PARALLEL TRIANGULAR MESH REFINEMENT BY LONGEST EDGE BISECTION ON GPUs

by

Bilal Hatipoğlu

B.S, Computer Engineering, Yeditepe University, 2008

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2011

PARALLEL TRIANGULAR MESH REFINEMENT BY LONGEST EDGE BISECTION ON GPUs

APPROVED BY:

Assoc. Prof. Can Özturan (Thesis Supervisor)	.....
Prof. Oğuz Tosun	.....
Assoc. Prof. Ali Ecder	.....

DATE OF APPROVAL: 27.09.2011

ACKNOWLEDGEMENTS

I would like to thank my advisor, Assoc. Prof. Can Özturan, for pointing me in the right direction and sharing his valuable opinions during the course of this study. I am also grateful to Selçuk Cihan for providing me his LaTeX sources which assisted me in formatting this thesis.

ABSTRACT

PARALLEL TRIANGULAR MESH REFINEMENT BY LONGEST EDGE BISECTION ON GPUs

In numerical analysis, mesh refinement techniques are used in many different areas such as in Finite Element Methods for the solution of partial differential equations. Formerly, a variety of different mesh refinement techniques have been proposed including both sequential and parallel implementations on clusters and multi-core CPUs. Since, today, both computational capacity and memory bandwidth of GPUs are better and still developing faster than CPUs, general purpose computing on GPUs (GPGPU) has become important in many application areas that require high computation and data throughput. In this thesis, we focus on refining non-uniform triangular meshes and present a new parallel adaptive mesh refinement technique that can be easily implemented on GPU architectures. We also present an implementation of our algorithm on CUDA architecture that achieve significant speed-ups. This thesis includes the algorithm and implementation details, as well as running time analysis and performance comparison of sequential implemetation on CPU and parallel implementation on GPU.

ÖZET

GPU'LAR ÜZERİNDE UZUN KENAR BÖLME YÖNTEMİYLE PARALEL ÜÇGENSEL ÖRGÜ İYİLEŞTİRME

Nümerik analizde örgü iyileştirme yöntemleri, sonlu eleman yöntemleri gibi birçok alanda kısmi diferansiyel denklemlerin çözümünde kullanılmaktadır. Halihazırda kümeler ve çok çekirdekli CPU işlemciler üzerinde hem seri hem de paralel uygulamaları olan birçok örgü iyileştirme yöntemi bulunmaktadır. Ancak, bugün grafik işlemcilerin (GPU) gerek hesaplama kapasitesi, gerekse hafıza bant genişliği olarak CPU'lardan daha iyi bir noktada olduğu ve daha hızlı geliştiği göz önünde bulundurulduğunda, yüksek hesaplama ve veri hacmi ihtiyacı olan genel amaçlı uygulamaları da grafik işlemciler üzerinde çalıştırmak önem kazanmıştır. Bu tezde, düzensiz üçgensel örgülerin (non-uniform triangular mesh) iyileştirilmesi konusunda, grafik işlemciler üzerinde uygulanabilecek yeni bir paralel algoritma sunuyoruz. Algoritmamızı NDIVIA CUDA mimarisi üzerinde uygulayarak, grafik işlemci üzerinde CPU'ya kıyasla çok daha hızlı çalıştığını gösterdik. Bu tez aynı zamanda uygulama detaylarını ve karşılaştırmalı çalışma zamanı ve performans analizlerini de içermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . x
LIST OF ACRONYMS/ABBREVIATIONS . . . xii
1. INTRODUCTION . . . 1
1.1. Adaptive Mesh Refinement Techniques . . . 2
1.2. Parallel Adaptive Mesh Refinement Techniques . . . 4
1.3. GPU-vs-CPU . . . 5
2. NVIDIA COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA) . . . 7
2.1. Hardware Architecture . . . 7
2.2. Memory Bandwidth . . . 8
2.3. API and Execution Model . . . 8
2.4. Thread Hierarchy . . . 9
3. PARALLEL TRIANGULAR MESH REFINEMENT ALGORITHM . . . 11
3.1. Triangle-Edge Ownership . . . 13
3.2. Marks of Edges . . . 13
3.3. Edge Division Strategy . . . 13
3.4. Data Structures Used in the Algorithm . . . 15
4. DETAILS OF THE REFINEMENT ALGORITHM . . . 17
4.1. Steps 1 and 2 of the Algorithm . . . 17
4.2. Steps 3 and 4 of the Algorithm . . . 17
4.3. Steps 5 and 6 of the Algorithm . . . 20
4.4. Step 7: Refinement . . . 21
4.5. Example Run . . . 22
5. TESTS AND RESULTS . . . 26
5.1. Test Environment . . . 26

5.2. CPU-vs-GPU Runtime Analysis 28
6. CONCLUSION 32
6.1. Future Work 32
APPENDIX A: SOME EXAMPLES OF REFINED MESHES 34
APPENDIX B: DESCRIPTION OF THE SOURCE CODE 39
B.1. Input File Format 39
B.2. Output File Format 40
B.3. Source Code Snippet 40
REFERENCES 45

LIST OF FIGURES

Figure 1.1. An example unstructured triangular mesh before and after refinement. 2

Figure 1.2. Rivara's refinement templates. 3

Figure 1.3. The refinement of a mesh using regular refinement (top row) and bisection of the longest side (bottom row). The process of propagation is shown from left to right. 4

Figure 2.1. Thread and Memory hierarchy in CUDA. 10

Figure 3.1. An example showing the construction of directed trees (a), the propagation of refinement (b) and the final refined mesh (c). . . . 11

Figure 3.2. Steps of our parallel refinement algorithm. 12

Figure 3.3. Created edges after an edge division. 14

Figure 4.1. Details of Step 3. 18

Figure 4.2. Algorithm used in Step 4. 19

Figure 4.3. An example showing the operation of step 4: (a) three triangles are initially marked for refinement. (b) edge markings after first iteration. (c) final edge markings after second iteration. 20

Figure 4.4. Triangle templates for all 18 cases of triangle division. 23

Figure 4.5. An example mesh to refine. 24

Figure 4.6. Final mesh, after refinement of the mesh shown in Figure 4.5. . . . 25

Figure 5.1. Refinement time comparison for Tesla M2050, Intel X5550 and Intel X5570. 29

Figure 5.2. Speed-up by the number of Triangles between Tesla M2050 and Intel X5570. 29

Figure A.1. An example mesh having 100k triangles. 35

Figure A.2. The triangles in the lower left corner of the mesh in Figure A.1 are refined. 36

Figure A.3. Refined triangles are randomly distributed. 37

Figure A.4. All triangles of the mesh in Figure A.1 are refined. 38

Figure B.1. Example input file. 41

Figure B.2. Example output file. 42

Figure B.3. Code snippet of main() function. 43

Figure B.4. Code snippet of main() function. 44

LIST OF TABLES

Table 3.1. Fields of Point and Edge Records. 15

Table 3.2. Fields of Triangle Record. 16

Table 4.1. An example showing the prefix sum and new element slots relationship. 21

Table 4.2. Initial edge array for example mesh given in Figure 4.5. 24

Table 4.3. Initial triangle array for example mesh given in Figure 4.5. 24

Table 4.4. Edge array after step 2. 24

Table 4.5. Edge array after step 4. 24

Table 4.6. The final edge array after refinement. 25

Table 4.7. The final triangle array after refinement. 25

Table 5.1. Specifications of test hardware. 27

Table 5.2. Detailed timings for each step for Tesla M2050 (in ms). 30

Table 5.3. Detailed timings for each step for Tesla C1060 (in ms). 30

Table 5.4. Detailed timings for each step for Intel X5570 (in ms). 30

Table 5.5. Detailed timings for each step for Intel X5550 (in ms). 31

Table 5.6. Memory transfer time from host to Tesla M2050. 31

LIST OF ACRONYMS/ABBREVIATIONS

SIMD	Single Instruction Multiple Data
SP	Streaming Processor
SM	Streaming Multiprocessor
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
API	Application Programming Interface
CUDPP	CUDA Data Parallel Primitives Library
OpenMP	An API for multi-platform shared-memory parallel programming
MPI	Message Passing Interface
ECC	Error Correction Code
GPL	General Public License

1. INTRODUCTION

Partial differential equations are used for modeling many different physical problems. Two most common examples are the Poisson equation (e.g. for electric or gravity fields) and Navier-Stokes equation (e.g. fluid flow around an obstacle). Finite-element and Finite-volume methods that operate on spatial grids are widely used methods for solving such equations [1]. The combination of finite element methods and adaptive mesh refinement provides an effective solution in a variety of applications.

GPU architectures are designed to run in an efficient way if the parallel parts of the running code are distributed among similar threads which *converges* in execution, i.e. executes the same set of instructions on different data in an ordered way. The memory accesses for each operation should also occur in an ordered way among threads to effectively use available memory bandwidth. Achieving synchronization is hard and costly in GPUs as well as the atomic operations, which results in serialization of threads.

Mesh refinement problem is not structured and hence, it is not trivial to divide the problem into convergent threads or access data elements in a uniform fashion. Atomic operations or barrier synchronization are required during the execution. We contribute a new parallel adaptive mesh refinement algorithm specialized for GPUs and implement it on CUDA architecture. First, we review the literature on mesh refinement techniques and recent algorithms. Then, we provide a short introduction about GPU computing and CUDA architecture. Our algorithm is detailed in Chapters 3 and 4. At the end, we provide results of our experiments, discuss advantages/disadvantages of our implementation and conclude with ideas for improvements that can be made in the future.

1.1. Adaptive Mesh Refinement Techniques

For solving many partial differential equations, adaptive mesh refinement techniques are used to reduce the computational and storage requirements. [2]. A uniform mesh has grid points evenly spaced on a domain. Instead of using a such a uniform mesh, adaptive mesh refinement techniques makes it possible to place more elements in order to concentrate into the areas where the local error in the solution is largest. According to local error estimates, the mesh is adaptively refined and/or coarsened during the computation. This technique is much more efficient than the use of uniform meshes when the solution is changing much more rapidly in some areas than in others [3].

Some example meshes are illustrated in [4]. An example mesh which has initially 245 triangles and models a square plate with a square hole in the center is given in Figure 1.1. The initial mesh is on the left. The triangles around the hole are refined twice to obtain the refined mesh on the right.

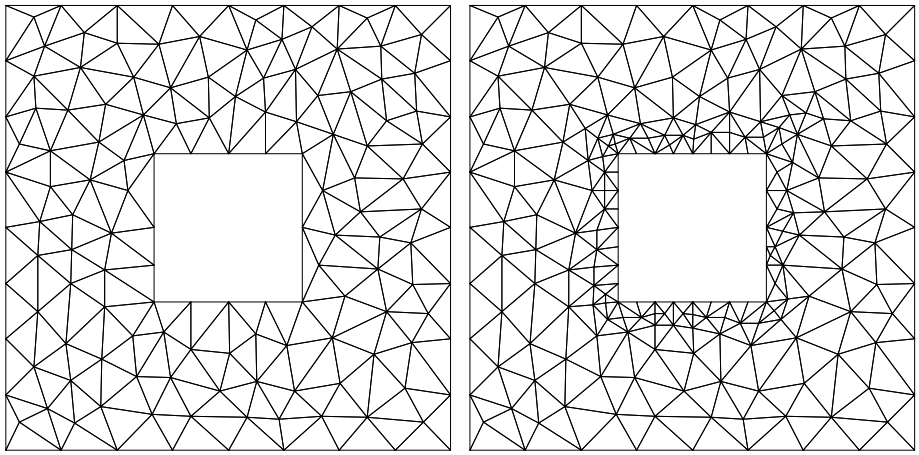


Figure 1.1. An example unstructured triangular mesh before and after refinement.

