Geometric algorithms on CUDA

Antonio Rueda, Lidia Ortega

Departamento de Informática
Universidad de Jaén
Paraje Las Lagunillas s/n. 23071 Jaén - Spain
email: {ajrueda,lidia}@ujaen.es

Abstract

The recent launch of the NVIDIA CUDA technology has opened a new era in the young field of GPGPU (General Purpose computation on GPUs). This technology allows the design and implementation of parallel algorithms in a much simpler way than previous approaches based on shader programming. The present work explores the possibilities of CUDA for solving basic geometric problems on 3D triangle meshes like the point inclusion test or the self-intersection detection. A solution to these problems can be implemented in CUDA with only a small fraction of the effort required to design and implement an equivalent solution using shader programming, and the results are impressive when compared to a CPU execution.

Keywords: GPGPU, CUDA, 3D triangle meshes, inclusion test, self-intersection test

First presented at the International Conference on Computer Graphics Theory and Applications (GRAPP) 2008, extended and revised for JVRB

1 Introduction

The General-purpose computing on graphics processing units (GPGPU) is a young area of research that has attracted attention of many research groups in the last years. Although graphics hardware has been used for general-purpose computation since the 1970s, the flexibility and power processing of the modern graphics processing units (GPUs) has generalized its use for solving many problems in Signal Processing, Computer Vision, Computational Geometry or Scientific Computing [OLG+07].

The programming capabilities of the GPU evolve very rapidly. The first models only allowed limited vertex programming; then pixel programming was added and gradually, the length of the programs and its flexibility (use of loops, conditionals, texture accesses, etc.) were increased. The last generation of NVIDIA GPUs (8 Series) supports programming at a new stage of the graphics pipeline: the geometry assembling. Several new programming languages like ARB GPU assembly language, GLSL [Ros06], HLSL or Cg [FK03] were developed aiming at exploiting GPU capabilities. GPU programming has been extensively used in the last years for implementing impressive real-time physical effects, new lighting models and complex animations [Fer04, PF05], and have allowed a major leap forward in the visual quality and realism of videogames.

But it should be kept in mind that vertex, pixel and geometry programming capabilities were aimed at implementing graphics computations. Their use for general purpose computing is difficult in many cases, implying the complete redesign of algorithms whose implementation in CPU require only a few lines. Clearly the rigid memory model is the biggest problem: mem-

urn:nbn:de:0009-6-348, ISSN 1860-2037

ory reads are only possible from textures or a limited set of global and varying parameters, while memory writes are usually performed on a fixed position in the framebuffer. Techniques such as multipass rendering, rendering to texture, and use of textures as lookup tables are useful to overcome these limitations, but programming GPUs remains being a slow and error-prone task. On the positive side, the implementation effort is usually rewarded with a superb performance, up to 100X faster than CPU implementations in some cases.

The last advance in GPGPU is represented by the CUDA technology of NVIDIA. For the first time, a GPU can be used without any knowledge of OpenGL, DirectX or the graphics pipeline, as a general purpose coprocessor that helps the CPU in the more complex and time-expensive computations. With CUDA a GPU can be programmed in C, in a very similar style to a CPU implementation, and the memory model is now simpler and more flexible.

In this work we explore the possibilities of the CUDA technology for performing geometric computations, through two case-studies: point-in-mesh inclusion test and self-intersection detection. So far CUDA has been used in a few applications [Ngu07] but this is the first work which specifically compares the performance of CPU vs CUDA in geometric applications.

Our goal has been to study the cost of implementing two typical geometric algorithms in CUDA and its benefits in terms of performance against equivalents CPU implementations. The algorithms used in each problem are far from being the best, but the promising results in this initial study motivate a future development of optimized CUDA implementations of these and similar geometric algorithms.

2 Common Unified Device Architecture (CUDA)

The CUDA technology was presented by NVIDIA in 2006 and is supported by its latest generation of GPUs: the 8 series. A CUDA program can be implemented in C, but a preprocessor included in the CUDA toolkit is required to translate its special features into code that can be processed by a C compiler. Therefore host and device CUDA code can now be combined in a straightforward way.

A CUDA-enabled GPU is composed of several MIMD multiprocessors that contain a set of SIMD processors [NVI07]. Each multiprocessor has a shared memory that can be accessed from each of its processors, and there is a large global memory space common to all the multiprocessors (Figure 1). Shared memory is very fast and is usually used for caching data from global memory. Both shared and global memory can be accessed from any thread for reading and writing operations without restrictions.

