
| Möller | 22,09 | 0,5% | 2746,80 | 0,0% | 22,12 | 0,6% | 2739,25 | -0,3% |
| Segura | 14,01 | 2,2% | 1836,02 | 0,6% | 15,78 | 15,1% | 1821,91 | -0,2% |
| Jiménez | 21,68 | 10,1% | 2667,14 | 0,0% | 24,73 | 25,6% | 2664,25 | -0,1% |
| Plücker | 13,54 | -0,1% | 1946,05 | 0,0% | 16,28 | 20,2% | 1945,01 | 0,0% |

| | **Interference test: culling+segments+p.data** | | | | **Ray casting (shared origin): attributes+p.data** | | | |
|---|---|---|---|---|---|---|---|---|
| | CPU x1 | | GPU | | CPU x1 | | GPU | |
| | Mi/s | Gain | Mi/s | Gain | Mi/s | Gain | Mi/s | Gain |
| Badouel | 24,48 | 86,1% | 1484,88 | 15,5% | 16,13 | 22,7% | 1413,20 | 9,9% |
| Möller | 25,86 | 17,6% | 2887,83 | 5,1% | 25,52 | 16,0% | 2783,49 | 1,3% |
| Segura | 26,99 | 96,8% | 1983,23 | 8,7% | 16,94 | 23,6% | 1880,90 | 3,1% |
| Jiménez | 42,69 | 116,8% | 2669,00 | 0,1% | 32,08 | 62,9% | 2681,24 | 0,5% |
| Plücker | 17,71 | 30,7% | 1946,25 | 0,0% | 17,15 | 26,6% | 1995,10 | 2,5% |

noticeably faster that working for rays for most of the algorithms. The exceptions were Segura's and Jiménez's methods, which are specifically optimized for this type of test. However, this improvement only applies to the CPU scenario, since we found no improvement on the GPU. This is because the segment-triangle test adds an additional check on the *t* parameter that implies additional flow-control instructions.

**Pre-processing.** According to our experiments, using pre-calculated values is a very interesting way to increase the performance of most CPU-based ray-triangle algorithms. The only exception was the Möller's algorithm, which does not rely on any pre-calculated data. In contrast, our experiments also showed that the use of pre-calculated values is typically counterproductive in the GPU environment. This stems from the fact that the time required for a GPU to compute these values is actually smaller than the time required to fetch them from memory. This fact is relevant because the use of pre-computed values is a universally acepted way to improve the efficiency of this kind of algorithms on the CPU. However, the reported results evident that this optimization is futile on the GPU. As a rule of thumb, a pre-computed value must save a very large amount of calculations in order to be worthwhile on a GPU. The ray-triangle algorithms typically use pre-calculated values to save single cross products or simple operations with vectors in run-time. However, a GPU can handle these operations very fast in a highly parallel way.

As can be appreciated, the optional algorithm features do not have a significant impact on the performance of the GPU-based implementations. Interestingly, the calculation of the attributes of the intersection point can be carried out without a noticeable performance loss. For the time-saving features, on the other hand, only the use of back-face culling implies some actual advantage on the GPU.

We have performed additional experiments based on the combination of several features that could be interesting for common applications, such as interference tests and ray casting. For evaluating the performance in a ray casting application, we created a set of rays that shared the same origin. Table 2 shows the results obtained under these circumstances. These configurations increased the efficiency of all the methods running on the CPU. However, the gain is more subtle with the GPU, although remarkable in some cases. Generally speaking, Möller's and Jiménez's were the most efficient ray-triangle algorithms for the majority of situations. Specifically, on the CPU the fastest method was Jiménez's when using back-face culling, pre-computed data and/or a shared origin for the rays. However, on the GPU the Möller's method outperformed the others in every aspect.

## 5 CONCLUSIONS

In this paper we have presented a comprehensive analysis of several ray-triangle intersection algorithms and their implementation in GPU using CUDA. When programming high-performance GPU algorithms, it is important to keep in mind that the GPU architecture forces the developer to adopt different principles than the CPUs. Following these principles, this paper has proposed some common design guidelines for successfully porting most of the ray-triangle algorithms to the GPU. Our exposition describes relevant aspects such as the kernel implementation, memory access issues and synchronization between threads.