Two basic algorithms exist for the refinement of triangles: (i) bisection, and (ii) regular refinement. The *bisection* algorithm contributed by Rivara [5], bisects the longest edge of a triangle to form two new triangles with equal area. According to the Rivara's longest edge bisection method, possible division scenarios for a triangle is illustrated in Figure 1.2, called *refinement templates*.

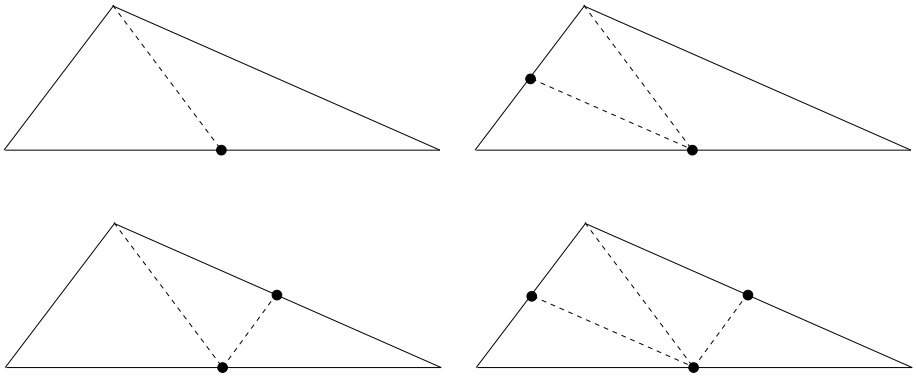


Figure 1.2. Rivara's refinement templates.

The second algorithm, called *regular refinement* due to Bank [6] divides a triangle into two or four similar triangles.

Refinement algorithms must finish with a conforming mesh. A conforming mesh is a mesh in which for all triangles in the mesh the edge of a triangle cannot contain a vertex other than its endpoints. In both Rivara's and Bank's methods, the refinement of a single triangle usually causes a propagation of refinement to other mesh elements. This propagation ensures that the final mesh satisfies the conformity criterion. The two methods differ in how they handle this propagation. The example illustrated in [7] provides easy understanding of the propagation and the difference between both methods. Figure 1.3, shows the operations of both of the algorithms as refinement occurs. The shaded triangles are the elements that have just been refined. The sequence of meshes from left to right shows how the refinement propagates to neighboring triangles.

Besides (i) the conformity requirement, the following additional properties are highly desirable in a refined mesh: (ii) the mesh gradation should be smooth, i.e., the areas of neighboring triangles should not differ drastically, and finally (iii) the angles in the mesh should be neither too small nor too large [8, 9]. Rivara's refinement algorithm [5] which is based on bisecting the triangles by their longest edge satisfies properties (i) and (ii). In relation to property (iii), it is proved that the smallest angle in the (successively) refined mesh is bounded by at worst one-half the smallest angle

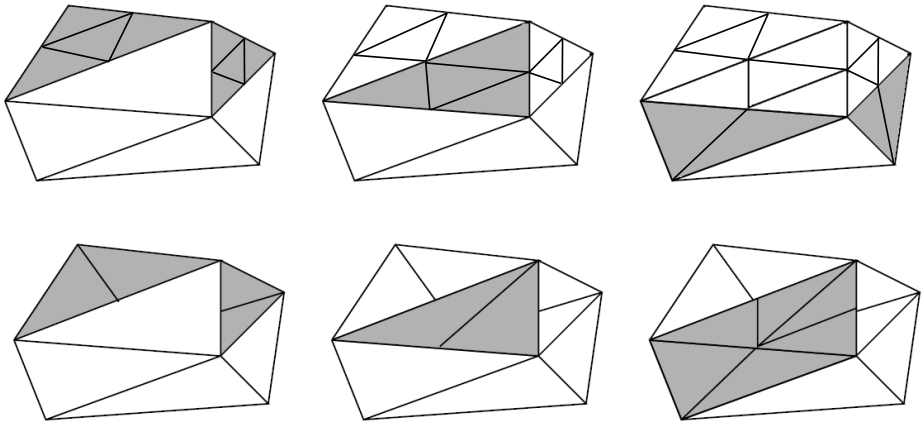


Figure 1.3. The refinement of a mesh using regular refinement (top row) and bisection of the longest side (bottom row). The process of propagation is shown from left to right.

in the original mesh [10].

1.2. Parallel Adaptive Mesh Refinement Techniques

A variety of techniques have been proposed to refine nonuniform triangular meshes sequentially in [5, 6, 11, 12].

Parallelization of mesh refinement procedures based on longest edge bisection have been considered in [3, 13–15]. The refinement procedures implemented by Jones and Plassmann [3] use randomized graph coloring heuristics to resolve data structure update conflicts. The data structures used in [13, 14] do not necessitate the use of any graph coloring heuristic to implement parallel mesh refinement. All these implementations, however, does not fit the requirements of an efficient implementation on GPUs and are hard to implement on GPUs because of either their synchronization requirements or diverging execution models.

A logarithmic data parallel algorithm to refine triangular meshes by Rivara's longest edge bisection procedure is presented by Ozturan [10]. Unlike the previous algorithms which apply the refinement templates and propagate refinement simulta-

neously, this algorithm first propagates the edge-markings to satisfy conformity and then applies the templates at the end. Application of the templates can be parallelized easily after appropriate edges have been marked for refinement. It also proposes the construction of a directed data dependency graph of refinement propagation, which determines the propagation and defines which triangles will be refined according to the propagation, before starting the refinement. These properties makes this algorithm easy to implement on GPUs with some minor modifications. So, we used this algorithm as a base in our proposed algorithm and its implementation.

We implemented our algorithm on NVIDIA's CUDA architecture and the analysis are done accordingly. However, our algorithm is not CUDA dependent, and can be implemented in other CPU and GPU architectures and paradigms.

1.3. GPU-vs-CPU

Increasing computational power and the fundamental paradigm shift of the underlying hardware towards parallelism and heterogeneity enables computational scientists to solve more complicated problems. Huge power demand and increased heat dissipation caused by increasing number of transistors per area pushes the chip manufacturers to scale the number of cores per chip rather than clock frequencies. But the number of transistors per area is approaching to the physical limits. At the same time, large increase in computational power causes the memory bandwidth to be a bottleneck. This memory wall problem is worsened in multi-core architectures: The available memory bandwidth typically scales with the number of sockets per compute node and not with the number of cores per chip. Since today's multi-core architectures are shared memory architectures, the contention caused by sharing the memory into multiple cores also degrades the memory performance. It gets even worse when the number of cores increases. However, graphics processor units (GPUs) provide a much higher memory bandwidth and processing throughput than commodity CPU designs. Considering the fact that number of cores increases day by day, GPU architecture and programming model is representative of future many-core architectures [16].

The programmability and performance of modern graphics processors (GPUs) has increased drastically and even outperform CPUs in some compute intensive applications. Since the performance difference is expected to increase in the future [17,18], for certain applications in many kind of different areas, researchers are adapting computationally intensive general-purpose algorithms to run on GPUs, and they usually achieve outstanding speed-up versus their CPU counterparts. Among them, some examples are given in [19], which achieve up to 10-20 times faster computation.

In recent years, there has been a growing body of work concerned with use of graphics processing units (GPUs) for general purpose computations (so-called GPGPU [20]). The motivation for this work has been the increasing flexibility of GPU architecture and the corresponding improvements in their programmability, as well as the dramatic rate of increase in their computational capacity far quicker than that of CPUs [21]. From the software perspective, the advent of NVIDIA's CUDA and AMD's STREAM platforms in late 2006 has resulted in a major surge of interest in developing algorithms and provides easier and more flexible implementation. The survey articles by Owens *et al.* [21, 22] provide an excellent overview of the field in terms of applications and techniques, and the community website GPGPU.ORG and the vendors' websites are referred for details and sample applications as well as the actual implementation of scientific algorithms on graphics hardware.

2. NVIDIA COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

2.1. Hardware Architecture

Like the modern GPU architectures, the CUDA does have a unified computing unit called "streaming processor" (SP), instead of two different, specialized vertex shader and pixel shader computing units. For example, the NVIDIA 8800GTX contains 128 SPs. Every 8 streaming processors constitute a "streaming multiprocessor" (SM). The SM has the signal instruction, multiple data (SIMD) architecture that executes the same instructions on different data.

There are four types of on-chip memory on CUDA architecture. First one is the general propose local registers per processor, which is the fastest in terms of access time. Second one is the shared memory which is shared by all SPs within a single SM, which allows SPs to communicate with each other without passing data outside the chip. Shared memory is shared among the threads running on the same block. Third one is a constant cache which is a read-only memory and maps to the constant memory, which is mostly used to store constant general data and the last type of on-chip memory is the texture cache which is also read-only and maps to the texture memory which is mostly used to store constant texture data. Constant cache and texture cache of the device DRAM is accessible by all SPs. In addition to the read-only constant cache and texture cache there exists a global on-device memory which is both readable and writable by SPs. In the former graphic cards, since the global memory lacks cache, the data access to the global memory is much slower than that of the constant memory and the texture memory. The most recent CUDA enabled GPUs having Compute Capability 2.0 has a multi-level cache on top of global memory, which significantly increases the local and global memory transaction speed for certain memory access patterns [23], but it's still more expensive compared to local memory access.

The details about the hardware architecture of the NVIDIA Tesla products which is used in the experiments mentioned in this paper can be found in [24].

2.2. Memory Bandwidth

The effective bandwidth of each memory space depends significantly on the memory access pattern. Since the device global memory is of much higher latency and lower bandwidth than on-chip shared memory, global memory accesses should be arranged so that simultaneous memory accesses of one block can be coalesced into a single contiguous, aligned memory access. This means that each block thread number N should access element N at byte address $BaseAddress + sizeof(type) \times N$, where N starts from zero and $sizeof$(type) is equal to 4, 8 and 16. Moreover $BaseAddress$ should be aligned to $16 \times sizeof(type)$ bytes, otherwise memory bandwidth performance breaks down significantly [23].

Starting from CUDA devices which support compute capability 2.x, there is a 2-level cache on top of device memory. This cache caches the accessed memory blocks upon a read/write operation, which eliminates the access pattern alignment requirement mentioned above for memory coalescing. As long as all threads in a warp try to access the same memory block (or another memory block which already stays in the cache) regardless of which thread needs which exact memory location the memory access on this warp is coalesced, since the entire block is cached. Our experiments also show that this cache implementation achieves significant speed-up in general purpose applications.

2.3. API and Execution Model

CUDA has C language extension which allows integrating GPU-specific code with C-based programs. CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. Kernels, coded

as C functions and invoked directly. But, kernels in a graphics API implementation are invoked indirectly.

CUDA C API offers two main features: (i) ease of coding by removal of irrelevant graphics-associated code, and (ii) greater flexibility in memory access. In particular, the memory access flexibility raises the possibility of scattered writes with some restrictions. For example, while scattering is allowed using CUDA there is no inbuilt mechanism for handling write conflicts between threads, and in fact implementation of such a mechanism is difficult due to the non-atomicity of GPU operations [25].