A CUDA execution runs several blocks of threads. Each thread performs a single computation and is executed by a SIMD processor. A block is a set of threads that are executed on the same multiprocessor and its size should be chosen to maximize the use of the multiprocessor. A thread can store data on its local registers, share data with other threads from the same block through the shared memory or access the device global memory. The number of blocks usually depends on the amount of data to process. Each thread is assigned a local index inside the block with three components, starting at (0, 0, 0), although in most cases only one component (x) is used. The blocks are indexed using a similar scheme.

A CUDA computation starts at a host function by allocating one or more buffers in the device global memory and transferring the data to process to them. Another buffer is usually necessary to store the results of the computation. Then the CUDA computation is launched one or more times by specifying the number of blocks, threads per block, and thread function. Pointers to data and results buffers are passed as parameters of the thread function. After the computation has completed, the results buffer is copied back to CPU memory.

The learning curve of CUDA is much faster than that of GPGPU base on shader programming with OpenGL/DirectX and Cg/HLSL/GLSL. The programming model is more similar to CPU programming, and the use of the C language makes most programmers feel comfortable. CUDA is also designed as a stable scalable API for developing GPGPU applications that will run on several generations of GPUs. On the negative side, CUDA loses the powerful and efficient mathematical matrix and vector operators that are available in the shader languages, in order to keep its compatibility with the C standard. Moreover, it is likely that in many cases an algorithm carefully implemented in a shader language could run faster than its equivalent CUDA implementation.

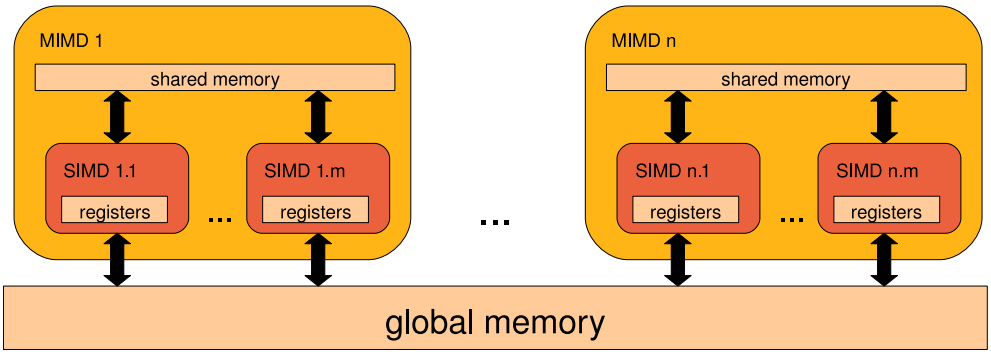

Figure 1: CUDA Architecture with n MIMD multiprocessors with $n \times m$ SIMD processors.

3 Point-in-mesh inclusion test on CUDA

The point-in-mesh inclusion test is a simple classical geometric algorithm, useful in the implementation of collision detection algorithms or in the conversion to voxel-based representations. A GPU implementation of this algorithm is only of interest with large triangle meshes and many points to test, as the cost of setting up the computation is high.

For our purpose we have chosen the algorithm of Feito & Torres [FT97] which presents several advantages: it has a simple implementation, it is robust and can be easily parallelized. The pseudocode is shown next:

```
bool inclusionTest(Mesh m, Point p) {
  Point o = pointCreate(0,0,0) // Origin point
  float res = 0; // Inclusion counter

  for (int nf = 0; nf < meshNumFaces(m); nf++) {
    Face f = meshFace(m, nf);
    Tetrahedron t = tetrahedronCreate(f, o);
    if (tetrahedronPointInside(t, p)) {
      res += 1;
    } else if (tetrahedronPointAtFace(t, p)) {
      res += 0.5;
    }
  }

  return isOdd(res);
}
```

The algorithm constructs a set of tetrahedra between the origin of coordinates and each triangular face of the mesh. The point is tested for inclusion against each tetrahedron and a counter is incremented if the result of the test is positive. If the point is inside an odd number of tetrahedra, the point is inside the mesh. Notice that if the point falls at a face shared by two tetrahedra, the counter is added 0.5 by each one to avoid a double increment that would lead to incorrect results.