We have also compared these GPU-based versions with the original CPU implementations. A thorough study of the influence of several optional features and performance optimizations has also been reported. Interestingly, our experiments show that the use of precomputed values (a typical optimization used by most CPU ray-triangle algorithms) is no longer required and even counterproductive. Also, those optimizations that require additional branch control instructions or global memory accesses should be minimized or avoided if possible, even if they apparently save some computations.

## ACKNOWLEDGEMENTS

This work has been supported by the *Ministerio de Economía y Competitividad* of Spain and the European Union (via ERDF funds) through the research project TIN2011-25259.

## REFERENCES

Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, pages 145–149.

Amanatides, J. and Choi, K. (1995). Ray tracing triangular meshes. In *Proc. Western Computer Graphics Symposium*, pages 43–52.

Badouel, D. (1990). An efficient ray-polygon intersection. In *Graphics Gems I*, pages 390–394. Academic Press Inc.

Benthin, C. (2006). *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University.

Foley, T. and Sugerman, J. (2005). Kd-tree acceleration structures for a gpu ray tracer. In *Proc. of the ACM Siggraph/Eurographics Conference on Graphics Hardware*, pages 15–22.

Glassner, A. (1989). *An Introduction to Ray Tracing*. Academic Press, New York.

Havel, J. and Herout, A. (2010). Yet faster ray-triangle intersection (using sse4). *IEEE Trans. Visualization and Computer Graphics*, 16(3):434–438.

Jiménez, J. J., Segura, R. J., and Feito, F. R. (2010). A robust segment/triangle intersection algorithm for interference tests. efficiency study. *Computational Geometry: Theory and Applications*, 43(5):474–492.

Jones, R. (2000). Intersecting a ray and a triangle with plücker coordinates. *Ray Tracing News*, 13(1).

Kensler, A. and Shirley, P. (2006). Optimizing ray-triangle intersection via automated search. In *Proc. IEEE Symposium on Interactive Ray Tracing*, pages 33–38.

Kim, S., Nam, S., Kim, D., and Lee, I. (2007). Hardware-accelerated ray-triangle intersection testing for high-performance collision detection. In *Proc. Journal of WSCG*, volume 15.

Kirk, D. and Hwu, W. (2010). *Programming Massively Parallel Processors*. Morgan Kaufmann.

Lin, M. and Gottschalk, S. (1998). Collision detection between geometric models: A survey. In *Proc. IMA Conf. on Mathematics of Surfaces*, pages 37–56.

Löfstedt, M. and Akenine-Möller, T. (2005). An evaluation framework for ray-triangle intersection algorithms. *Journal of Graphics Tools*, 10(2):13–26.

Möller, T. and Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28.

Noguera, J., Urena, C., and García, R. (2009). A vectorized traversal algorithm for ray tracing. In *Proc. of the Fourth International Conference on Computer Graphics Theory and Applications (GRAPP 2009)*, pages 58–63. INSTICC Press.

Plucker, J. (1865). On a new geometry of space. *Phil. Trans. Royal Soc. London*, 155:725–791.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712.

Rueda, A. J. and Ortega, L. (2008). Geometric algorithms on cuda. In *Proc. 3rd International Conference on Computer Graphics Theory and Applications (Grapp)*, pages 107–112.

Segura, R. J. and Feito, F. R. (1998). An algorithm for determining intersection segment-polygon in 3d. *Computer and Graphics*, 22(5):587– 592.

Segura, R. J. and Feito, F. R. (2001). Algorithms to test ray-triangle intersection. comparative study. *Journal of WSCG*, 9(3):76–81.

Shevtsov, M., Soupikov, A., and Kapustin, A. (2007). Ray-triangle intersection algorithm for modern cpu architectures. In *Proc. International Conference on Computer Graphics and Vision (GraphiCon 2007)*.

Woop, S., Benthin, C., and Wald, I. (2013). Watertight ray/triangle intersection. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):65–82.