2.4. Thread Hierarchy

In CUDA programming, the original program is first compiled to conform to the CUDA device instruction set and it becomes a new parallelized program, The kernel, and is downloaded to the GPU device that acts as a co-processor to the host (CPU). It is executed by the mechanism of "threads" that are organized as thread blocks. The threads within a thread block can co-work with each other through the shared memory and can synchronize their execution to coordinate their memory access. Thread and memory hierarchy is shown in Figure 2.1. The maximum number of threads within a thread block is limited; however, thread blocks that execute the same kernel can be batched together to form a grid of blocks. Therefore, the total number of threads that execute a single kernel can be much larger. Nevertheless, threads in different thread blocks are unable to access the same shared memory and thus, they run independently. It is also impossible to provide implicit synchronization between threads of different blocks [26]. The SM may execute one or more thread blocks concurrently depending on the shared memory and register occupancy. Each thread blocks is split into SIMD groups of threads called "warps"; these warps contain 32 successive threads that are executed by the SM in a SIMD fashion. Since the shared memory has only 16 banks, the shared memory requests in a warp are split into two halves. A thread scheduler periodically switches from one warp to another to maximize the SM's computational resources [23].

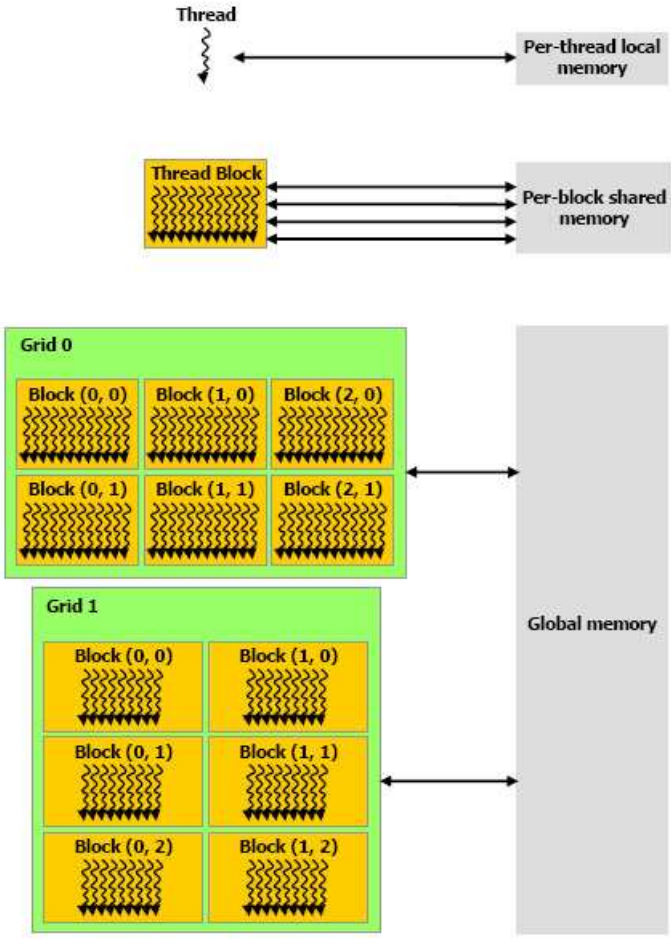


Figure 2.1. Thread and Memory hierarchy in CUDA. [23]

3. PARALLEL TRIANGULAR MESH REFINEMENT ALGORITHM

Current triangular mesh refinement techniques form a data dependency due to the refinement propagation, which can be expressed as a forest of directed trees. This data dependency makes the problem hard to run in parallel. One approach is to partition the mesh into independent parts, and run each part on a different CPU. But this approach has high overhead of partitioning and lacks of scalability. Our approach is to generate the dependency tree, determine the triangles to refine and apply the refinement templates (creating the new edges and triangles), all done in parallel in each other.

Figure 3.1 shows an example illustration of Rivara's longest edge bisection mesh refinement and shows the propagation of the refinement.

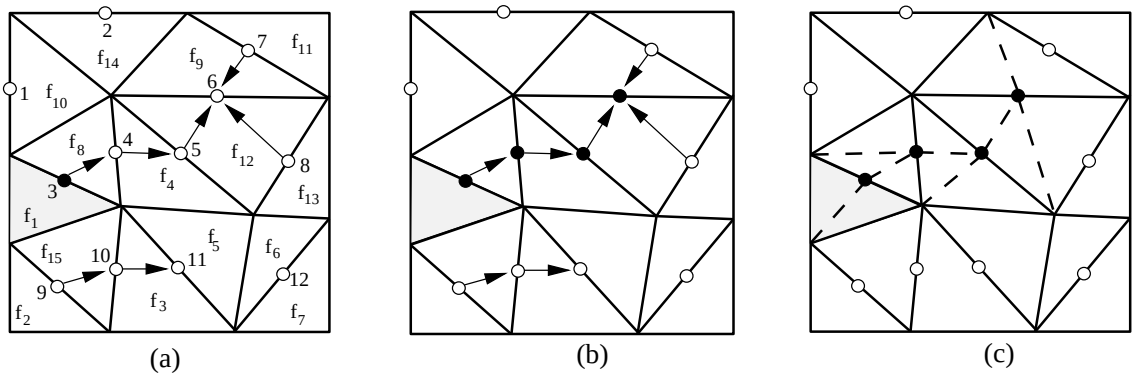


Figure 3.1. An example showing the construction of directed trees (a), the propagation of refinement (b) and the final refined mesh (c).

Most of the recent algorithms refine the triangles gradually, i.e. bisect the triangle to refine to two, then check the conformity and bisect other triangles if needed. Then, simultaneously check if additional bisection is required, and apply bisection one by one. In this approach, which triangles will be refined and how many additional edges triangles will be created is not known at the beginning. However, this information is required in order to allocate additional space in device memory before the

actual refinement work. Our algorithm first propagates the edge-markings to satisfy conformity, then calculates a prefix count of which triangles are divided into how many parts. These operations all can be done in parallel in each other. After this, we allocate the required space, and at the end, apply the templates in a single step, in parallel.

The steps of our algorithm is given in Figure 3.2. Each step itself is designed to run by multiple threads in parallel, providing massively data parallel execution without any need for synchronization or atomic operations.

```
Algorithm GPUParallelRefine
begin
```

1. Calculate edge length and mid point of each edge.
2. Mark longest edge of each triangle (white circular marks in Figure 3.1a) and additionally mark the longest edges of the triangles to be refined for division (black marks in Figure 3.1b).
3. Form dependency trees of the edges by creating dependency links between marked edges.
4. Propagate the edges to be divided by following the links on dependency tree.
5. Get the count of edges to be divided for each triangle.
6. Perform prefix count of edges to be divided for each triangle, to calculate the new edge and triangle count and allocate new space for the edges and triangles to be created.
7. Refine each triangle (create new edges and triangles) simultaneously, by applying the appropriate refinement templates according to the divided edges.

```
end
```

Figure 3.2. Steps of our parallel refinement algorithm.

We will first explain some key concepts used in the algorithm. Detailed descriptions of algorithmic steps are explained in Chapter 4.

3.1. Triangle-Edge Ownership

Each edge is shared by two triangles, except for the boundary edges. To prevent synchronization issues or write-after-write hazard, we assumed that the input edges are directional, and each triangle owns the edges oriented in *counter-clockwise* direction. In this way, interior edges are owned by only one triangle, but this requires the boundary edges to be in the direction of the boundary triangle. Most of the mesh generation applications and procedures satisfy this (like the Triangle Mesh Generator [27] that we used to generate input), so this is not a problem at all. To indicate the triangle-edge ownership, in the input file, the indexes of edges owned by a triangle are set to positive numbers, and the indexes that are in the reverse direction are taken as negative. If the input does not satisfy this rule, it should be converted to this format.

3.2. Marks of Edges

In the implementation, there are two marks (boolean values indicating marked or not) that are used for each edge. "Mark" is for indicating that an edge is a longest edge of a triangle (and not necessarily the owner triangle), and "Black Mark" is for indicating that in the refinement step, the edge will be bisected. These are just the names that we define for them and this notation is used in the algorithm description below.

3.3. Edge Division Strategy

We apply Rivara's refinement templates given in Figure 1.2 to each triangle to be refined, using the following rules:

(i) A black-marked edge owned by a triangle is divided into two by the thread responsible for that triangle. Two new edges are created in the same direction of the original edge (edges a and b in Figure 3.3). After that, the edge itself is updated to form a new interior edge, from the opposite point to the mid point

of the divided edge, always having the direction towards the divided edge (edge c in Figure 3.3).

(ii) A black-marked edge in the reverse direction of a triangle leads to the creation of a new interior edge in this triangle, from opposite point to the divided edge. This edge is created in such a way that, it always have direction towards the divided edge (edge d in Figure 3.3).

(iii) One black-marked edge divides the triangle into two; two black-marked edges divides the triangle into three; and three black-marked edges divides the triangle into four. By updating the data structure of the original triangle, one of the created triangles will be stored in the memory location of the original triangle.

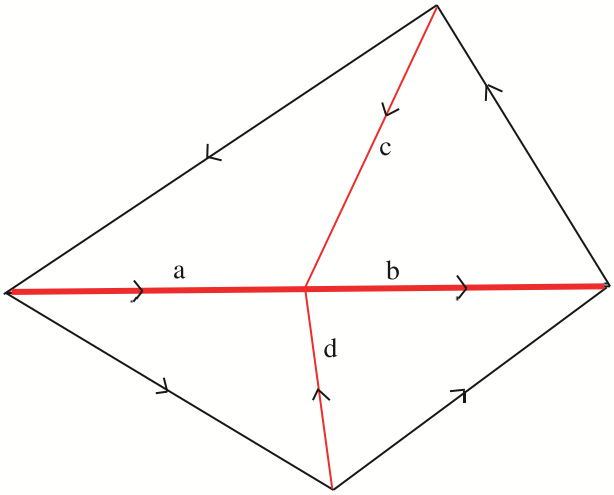


Figure 3.3. Created edges after an edge division.

Let f denote the number of divided edges in the forward direction of a triangle, let r denote the number of divided edges in the reverse direction of a triangle, and let new_t and new_e stands for the number of newly created triangles and the number of newly created edges respectively, in the triangle, according to the division strategy above, we get the following formula regarding the number of newly created elements: $new_t = f + r$ and $new_e = 2 \times f + r$

3.4. Data Structures Used in the Algorithm

Three records are used to represent Point, Edge and Triangle records respectively. Table 3.1 and 3.2 gives the fields of these records. In the implementation phase, rather than using array of these records to store input and output data, we prefer to use separate arrays for each element of these records. In this way, we maximize the alignment of memory accesses by the threads which in turn facilitates coalescing, and reduces the memory accesses and hence execution time.

Table 3.1. Fields of Point and Edge Records.

Record	Type	Name	Description
Point	float	x	x coordinate of point
	float	y	y coordinate of point
Edge	Point	$point0$	start point of edge
	Point	$point1$	end point of edge
	Point	mid_p	mid-point of edge
	float	len	length of edge
	unsigned int	mn	joint structure used to store edge mark (first two bits) and index of next edge in dependency tree (remaining 30 bits)
	int	new_edge0	index of newly created edge (left side) if this edge is bisected
	int	new_edge1	index of newly created edge (right side) if this edge is bisected
	int	new_edge2	index of newly created edge (opposite side) if this edge is bisected

Table 3.2. Fields of Triangle Record.