The programming model of CUDA fits especially well with problems whose solution can be expressed in a matrix form. In our case, we could construct a matrix in which the rows are the tetrahedra to process, and the columns the points to test. This matrix is divided into blocks of threads, and each thread is made responsible of testing the point in the column j against the tetrahedron in the row i, and adding the result of the test $(0, 1, 0.5)$ to the counter j (see Figure 2). This approach has a minor drawback: in order to ensure a correct result after several add operations on the same position in global memory, performed by concurrent threads, support for atomic functions is required. This feature is only available in newer devices of GeForce and Quadro series with compute capability 1.1 [NVI07]. The need of atomic functions can be avoided if each thread stores the result of the point-in-tetrahedron inclusion test in the position (i, j) of a matrix of integers, but the memory requirements for this matrix can be very high when working with large meshes and many points to test. But the main problem of these two approaches is the high number of memory access conflicts that they generate, as every thread in row i requires triangle i to work and every thread in column j requires testing point j. This leads to poor results when compared with a CPU implementation of the algorithm.

We choose a different strategy, computing in each thread the inclusion test of one or several points against the entire mesh. Each thread iterates on the mesh, copying a triangle from global memory to a local variable and performing the inclusion test on the points, then it accumulates the result in a vector that stores an inclusion counter per point (Figure 3). It could be argued that the task assigned to each thread is very heavy, specially when compared with the matrix-based implementations, but in practice it works very

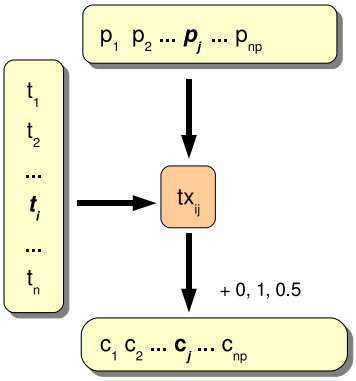

Figure 2: CUDA matrix-based implementation of the inclusion test.

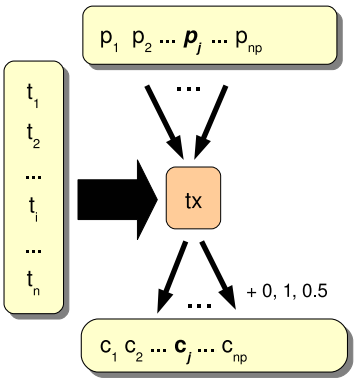

Figure 3: CUDA implementation of the inclusion test.

well. Two implementation aspects require a special attention. First, the accesses from the threads to the triangle list must be interleaved to avoid conflicts that could penalize performance. And second, the relatively high cost of retrieving a triangle from global memory makes interesting testing it against several points. These points must be cached in processor registers for maximum performance.

The host part of the CUDA computation starts by allocating a buffer of triangles and a buffer of points to test in the device memory, and copying these from the data structures in host memory. Another buffer is allocated to store the inclusion counters, which are initialized to 0. The number of blocks of threads (B) are estimated as a function of the total number of points to test (n), the number of threads per block (T) and the number of points processed by a single thread (P): $B = n/(T \cdot P)$. The last two constants should be chosen with care to maximize performance: a high number of threads per block limits the number of registers available in each thread, and therefore, the number of points that can be cached and processed. A low number of threads per block makes a poor use of the multiprocessors. Finally, after the GPU computation has completed, the buffer of inclusion counters is copied back to the host memory.

A thread begins by copying P points from the point buffer in global memory to a local array, that is stored in registers by the CUDA compiler. The copy starts at the position $(b_i \cdot T + t_i) \cdot P$, where b_i is the index of the block of thread and t_i is the index of the thread in the corresponding block. This assigns a different set of points to each thread. The iteration on the triangle list starts by copying each triangle to a local variable and calling a point-in-tetrahedron inclusion test function. In case of success, the inclusion counter of the corresponding point is updated. A good interleaving is ensured by starting the iteration at the position given by $b_i \cdot T + t_i$. Using float counters is not really necessary as we can add 2 when the point is inside a tetrahedron and 1 when it falls at a face, and divide it by 2 at the end of the iteration. The full source code of this thread is shown in the appendix.

We have compared the performance of a CPU version of the algorithm against the CUDA implementation using blocks of 64 threads and testing 16 points per thread. The computer used for the experiments has an Intel Core Duo CPU running at 2.4GHz., a NVIDIA GeForce 8800GTX GPU and Linux-based operating system. Four different models, with increasing number of faces were used (Figure 4). The improvements of the GPU against the CPU version of the algorithm are shown in Table 2. As expected, the CPU only beats the GPU implementation with very simple meshes. In the rest of cases, the GPU outperforms the CPU up to 77X in the case of the largest model and a high number of points to test. Notice that the GPU completes this computation in less than 7 seconds, while the CPU requires 8 minutes.