Record	Type	Name	Description
Triangle	int	$edge0$	index of first edge (counter-clockwise order)
	int	$edge1$	index of second edge (counter-clockwise order)
	int	$edge2$	index of third edge (counter-clockwise order)
	int	$longest$	index of longest edge
	unsigned int	cnt	counter for the # of forward-edges to be divided
	unsigned int	rev_cnt	counter for the # of backward-edges to be divided

4. DETAILS OF THE REFINEMENT ALGORITHM

4.1. Steps 1 and 2 of the Algorithm

In *step 1*, length and mid point of each edge is calculated and stored in an array. It is obvious that the operation on each edge is independent from other edges, so, this operation is done concurrently for each edge in $O(1)$ time by assigning one thread to each edge. The calculated edge lengths are used in the next step, and the mid points are used in the refinement step, if the edge is divided.

In *step 2*, we mark the longest edge of each triangle by comparing the lengths of the edges of a triangle. This step is also independent, even all of the edges are shared by two triangles except the boundary edges. Since the written data (mark) is independent than the read data (edge lengths), there is no read-after-write hazard. On the other hand, if one edge is the longest edge of both two triangles sharing that edge, it will be marked twice. Since the mark is the same boolean data, even if there is a write-after-write hazard, it will not cause any problem. So, by assigning one thread to each triangle, this step will take $O(1)$ time for each thread running in parallel. In this step, the longest edges of the triangles to be refined are also marked as "should be divided", which is denoted by "Black Mark". This mark is stored in a different array than the previous mark for preventing a write-after-write condition. This mark is used for generating the edge dependency tree in *step 3*.

4.2. Steps 3 and 4 of the Algorithm

In *step 3*, we form an edge dependency tree. Actually, for the entire mesh, the edge dependency tree is a forest of directed trees, indicating that if an edge is divided according to the refinement, all child edges of this edge should also be divided. Forming the dependency tree is illustrated in Figure 3.1a. The algorithm is described in Figure 4.1.

```
T: Array of input triangles
LE(t): Longest edge of triangle t
Mark(e): Boolean value which is true if edge e is marked, false otherwise
Next(e): Index of the child element of edge e in the dependency tree

For each t in T in parallel do
    For each edge e of t
        If (e is not LE(t)) and (Mark(e) is true)
            Next(e) ← LE(t)
        End If
    End For
End For
```

Figure 4.1. Details of Step 3.

According the theorem 3.1. in [10], there exists at most one link out from one edge. So, we store the tree as an array that consists of the index of next edge for each edge (which is Next(e) in the pseudo-code above). Since the next index of an edge may be written by only one triangle (which is the owner triangle of the edge), there is no write-after-write hazard. So, by assigning one thread for each triangle, this step will also be completed in $O(1)$ time in parallel.

In *step 4*, we propagate the division of each divided edge to its child edges in the dependency tree. So, after this step, the edges to be divided according to the refinement operation are stated. The propagation is illustrated in Figure 3.1b. By marking the divided edges instead of the refined triangles, we can calculate how many edges and triangles will be created after the refinement (explained in the next step). Although this step can be done theoretically on $O(logn)$ time according to the theorem 3.2 of [10] for implementation simplicity and practical reasons this step is implemented using the algorithm described in Figure 4.2, which has a worst case complexity of $O(n)$.

An example is illustrated in Figure 4.3 to show the operation of *step 4*: Three

```
E: Array of input edges
Blackmark(e): Boolean value which is true if edge e is
              black marked (mark to be divided), false otherwise
Next(e): Index of the child element of edge e in the dependency tree

For each e in E in parallel do
    If (Blackmark(e) is true)
      While Next(e) is not NULL
            If (Blackmark(Next(e)) is false)
              Blackmark(Next(e)) ← true
              Next(e) ← Next(Next(e))
            Else
              Next(e) ← NULL
            End If
      End While
    End If
End For
```

Figure 4.2. Algorithm used in Step 4.

triangles are initially marked for refinement, so, three threads are executing concurrently to propagate the refinement and mark the corresponding edges. After two iterations, the operation is completed.

This algorithm is designed to minimize the number of memory transactions, by accessing each edge to be marked only once, and have a worst case running time of $O(n)$. But, in practice, by assigning one thread to each black marked edge, this step is done in parallel and for most of the practical cases the propagation of each edge is limited to a couple of other edges.

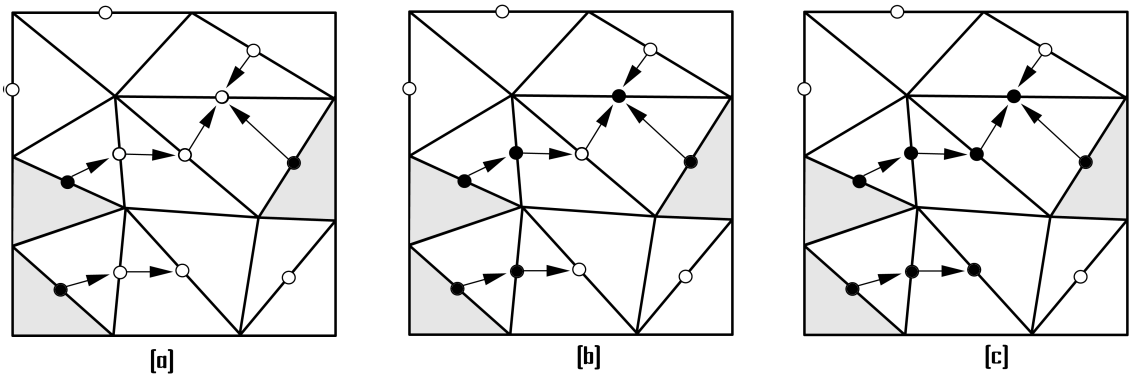


Figure 4.3. An example showing the operation of step 4: (a) three triangles are initially marked for refinement. (b) edge markings after first iteration. (c) final edge markings after second iteration.

4.3. Steps 5 and 6 of the Algorithm

In *step 5*, we get the count of black-marked edges of each triangle. By using two separate counters for forward and reverse direction edges of a triangle, we are able to apply the division strategy defined in Section 3.3.

In *step 6*, we run a prefix sum on the counters we get in the previous step, to calculate how many additional memory space is needed for new edges and triangles. For parallel concurrent creation of new elements without using any locking mechanism, each triangle should know which slots it should use for the creation of new elements that it is responsible to. This prefix is also used to provide this information. To obtain the prefix sum of the array, we used CUDPP library [28]. CUDPP library performs very fast computation of prefix sum on CUDA and it is able to be configured as needed, easily.

Table 4.1 shows an example to clarify how the prefix sum is computed from forward (f) and reverse (r) counts and how the new element slots are determined.

Table 4.1. An example showing the prefix sum and new element slots relationship.

Triangles	f	r	prefix f	prefix r	new triangle slots	new edge slots
T0	0	2	0	2	0 to 2	0 to 2
T1	1	1	1	3	2 to 4	2 to 4
T2	1	0	2	3	4 to 5	5 to 7
T3	2	0	4	3	5 to 7	7 to 11
T4	0	1	4	4	7 to 8	11 to 12
T5	0	0	4	4	-	-
T6	1	1	5	5	8 to10	12 to 15

4.4. Step 7: Refinement

This last step is where the actual triangle subdivision is done. New elements are created according to Rivara's refinement templates shown in Figure 1.2. This step is the hardest to implement compared to the previous ones due to the concurrent accesses/updates on the same elements, that cause write-after-read and write-after-write conflicts. The problem arises because the old locations of divided edges and triangles may be updated and used for one of the new ones for memory efficiency. The problem is as follows: If we update the shared entities (mostly the divided edges which are shared by two triangles) before they are used, we totally corrupt the original mesh. So, we need synchronization points to coordinate this step. For this reason, we divided this step into three sub-steps, which are executed in order, i.e. each sub-step runs after the previous sub-step has completed.

If an edge is divided, four new edges are created: Two is for the two divided parts of the edge, and two is for the opposite edges on both sides, to satisfy conformity (see Figure 3.3). The creation of the first two edges plus one of the opposite ones is the responsibility of the owner triangle of that edge. The creation of the reverse opposite one is the responsibility of the neighbour triangle, i.e. each triangle is responsible for the edges owned by them, plus the ones created inside them. Since we run the algorithm in parallel by assigning each triangle to a different thread, this is the best way to divide the duties and provide independence: Each triangle accesses its own data and knows what is going on inside it.

First sub-step is the creation of new edges in the newly allocated memory slots (edges *a*, *b* and *d* in Figure 3.3). Since they will not be accessed until the creation of triangles, it can be done in parallel without any problem. But, updating the memory location of the original edge should be done after all new edges are created, which is done in the *second sub-step* (edge *c* in Figure 3.3). After filling the new edge slots, the index of each newly created edge caused by division of one edge, is recorded in the edge structure of the original edge, since the triangles are still pointing to their old edges and they need this information to form the new triangles and also to update themselves. The location and the direction of the newly created/updated edges are always stored ordered in the same way to prevent any confusion. i.e. after this sub-step, it is possible to know which part of the divided edge (left part, right part, and opposite parts in order) is in which memory location.

The *third sub-step* is the creation of new triangles (and updating the modified ones) using new edge indices. Each triangle is responsible for the creation of the new triangles inside it. So, each triangle can be assigned to one thread. Taking into account the number of edges of a triangle to be divided, and the direction of each edge, there are 18 different possibilities for forming new triangles. These are shown in Figure 4.4.

4.5. Example Run

A small example run is presented to illustrate how the edge and triangle data are updated step by step. This example has initially three triangles. The initial mesh is given in Figure 4.5, where the gray triangle is marked for refinement.

The initial edge and triangle arrays are shown in Tables 4.2 and 4.3 respectively. Note that the longest edge of the triangle to be refined (edge 3) is initially marked for refinement.

After step 2, the longest edges of each triangle is marked, and the edge array is

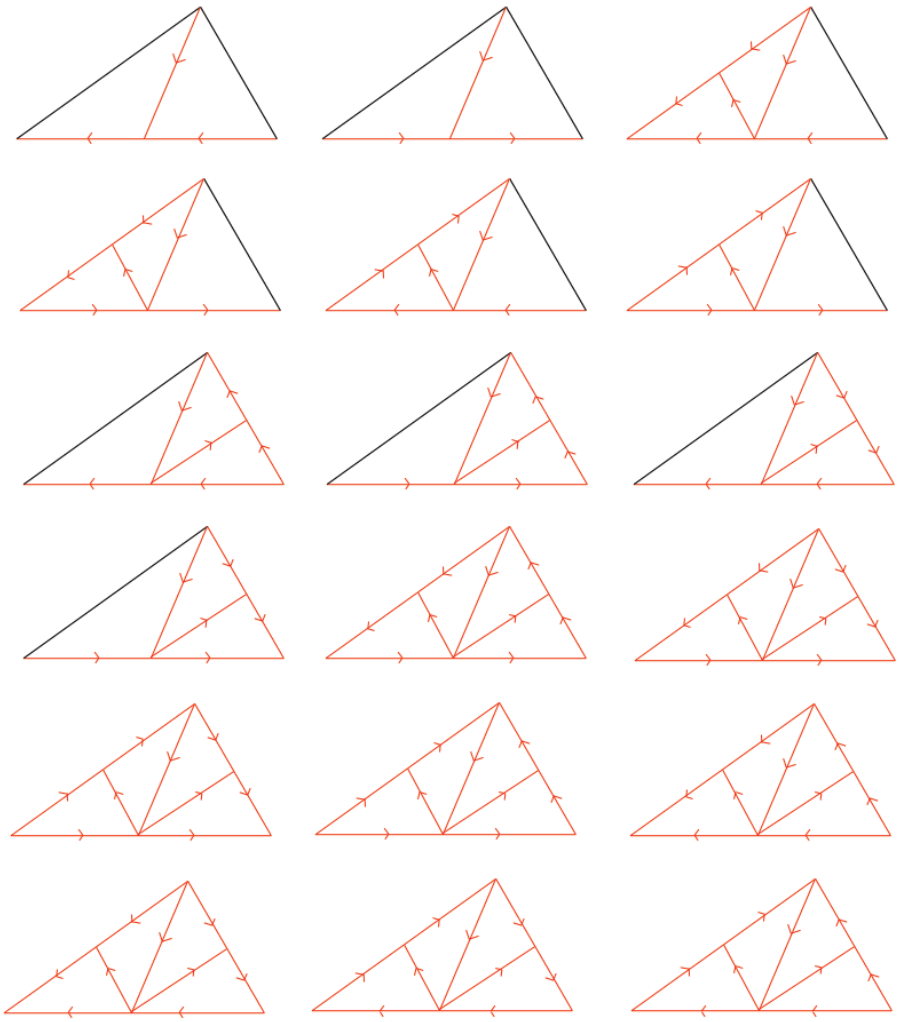


Figure 4.4. Triangle templates for all 18 cases of triangle division.

updating accordingly, as shown in Table 4.4.

After step 4, the dependency tree is created, and the division of edge 3 is propagated through edge 5, indicating that in the refinement phase, these two edges will be divided. The edge array after step 4 is shown in Table 4.5.

After the refinement phase, 6 new edges and 4 new triangles are created. Divided edges (edge 3 and edge 5) and triangles are also updated to form new elements. The final edge and triangle arrays after the operation is shown in Tables 4.6 and 4.7 respectively. The final mesh is shown in Figure 4.6.

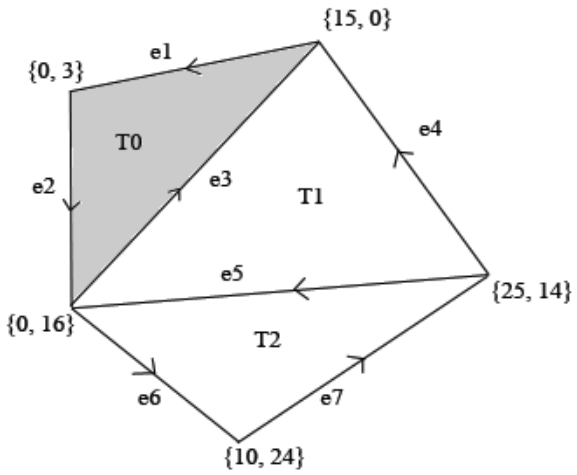


Figure 4.5. An example mesh to refine.

Table 4.2. Initial edge array for example mesh given in Figure 4.5.

	e1	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**
Point 0	{15, 0}	{0, 3}	{0, 16}	{25, 14}	{25, 14}	{0, 16}	{10, 24}
Point 1	{0, 3}	{0, 16}	{15, 0}	{15, 0}	{0, 16}	{10, 24}	{25, 14}
Mark/B.Mark	0/0	0/0	0/1	0/0	0/0	0/0	0/0

Table 4.3. Initial triangle array for example mesh given in Figure 4.5.

	T0	**T1**	**T2**
Edge0	1	2	3
Edge1	4	-3	-5
Edge2	5	6	7

Table 4.4. Edge array after step 2.

	e1	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**
Point 0	{15, 0}	{0, 3}	{0, 16}	{25, 14}	{25, 14}	{0, 16}	{10, 24}
Point 1	{0, 3}	{0, 16}	{15, 0}	{15, 0}	{0, 16}	{10, 24}	{25, 14}
Mark/B.Mark	0/0	0/0	1/1	0/0	1/0	0/0	0/0

Table 4.5. Edge array after step 4.

	e1	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**
Point 0	{15, 0}	{0, 3}	{0, 16}	{25, 14}	{25, 14}	{0, 16}	{10, 24}
Point 1	{0, 3}	{0, 16}	{15, 0}	{15, 0}	{0, 16}	{10, 24}	{25, 14}
Mark/B.Mark	0/0	0/0	1/1	0/0	1/1	0/0	0/0

Table 4.6. The final edge array after refinement.

	e1	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**
Point 0	{15, 0}	{0, 3}	{0, 3}	{25, 14}	{10, 24}	{0, 16}	{10, 24}
Point 1	{0, 3}	{0, 16}	{7.5, 8}	{15, 0}	{12.5, 15}	{10, 24}	{25, 14}
	e8	**e9**	**e10**	**e11**	**e12**	**e13**	
Point 0	{7.5, 8}	{0, 16}	{12.5, 15}	{15, 0}	{12.5, 15}	{25, 14}	
Point 1	{15, 0}	{7.5, 8}	{7.5, 8}	{12.5, 15}	{0, 16}	{12.5, 15}	

Table 4.7. The final triangle array after refinement.

	T0	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**
Edge0	-3	11	-5	3	10	-10	5
Edge1	2	-13	7	8	-9	-11	12
Edge2	9	4	13	1	-12	-8	6

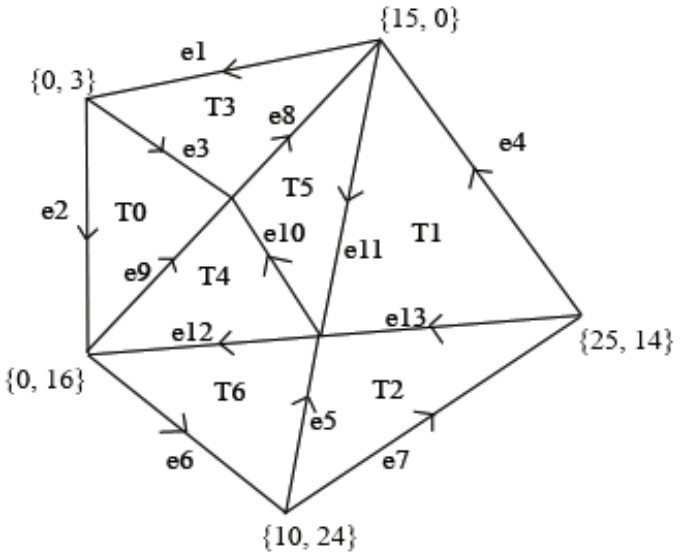


Figure 4.6. Final mesh, after refinement of the mesh shown in Figure 4.5.

5. TESTS AND RESULTS

Our algorithm is massively data parallel. Since the algorithm is not CUDA dependent, it can also be implemented in OpenMP or MPI. But since the problem itself is memory intensive rather than computationally intensive, speed-up that can be achieved is determined by memory bandwidth. The same algorithm is implemented to run sequentially on PC and parallel on CUDA architecture for benchmarking.

The implementation is done using the C programming language and CUDA C library. CUDA Runtime Library version 3.2 is used. For the prefix operation used in algorithm step 6, the well-known CUDPP library [28] is used.

Our algorithm is designed and implemented for 2-D meshes, but can also be extended to 3-D meshes.

5.1. Test Environment

The tests are run in two different cluster configuration: First one has two Intel Xeon X5550 CPUs and four Tesla C1060 GPU units. The second one is the Amazon Cluster GPU instance [29] having two Intel Xeon X5570 CPU and two Tesla M2050 GPU. Running the tests on publicly available Amazon Clusters makes it possible for everyone to easily reproduce the tests on a fixed environment. The technical specifications of each item is shown in Table 5.1.

Although both configuration have multiple CPUs and GPUs our implementation uses only one of the CPU cores and only one GPU card. The reason for CPU is obvious, since our implementation on CPU is sequential. The reason for using a single GPU is that each GPU unit has its own memory and accessing each others‘ memory directly is impossible for C1060 and possible but expensive in M2050. And also, distributing memory into GPUs is beyond the scope of this work. But since

Table 5.1. Specifications of test hardware.

	Intel Xeon X5550	**Intel Xeon X5570**
# of Cores	4	4
# of Threads	8	8
Clock Speed	2.66 GHz	2.93 GHz
Cache	8 MB	8 MB
	NVIDIA Tesla C1060	**NVIDIA Tesla M2050**
Compute Capability	1.3	2.0
Core Clock	1.3 GHz	1.15 GHz
# of Stream Processors	240	448
Available Memory	4 GB	3 GB (ECC off)
Peak Memory Bandwidth	102 GB/s	148.4 GB/s

in real implementations, the entire mesh is divided into parts and each is handled by different units, all GPUs may be utilized. Another reason is that, the problem is memory intensive problem rather than computationally intensive, so using more computation units will not change the running time unless they provide additional memory bandwidth.

Tesla M2050 has a configurable ECC option which is able to fix single-bit errors and report double-bit errors. But enabling ECC causes some of the memory to be used for the ECC bits resulting in available memory to decrease by 12.5%. It also decreases the memory bandwidth. So, this option is disabled in the tests. Tesla M2050 has the same on-chip memory that is used for both L1 cache and shared memory: It can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 KB of L1 cache. In our experiments it is configured in the latter way, which results in better timings for our algorithm.

Tests were run for different number of input triangles ranging from 100k to 30-40m, upper limit depending on the maximum amount of memory the GPU device has. For generating input meshes, Triangle application by J. R. Shewchuk [27] is used.

The number of input triangles marked for refinement is 0.1% of the total number of triangles, and uniformly distributed over all mesh in the following tests.

All computations are done in single precision. For single precision, internal data structures require 44 bytes for each input edge, and 32 bytes for each input triangle. Considering the fact that for n input triangles, there are approximately 3/2n input edges, we can be able to run the algorithm for up to 30 million triangles on the Tesla M2050 which has 3 GB memory, and for up to 42 million triangles on the Tesla C1060 which has 4 GB memory.

5.2. CPU-vs-GPU Runtime Analysis

Results show that, the algorithm runs best on the Tesla M2050 GPU. For 30m triangles, running time is approximately 40 times better than the sequential run on Xeon X5570. Running time details for different input sizes are given in Tables 5.2, 5.3, 5.4 and 5.5, for each step. The timings for steps other than refinement step is only given for reference, they are not included in the speed-up calculation. Refinement times for different input sizes are compared in Figure 5.1. The speed-up by the number of triangles between Tesla M2050 and Intel X5570 is shown in 5.2

Memory transfer time from host to GPU is also an important factor, because in real implementations, we need to copy the input to GPU memory before the computation and copy back the results to host after the computation. The memory transfer times for different input sizes is given in Table 5.6 for reference.