4 Self-intersection test on CUDA

Detecting self-intersections in a triangle mesh is useful in many applications. For instance, rapid prototyping using stereolithography technology usually requires hole-free meshes without self-intersections and consistent normal orientation in order to ensure a correct result. In interactive simulation of deformable objects, self-collisions are solved by detecting triangle pairs that do intersect. This problem is particularly difficult to solve efficiently, as updating pre-computed spatial data structures can be hard and inefficient with

mesh (number of triangles)	number of points tested 1000	10000	100000
simple (42)	0.05X	0.5X	3.7X
heart (1619)	1.2X	10X	38X
bone (10044)	2.6X	29X	55X
brain (36758)	3.5X	35X	77X

Table 1: Improvements of the GPU vs the CPU implementation of the inclusion test algorithm.

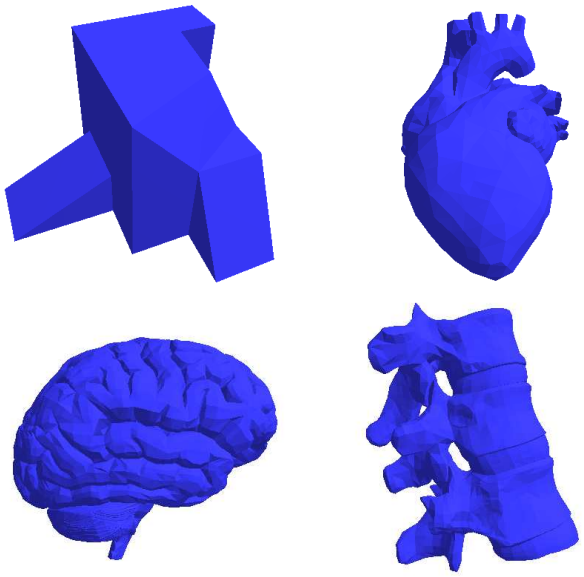

Figure 4: Models used for running the inclusion and self-intersection tests.

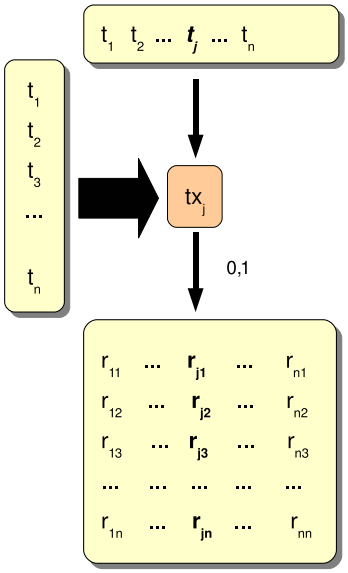

Figure 5: Generation of a matrix with the intersecting triangle pairs using CUDA.

large deformations.

We find different CPU strategies in the literature to detect, prevent or eliminate self-intersections in triangle meshes [LAM01, GD01, LL06]. Most of these methods are complex and inappropriate for many real-time applications. The algorithm of Govindaraju et al. [GLM05, GKLM07] is a GPU-accelerated collision detection algorithm that uses occlusion queries to cull non-intersecting triangles, but it still requires a final exact CPU-based intersection test on the remaining triangles of the Potentially Colliding Set. The method proposed by Choi et al. [CKK06] solves triangle intersections entirely in the GPU by using shader programming. It uses three 1D textures to store the triangles, a 2D texture for all pairwise combinations and a hierarchical structure in order to improve the readback of collision results from GPU. Prior to testing each pair of triangles for intersection, it performs a visibility culling similar to that proposed by Govindaraju et al. [GLM05], and a topology culling to avoid testing each pair of triangles twice. The remaining triangles are tested for collision using Möller triangle-triangle test [Mol97]. Although this method improves a CPU implementation up to 17X, the number of triangles that can be processed in a single pass is limited by the maximum size of a 1D texture in the GPU architecture.

We propose a simple solution to this problem using CUDA, based on testing each pair of triangles for intersection. The results can be stored in a symmetric matrix of booleans as shown in Figure 5, but its high storage requirements suggests using a different alternative. Our implementation only generates a list of indices of the intersecting triangle pairs, stored in a results buffer (Figure 6). The first position of this buffer keeps the current size of the list, and is updated by each thread every time a pair is added by using the atomic function *atomicAdd*.