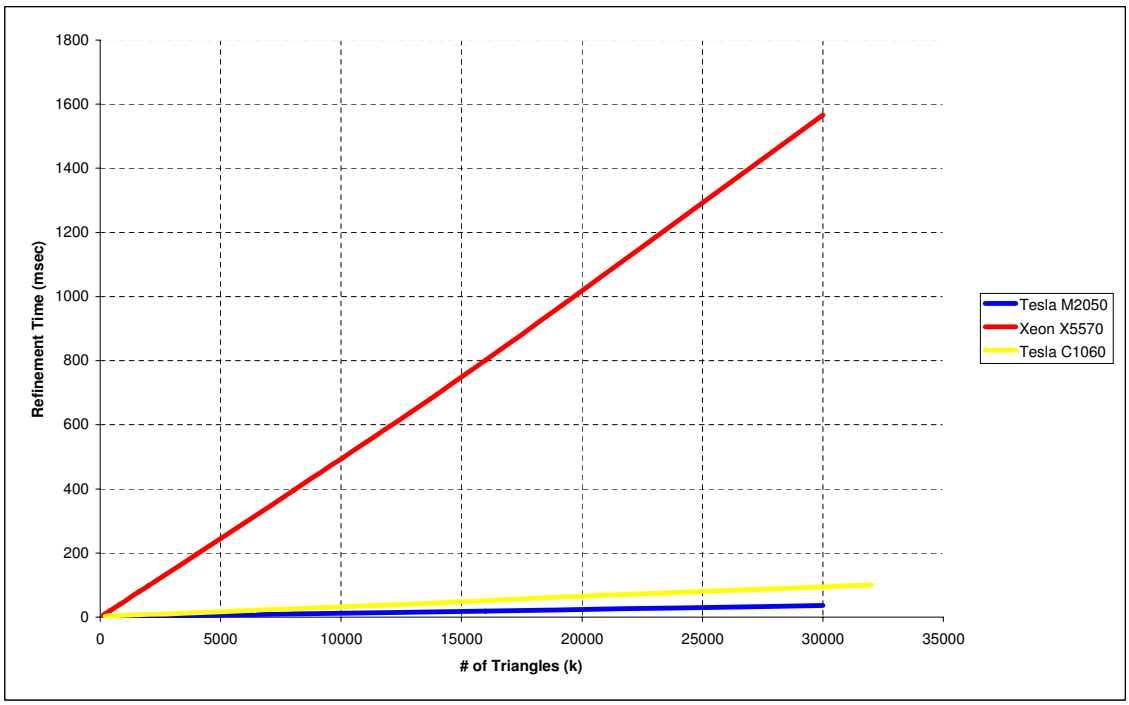


Figure 5.1. Refinement time comparison for Tesla M2050, Intel X5550 and Intel X5570.

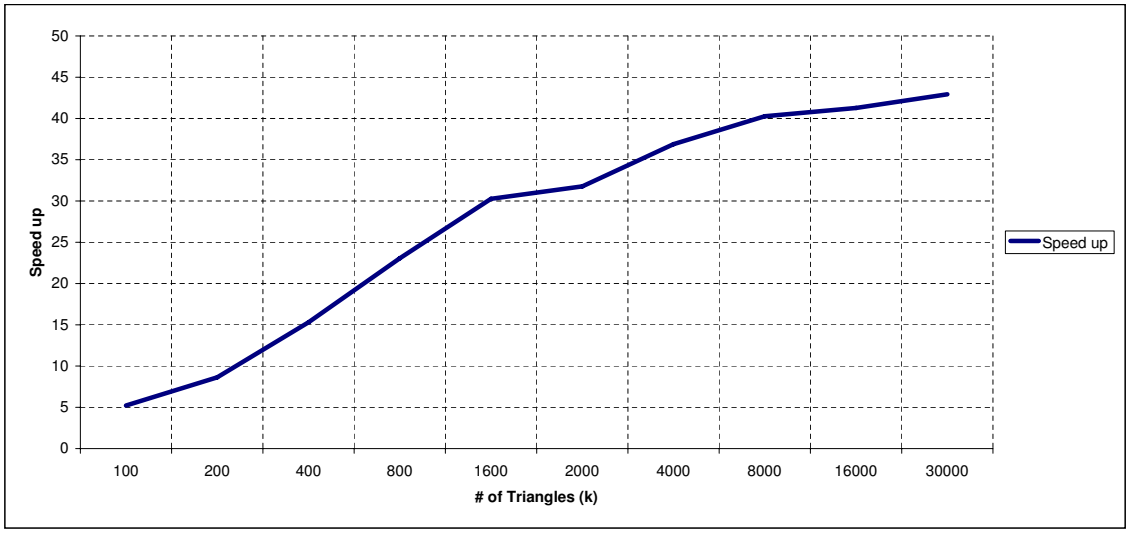


Figure 5.2. Speed-up by the number of Triangles between Tesla M2050 and Intel X5570.

Table 5.2. Detailed timings for each step for Tesla M2050 (in ms).

# of Triangles	Step1	Step2	Step3	Step4	Step5	Step6	Refinement
100k	0.4	0.3	0.3	0.5	0.3	1	0.9
200k	0.5	0.3	0.3	0.5	0.3	1.1	1.1
400k	0.7	0.4	0.4	0.5	0.4	1.6	1.3
800k	1	0.6	0.5	0.6	0.5	1.2	1.7
1.6m	1.7	0.9	0.8	0.7	0.7	2.3	2.6
2m	2.1	1	1	0.7	0.8	2.4	3.1
4m	3.9	1.7	1.7	1	1.4	2.9	5.3
8m	7.6	3.2	3.3	2	2.6	4	9.8
16m	15	6.4	6.5	3.5	5.2	6.3	19.4
30m	28.1	12	12.2	6.8	9.8	10.1	36.5

Table 5.3. Detailed timings for each step for Tesla C1060 (in ms).

# of Triangles	Step1	Step2	Step3	Step4	Step5	Step6	Refinement
200k	0.9	0.8	0.8	1.3	0.8	0.7	3
400k	1	1	1	1.4	0.9	0.8	3.5
800k	1.4	1.4	1.3	1.4	1.2	1	4.7
1.6m	2.1	2.2	2.1	1.6	1.9	1.7	7.2
3.2m	3.4	3.8	3.5	1.9	3.2	2.4	11.9
6.4m	6	6.9	6.4	2.4	5.7	3.7	21.4
12.8m	11.9	13.3	12.3	4.8	10.9	6.3	40.5
20m	17.8	21	19.5	6.2	17.3	9.3	64.7
32m	28.3	33	30.5	9.4	27	13.9	101.1

Table 5.4. Detailed timings for each step for Intel X5570 (in ms).

# of Triangles	Step1	Step2	Step3	Step4	Step5	Step6	Refinement
100k	2.1	1.8	1.9	0.1	0.5	0.2	4.7
200k	4.3	3.5	3.8	0.2	1	0.4	9.5
400k	9.4	7.2	7.7	0.5	2.2	0.9	19.9
800k	16.9	14.1	15.8	1	4.4	1.9	39.2
1.6m	34.6	28.3	31	2.1	9	3.7	78.7
2m	36.5	35.4	38.5	2.6	11.2	4.7	98.5
4m	61.4	70.9	76.7	5.2	22.2	9.5	195.5
8m	118.6	142.9	153	10.4	47.2	23.3	394.6
16m	238.4	289.3	308.7	20.7	96.3	47.1	800.8
30m	433.6	542.4	580.2	39	187.2	87	1566.8

Table 5.5. Detailed timings for each step for Intel X5550 (in ms).

# of Triangles	Step1	Step2	Step3	Step4	Step5	Step6	Refinement
100k	2.1	1.8	3.5	0.3	1	0.4	9.4
200k	4.3	6.1	6.9	0.6	2.2	0.8	19.1
400k	9.4	12.3	14	1.2	4.2	1.6	32
800k	16.9	24.7	28.1	2.4	8.5	3.3	54.5
1.6m	34.6	42.9	45.3	4.8	17	6.6	99.9
2m	36.5	48.3	49.4	6	21.2	8.4	116.3
4m	61.4	89.4	93	12	30.3	16.3	243.3
8m	118.6	153.2	175.4	25.9	60.3	43.2	484.8
16m	238.4	312.7	350.5	42.4	115.9	62	955.8
32m	433.6	605.7	684.2	69.5	212.4	109.4	1866.2
42m	567.3	786.1	891.2	87.5	277.2	139.4	2462.3

Table 5.6. Memory transfer time from host to Tesla M2050.

# of Triangles	Total Bytes (MB)	Total Time (ms)
100k	9.3	2.323
200k	18.7	4.114
400k	37.4	6.665
800k	74.8	23.237
1.6m	149.5	12.394
2m	186.9	28.356
4m	373.8	55.341
8m	747.7	109.060
16m	1495.4	216.710
30m	2803.8	405.953

6. CONCLUSION

By testing the same algorithm in different environments and on different hardware, it is observed that since the algorithm is massively parallel, the refinement problem can be characterized as being memory intensive, rather than computationally intensive (i.e. the speed-up is related to the memory bandwidth and not the clock speed or FLOPS or the number of threads). Today's GPU architectures have much better memory bandwidth especially if the memory accesses are coalesced. So, the hard part of GPU programming involves regular data access patterns. With evolution of NVIDIA GPU hardware from compute capability 1.x to 2.x, (i.e. the production of Tesla M2050 series) the memory bandwidth has been increase significantly by presenting a 2-level cache on top of the global memory. It is also much easier to coalesce data accesses when a large cache is present, which eliminates the need of ordered accesses by the threads (but memory accesses should be still aligned). This makes it easier to develop general-purpose computation on GPUs by letting us obtain better speed-ups than before. In our case, our algorithm runs approximately twice faster for 30m elements on the Tesla M2050, compared to the Tesla C1060.

Mesh refinement routine gets input from a finite-element solver, which actually decides which triangles are to be refined. Using a solver implemented on a GPU along with our refinement algorithm will provide more advantage since the data does not have to be moved between the CPU and the GPU back and forth, which is costly and usually costs more than the entire process of refinement.

6.1. Future Work

Our algorithm is designed and implemented for 2-D meshes, but it may be extended to 3-D meshes which is left as a future work. The algorithm is implemented for a single GPU, which can be also extended to run on shared memory multiple GPU platforms.

Our algorithm can also be implemented on other parallel architectures such as Cell-BE or OpenMP.

APPENDIX A: SOME EXAMPLES OF REFINED MESHES

Figure A.1 shows an initial mesh with 100k triangles. Figure A.2 shows the final mesh which the triangles on the lower left corner are refined. Figure A.3 shows the final mesh which the refined triangles are uniformly distributed and randomly selected over all mesh. Figure A.4 shows the final mesh where all the triangles in the mesh are refined.

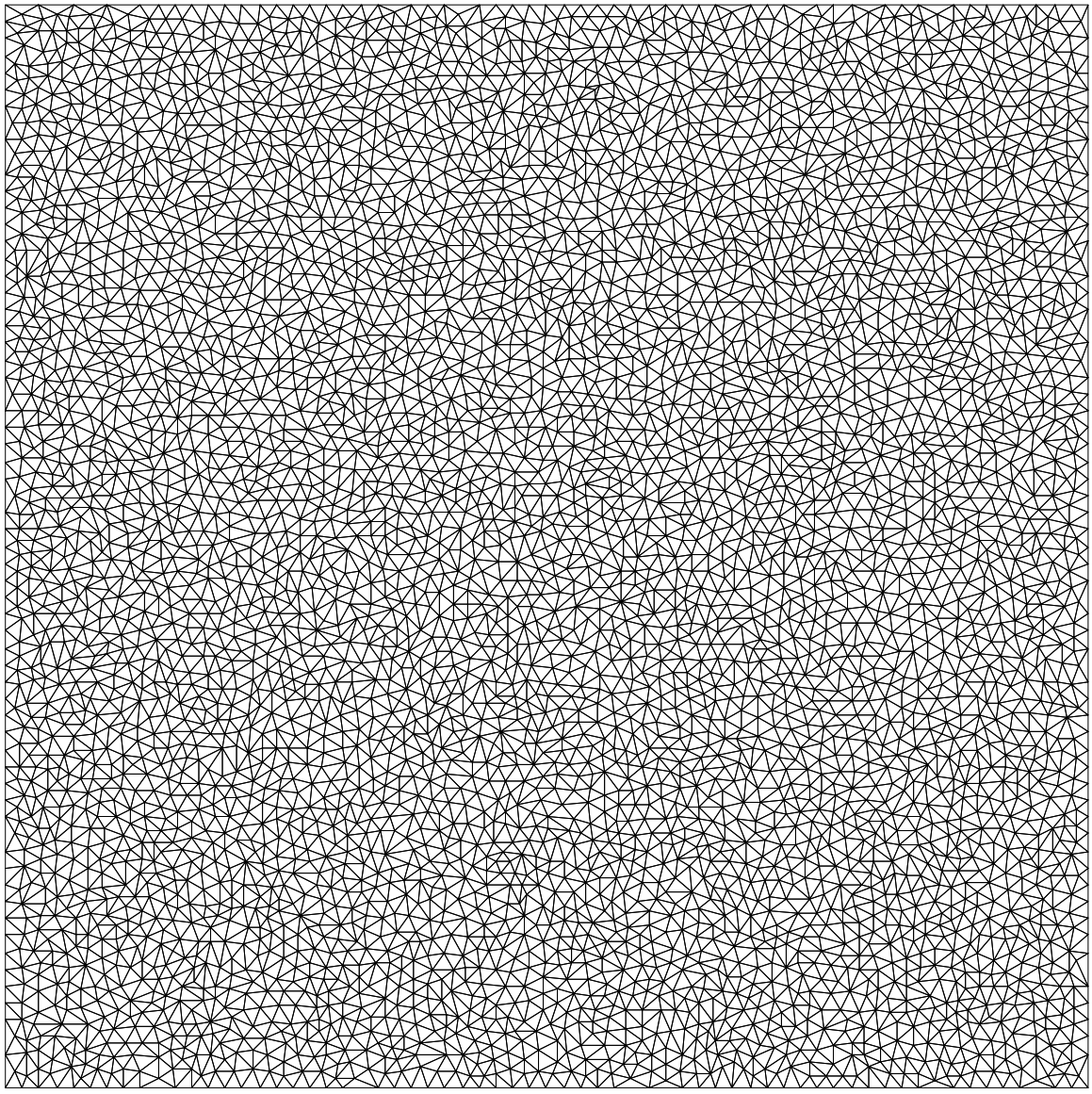


Figure A.1. An example mesh having 100k triangles.

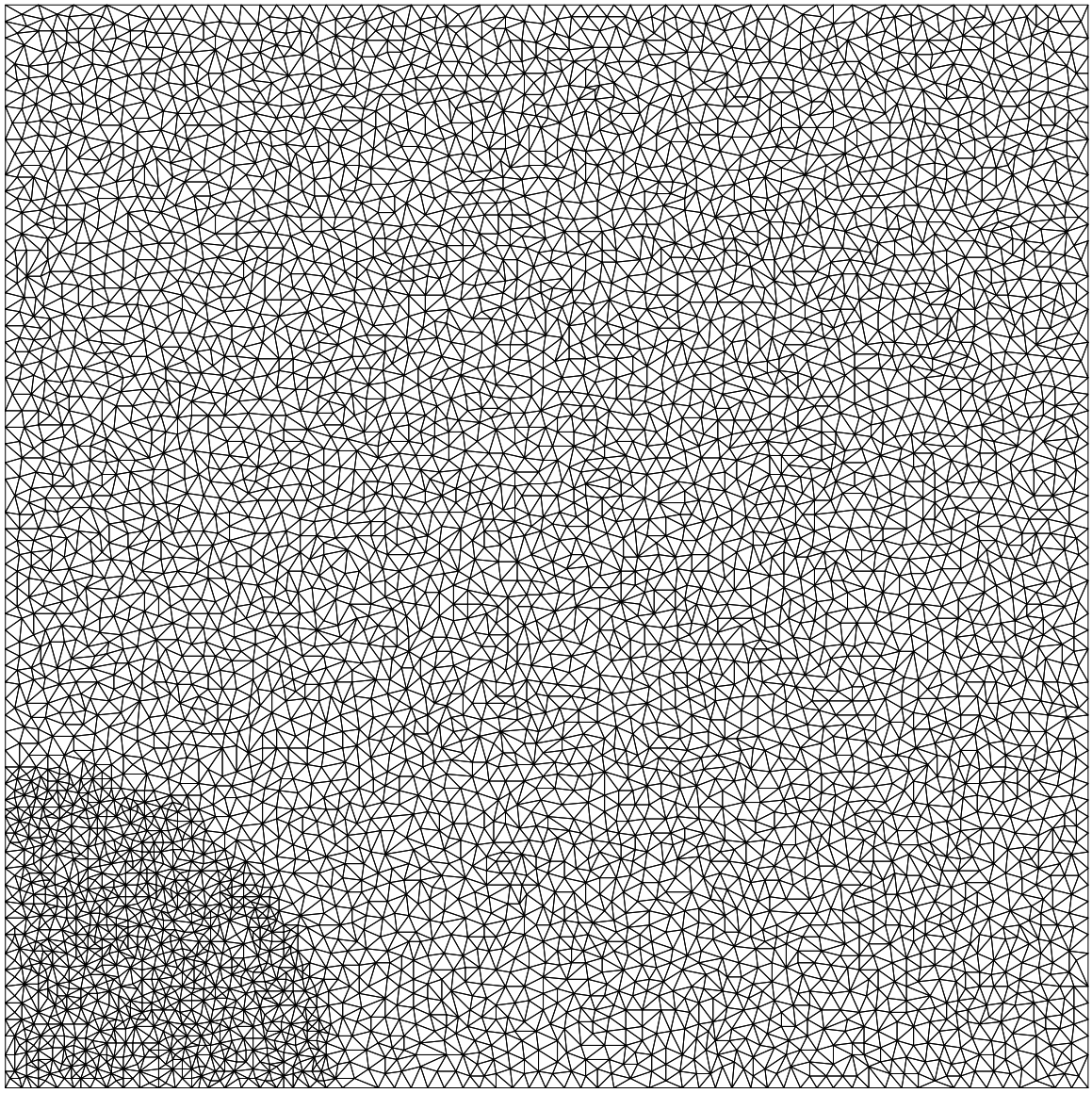


Figure A.2. The triangles in the lower left corner of the mesh in Figure A.1 are refined.

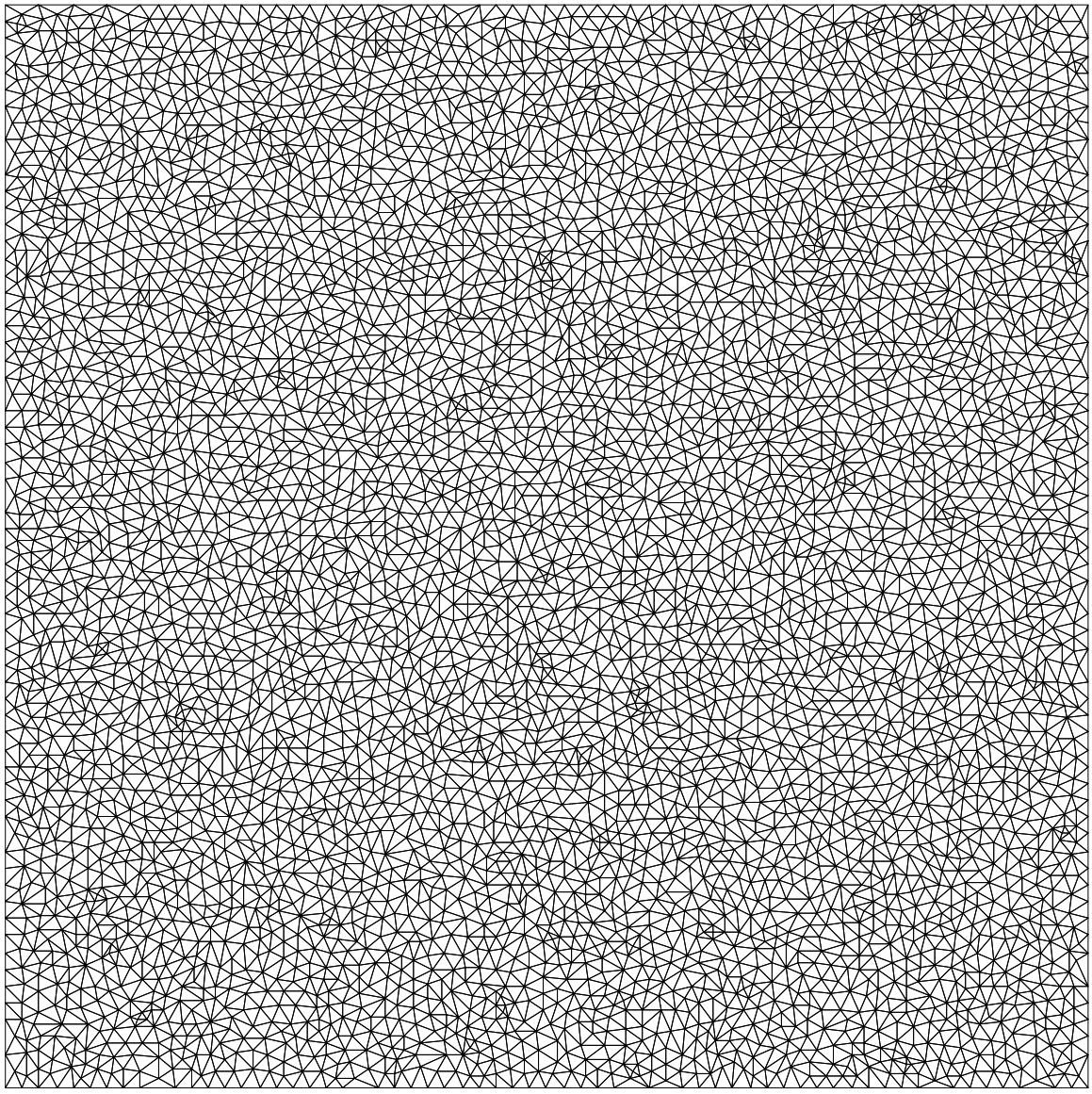


Figure A.3. Refined triangles are randomly distributed.