Following a similar strategy to the point-in-mesh inclusion test described in the previous section, each

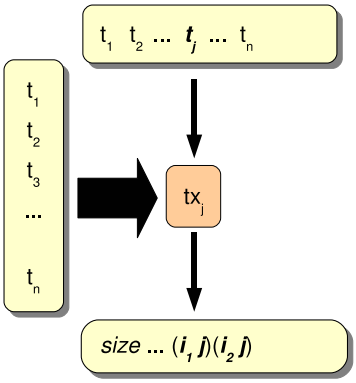

Figure 6: Self-intersection test using CUDA.

thread tests a triangle against the rest of triangles in the mesh, using the triangle-triangle test of Möller [Mol97]. In order to avoid duplicated comparisons, triangle t_i is only tested for intersection with triangles $t_{i+1}...t_n$. See the Listing 2 in the appendix for details.

Table 2 shows the results of a comparison against a CPU implementation considering different meshes. The system is similar to that used in the previous section, but the GPU is a 8800GTS with atomic functions support. In the CUDA implementation, the number of threads per block has been set to 64. The best results are found in larger meshes: for instance, the *brain* model requires a computation time of 3.6 minutes in CPU while the GPU detects the self-intersections in less than 6 seconds.

5 Conclusions and future works

The CUDA architecture allows the design and implementation of algorithms in the GPU with only a fraction of the effort required with shader-based programming. The memory model is now flexible, and the solution can be implemented in C language with little or no previous knowledge of the graphics pipeline operation.

Most geometric algorithms can be rewritten for CUDA, however the setup and readback time implied in the execution suggests its application to algorithms with a high computation cost. Both the self-intersection and the point-in-mesh inclusion test for large point sets presented in this work are cuadratic problems well suited for this purpose, running up to 77X faster than the CPU implementations with large meshes. We have implemented other geometric algorithms in CUDA like the computation of the axis-aligned minimal bounding box and convex hull of large meshes but the results have been poor. The first is a simple linear problem that requires little computational power, and in the second case the problem can hardly be decomposed into simpler independent tasks that can be assigned to the threads.

The benefits of CUDA technology can also be applied to other geometric problems like mesh simplification or computation of boolean operations. Currently we are also studying accelerating GIS raster operations by GPU using this programming model.

6 Acknowledgements

This work has been partially granted by the Ministerio de Ciencia y Tecnología of Spain and the European Union by means of the ERDF funds, under the research project TIN2007-67474-C03-03 and by the Conserjería de Innovación, Ciencia y Empresa of the Junta de Andalucía, under the research projects P06-TIC-01403 and P07-TIC-02773.

References

[CKK06] Yoo-Joo Choi, Young J. Kim, and Myoung-Hee Kim, *Rapid pairwise intersection tests using programmable gpus*, The Visual Computer **22** (2006), no. 2, 80–89.

[Fer04] Randima Fernando, *Gpu gems: Programming techniques, tips and tricks for real-time graphics*, Pearson Higher Education, 2004.

[FK03] Randima Fernando and Mark J. Kilgard, *The cg tutorial: The definitive guide to programmable real-time graphics*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FT97] F.R. Feito and J.C. Torres, *Inclusion test for general polyhedra*, Computer & Graphics **21** (1997), 23–30.

[GD01] James E. Gain and Neil A. Dougson, *Preventing self-intersection under free-form deformation*, IEEE Transations On Visualization and Computer Graphics **7** (2001).

mesh (number of triangles)	CPU time(ms)	GPU time(ms)	improvement
simple (42)	0.57	104	0.06X
heart (1619)	462	109	4.2X
bone (10044)	24524	583	42X
brain (36758)	217700	5596	39X

Table 2: Improvements of the GPU vs the CPU implementation of the self-intersection test algorithm.

[GKLM07] Naga K. Govindaraju, Ilknur Kabul, Ming C. Lin, and Dinesh Manocha, *Fast continuous collision detection among deformable models using graphics processors*, Comput. Graph. **31** (2007), no. 1, 5–14.

[GLM05] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha, *Quick-cullide: fast inter- and intra-object collision culling using graphics hardware*, SIGGRAPH '05: ACM SIGGRAPH 2005 Courses (New York, NY, USA), ACM, 2005, p. 218.