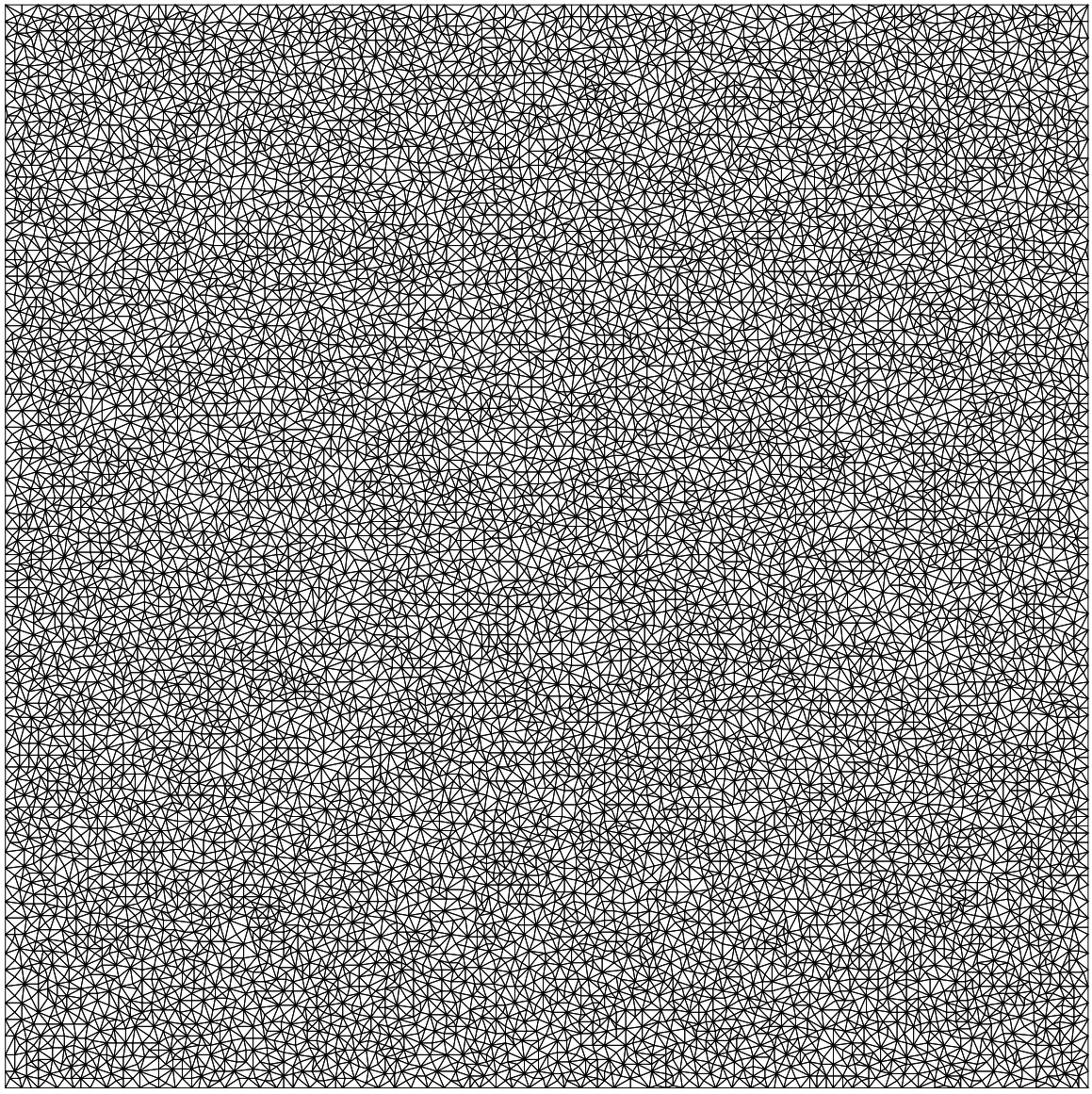


Figure A.4. All triangles of the mesh in Figure A.1 are refined.

APPENDIX B: DESCRIPTION OF THE SOURCE CODE

The source code of the example implementation is published in Google code project called *gpu-mesh-refine* with GPL license and can be found in [30]. The details of the implementation and usage are described below.

B.1. Input File Format

A custom input file format is used as the input mesh structure to the application. The input file is a text file consisting of three sections: *Points*, *Edges* and *Triangles*. In the *Points* section, there exists 2-D coordinates of the edge points. Each edge consists of two points. The *Edges* section contains the edges having two point indexes each. Each triangle consists of three edges. The *Triangles* section contains the edge indexes of each triangle.

An example input file having the example input used in Chapter 4.5 is shown in Figure B.1.

Note that blank lines and lines starting with # comment symbol are skipped while reading input.

The first line of each section is an integer indicating the number of entities in that section. The following lines in the *Points* section has two floating point numbers which are the point coordinates in x and y directions respectively, starting from the 0th point. Starting from the 1st edge, the following lines in the *Edges* section has two integer numbers which are the point indexes in the order of the input points. Starting from the 1st triangle, the following lines in the *Triangles* section has 4 integer numbers: First three numbers are the directional edge indexes, having either a negative value indicating the edge is in the reverse direction of the triangle, or a positive value indicating the edge is in the forward direction. The last number which is ither 1 or 0,

indicates if the triangle is initially marked for refinement or not respectively.

B.2. Output File Format

Output file is similar to the input file except it does not have *Points* section. Instead, it has the point coordinates of each edge in the *Edges* section.

An example output file belongs to the example used in Chapter 4.5 is shown in Figure B.2.

B.3. Source Code Snippet

The source code snippet of the main() function is given in Figures B.3 and B.4 for reference.

```
#points
5
0 3
15 0
0 16
25 14
10 24

#edges
7
1 0
0 2
2 1
3 1
3 2
2 4
4 3

#triangles
3
1 2 3 1
4 -3 -5 0
5 6 7 0

#end
```

Figure B.1. Example input file.

```
#edges
13
[15  0]   [0  3]
[0  3]    [0  16]
[0  3]    [7.5  8]
[25  14]          [15  0]
[10  24]          [12.5  15]
[0  16]   [10  24]
[10  24]          [25  14]
[7.5  8]          [15  0]
[0  16]   [7.5  8]
[12.5  15]        [7.5  8]
[15  0]   [12.5  15]
[12.5  15]        [0  16]
[25  14]          [12.5  15]

#triangles
7
-3        2         9         0
11        -13       4         0
-5        7         13        0
3         8         1         0
10        -9        -12       0
-10       -11       -8        0
5         12        6         0
```

Figure B.2. Example output file.

```
int main(int argc, char** argv)
{
        // parse arguments
        ...
        // input check
        ...
        // read input file
        ...
        // validate input
        ...
        printf("Starting process...\n");

        malloc_copy_input_to_device();

        // step 1: calculate edge lengths and mid points
        calc_edge_lengths_mid_p();

        // step 2: mark longest edge of each triangle
        mark_longest_edges();

        // step 3: establish links
        establish_links();

        // step 4: follow links
        follow_links();

        // step 5: get the initial value of counters
        get_counters();
```

Figure B.3. Code snippet of main() function.

```
        // step 6: prefix the counters
        prefix_counters();

        create_new_elem_arrs();

        // step 7: refine the mesh: create new edges
        refine();

        // OKI DOKI! copy back the data to host
        copy_back();

        // free the device stuff
        device_cleanup();

        // generate output
        ...
}
```

Figure B.4. Code snippet of main() function.

REFERENCES

1. Heuveline, V., C. Subramanian, D. Lukarski and J. Weiss, "A multi-platform linear algebra toolbox for finite element solvers on heterogeneous clusters", *IEEE International Conference on Cluster Computing*, 2010.

2. Mitchell, W., "A comparison of adaptive refinement techniques for elliptic problems", *ACM Transactions on Mathematical Software*, Vol. 15, No. 4, pp. 326–347, 1989.

3. Jones, M. and P. Plassmann, "Parallel algorithms for adaptive mesh refinement", *SIAM Journal on Scientific Computing*, Vol. 18, No. 3, pp. 686–708, 1997.

4. Nambiar, R., R. Valera and K. Lawrence, "An Algorithm for Adaptive Refinement of Triangular Element Meshes", *International journal for numerical methods in engineering*, Vol. 36, No. 3, pp. 449–509, 1993.

5. Rivara, M., "Algorithms for refining triangular grids suitable for adaptive and multigrid techniques", *International Journal for Numerical Methods in Engineering*, Vol. 20, No. 1, pp. 745–756, 1984.

6. Bank, R., A. Sherman and A. Weiser, "Some refinement algorithms and data structures for regular local mesh refinement", *IMACS Transactions on Scientific Computing*, Vol. 1, No. 1, pp. 3–17, 1982.

7. Jones, M. and P. Plassmann, "Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes", *Proceedings of the Scalable High-Performance Computing Conference*, 1994.

8. Babuska, I. and K. Aziz, "On the angle condition in the finite element method", *SIAM Journal of Numerical Analysis*, Vol. 13, No. 2, pp. 214–226, 1976.

9. Fried, I., "Condition of finite element matrices generated from non-uniform meshes", *AIAA Journal*, Vol. 10, No. 2, pp. 219–221, 1972.

10. Ozturan, C., "Worst Case Complexity of Parallel Triangular Mesh Refinement by Longest Edge Bisection", *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

11. Bansch, E., "An adaptive finite-element strategy for the three-dimensional time-dependent navier-stokes equations", *Journal of Computational and Applied Mathematics*, Vol. 36, No. 1, pp. 3–28, 1991.

12. Rivara, M., "Mesh refinement processes based on the generalized bisection of simplices", *SIAM Journal of Numerical Analysis*, Vol. 21, No. 3, pp. 604–613, 1984.

13. Castanos, J., *The dynamic adaptation of parallel mesh-based computation*, M.S. Thesis, Brown University, 1996.

14. Shephard, M., J. Flaherty, H. DeCougny, C. Ozturan, L. Bottasso and M. Beall, "Parallel automated adaptive procedures for unstructured meshes", *Parallel Computing in CFD, AGARD, Neuilly-Sur-Seine*, 1995.

15. Williams, R., *A dynamic solution-adaptive unstructured parallel solver*, Tech. rep., Supercomputing Facility, California Institute of Technology, California, 1992.

16. Gšoddeken, D., S. Buijssen, H. Wobker and S. Turek, "GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver", *International Conference on High Performance Computing*, 2009.

17. Khailany, B., W. Dally, S. Rixner, U. Kapasi, J. Owens and B. Towles, "Exploring the VLSI Scalability of Stream Processors", *The Ninth Symposium on High Performance Computer Architecture*, 2003.

18. Association, S. I., *The International Technology Roadmap for Semiconductors*, Tech. rep., 2003.

19. Ujaldon, M. and J. Saltz, "The GPU on irregular computing: performance issues and contributions", *Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics*, 2005.

20. *A Web page dedicated to the latest developments in general-purpose on the GPU*, `www.gpgpu.org`, Accessed at June 2011.

21. Owens, J., D. Luebke, N. Govindaraju, M. Harris, J. Kršuger, A. Lefohn and T. Purcell, "A survey of general-purpose computation on graphics hardware", *Computer Graphics Forum*, Vol. 26, No. 1, pp. 80–113, 2007.

22. Owens, J., M. Houston, D. Luebke, J. S. S. Green and J. Phillips, "GPU computing", *Proceedings of the IEEE*, Vol. 96, No. 5, pp. 879–899, 2008.

23. *CUDA C Programming Guide*, `http://developer.download.nvidia.com`, Accessed at June 2011.

24. Lindholm, E., J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", *IEEE Micro Journal*, Vol. 28, No. 2, pp. 39–55, 2008.

25. Taylor, Z., M. Cheng and S. Ourselin, "High-Speed Nonlinear Finite Element Analysis for Surgical Simulation Using Graphics Processing Units", *IEEE Transactions on Medical Imaging*, Vol. 27, No. 5, pp. 650–663, 2008.

26. Chen, W. and H. Hang, "H.264/AVC motion estimation implementation on Compute Unified Device Architecture (CUDA)", *IEEE International Conference on Multimedia and Expo*, 2008.

27. *A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*,

http://www.cs.cmu.edu/ quake/triangle.html, Accessed at June 2011.

28. *CUDA Data Parallel Primitives Library (CUDPP)*, http://gpgpu.org/developer/cudpp, Accessed at June 2011.

29. *Amazon EC2, Cluster GPU Instance*, http://aws.amazon.com/ec2/#instance, Accessed at June 2011.

30. *Parallel Triangular Mesh Refinement by Longest Edge Bisection on CUDA (Project Homepage)*, http://code.google.com/p/gpu-mesh-refine, Accessed at August 2011.