[LAM01] T. Larsson and T. Akenine-Möller, *Collision detection for continuously deforming bodies*, Proceedings of the Eurographics 2001, 2001, pp. 325–333.

[LL06] Jiing-Yih Lai and Hou-Chuan Laia, *Repairing triangular meshes for reverse engineering applications*, Advances in Engineering Software **37** (2006), no. 10, 667–683.

[Mol97] Tomas Moller, *A fast triangle-triangle intersection test*, journal of graphics tools **2** (1997), no. 2, 25–30.

[Ngu07] Hubert Nguyen, *Gpu gems 3*, Addison-Wesley Professional, 2007.

[NVI07] Corporation NVIDIA, *Nvidia cuda programming guide (version 1.0)*, NVIDIA Corporation, 2007.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell, *A survey of general-purpose computation on graphics hardware*, Computer Graphics Forum **26** (2007), no. 1, 80–113.

[PF05] Matt Pharr and Randima Fernando, *Gpu gems 2 : Programming techniques for high-performance graphics and general-purpose computation*, Addison-Wesley Professional, March 2005.

[Ros06] Randi J. Rost, *Opengl(r) shading language (2nd edition)*, Addison-Wesley Professional, January 2006.

Appendix

Listing 1: Source code of the point-in-mesh inclusion test thread.

```
__global__ void inclusionTestGPUThread(
  Triangle *dMesh,
  unsigned nTriangles,
  Point *dPoints,
  int *dTestResults,
  unsigned nTests)
{
  // Index of first point to process
  int np = (blockIdx.x * THREADS_PER_BLOCK +
    threadIdx.x) * POINTS_PER_THREAD;
  int npl;

  // Local copy of the points to process
  Point points[POINTS_PER_THREAD];
  // Result counters
  int c[POINTS_PER_THREAD];

  // Copy points to process
  for (npl = 0; npl < POINTS_PER_THREAD;
    npl++) {
    vertexCopy(points[npl], dPoints[np + npl]);
    c[npl] = 0;
  }

  // Start at a different triangle
  // to avoid conflicts
  int nt = np % nTriangles;
  int tc = nTriangles;

  // Local copy of the tetrahedron
  Point tetraPoints[3];

  // Iterate on the triangle list
  while (tc--) {
    vertexCopy(tetraPoints[0], dMesh[nt][0]);
    vertexCopy(tetraPoints[1], dMesh[nt][1]);
    vertexCopy(tetraPoints[2], dMesh[nt][2]);

    // Test points on the tetrahedron
    for (npl = 0; npl < POINTS_PER_THREAD;
         npl++) {
      // Add 2, 1 or 0 to the counter
      c[npl] += inclusionTestInTetrahedron(
        tetraPoints[0],
        tetraPoints[1],
        tetraPoints[2],
        points[npl]);
    }
    nt = (nt + 1) % nTriangles;
  }


  // Write results, dividing the counters by 2
  // and checking parity
  for (npl = 0; npl < POINTS_PER_THREAD; npl++) {
    dTestResults[np + npl] = ((c[npl] >> 1) & 1);
  }
}
```

Listing 2: Source code of the self-intersection test thread.

```
__global__ void selfIntersectionGPUThread(
  Triangle *dMesh,
  unsigned nTriangles,
  int *dTestResults)
{
  // Index of triangle to process
  int nt = blockIdx.x * BLOCK_SIZE + threadIdx.x;

  // Get triangle from global memory
  Vertex v0, v1, v2, u0, u1, u2;
  vertexCopy(v0, dMesh[nt][0]);
  vertexCopy(v1, dMesh[nt][1]);
  vertexCopy(v2, dMesh[nt][2]);

  // Start testing triangles from index nt + 1
  int ntt = nt + 1;

  while (ntt < nTriangles) {
    // Get triangle to test
    vertexCopy(u0, dMesh[ntt][0]);
    vertexCopy(u1, dMesh[ntt][1]);
    vertexCopy(u2, dMesh[ntt][2]);

    // Test triangles for intersection
    if (triTriIntersect(v0,v1,v2,u0,u1,u2)) {
      if (dTestResults[0] < MAX_RESULTS) {
        // Get and move current position of
        // the list. Store the indices of triangles
        int pos = atomicAdd(&dTestResults[0], 2);
        dTestResults[pos] = nt;
        dTestResults[pos + 1] = ntt;
      }
    }
    ++ntt;
  }
}
```