GPU Computing and CUDA technology used to accelerate a mesh generator application

Adriana Gaudiani[1], **Santiago Montiel**[1], **and Javier Pimás**[1]
[1]Instituto de Ciencias, Universidad Nacional de General Sarmiento
San Miguel, Buenos Aires, Argentina

Abstract— *The potential of GPU computing used in general purpose parallel programming has been amply shown. These massively parallel many-core multiprocessors are available to any users in every PCs, notebook, game console or workstation. In this work, we present the parallel version of a mesh-generating algorithm and its execution time reduction by using off-the-shelf GPU technology. We use commodities GPUs as a useful CPU co-processor to improve this kind of applications, characterized by a high level of data parallelism. Compared to the sequential algorithm, our techniques achieve 6X overall performance for GPU-CPU implementation; furthermore we achieve 50X speedup when implementing core operations of the algorithm. Results show that GPU provides a helpful platform for high performance computing to improve the execution time of these applications.*

Keywords: Parallel algorithms, Multicore processors, Graphics processing units.

1. Introduction

In the past few years, Graphics Processing Units Computing (*GPUs*) has demonstrated to provide an increased performance computing architecture for applications which can be written to take advantage of many core GPUs. The idea to use the GPU for general purpose computations starts about 2003. However, it was reserved only for specialist developers in graphic rendering. It was possible since 2007, when NVIDIA released the software technology called Compute Unified Device Architecture (CUDA) simultaneously with its TESLA Architecture [1] [2]. Since then, GPUs have increased in capability and programmability and have gained wide popularity among research community [3]. Owens et al. widely explain why GPUs increased faster than CPUs. Furthermore, this technology is available, inexpensive and can be found in off-the-shelf graphics cards for PCs. CUDA is designed for extended standard C/C++ code with GPUs parallel features and it provides a unified computing platform to take advantage of the GPUs power and to leverage general purpose parallel applications [4].

In this paper, we present as we used cutting-edge computing resources, such as GPUs multicores and CUDA, to accelerate the execution time of a finite element mesh generator algorithm, achieving a significant parallel speedup.

Mesh generation is a key step in many scientific computations and computer graphics. It had its origin in the 50's with structural analysis problems. Finite element is a numerical method used to solve partial differential equations approximately; whose first step is mesh generation.

We use GPUs as a floating point parallel CPU coprocessor to improve the mesh generation algorithm *Distmesh*, created by P. Persson and G. Strang. Distmesh authors wrote an efficient Matlab algorithm to provide a simple code to produce high quality meshes. We wrote a parallel version for this algorithm to introduce an interesting general purpose application for GPU computing.

Organization: The paper is structured as follows. In section 2 we present a general view of Persson-Strang mesh generator algorithm and an overview of architectural features of GPUs. In section 3 we present our sequential version of Distmesh algorithm and highlight its features. Section 4 describes the design and implementation of our GPU version of mesh generator algorithm. Section 5 gives experimental results. In section 6 we briefly introduce related works and section 7 gives conclusions.

2. Background

In this section we present important background concepts, relevant to this paper. First, we present a briefly description of Persson-Strang Algorithm. Then, we outline some important issues for using GPU.

2.1 Persson-Strang Algorithm

Per-Olof Persson and Gilbert Strang developed a simple and public mesh generator code for Matlab, called Distmesh [5]. They offer an iterative technique based on a physical analogy between a simple mesh and a trust structure, combining a signed distance function and forces movement at each node. The results obtained are high quality meshes.

Many problems are defined on irregularly shaped domains, so unstructured meshes, far better than structured meshes, can be flexibly tailored to the physics of these problems. The problem that arises is the complex, and nearly inaccessible, meshing software code. We have chosen Persson and Strang algorithm because of its simplicity and accuracy. We present below a brief description of the algorithm.

Initial nodes position may be chosen by equally spaced distribution, and this works well for simple geometries. The user can define a function *h* to set mesh resolution. This function *h(x,y)*, in 2D, is used to refine complex geometry mesh and thus achieve geometrical adaptivity; it needs to be resolved by small elements. The meshpoints define the truss structure and a Delaunay triangulation algorithm determines the topology. Delaunay method set triangles in 2D, or tetrahedra in 3D, to fill the convex hull of the input domain mesh points [6]. In response to the mechanical analogy, triangle edges correspond to bars (or springs) and mesh points correspond to truss joints. The numerical method assumes that a displacement force is exerted on the bars. In every iteration, the new location of the points is obtained by calculating a force of static equilibrium. Delaunay triangulation is needed whenever the points are separated far; thus adjust the topology. For a very detailed explanation on the Persson method, the interested reader should consult [7].

2.2 GPU and CUDA overview

GPU is a massively multi-threaded multiprocessor architecture and its data level concurrency stand out. The threads are organized in two-level hierarchy. The lower level is a *block*, which contains a large number of threads. The higher level is a *grid*, which consists of a group of blocks. The maximum dimension of blocks and grids is determined by the GPU architecture. Parallel threads share memory and synchronize using barriers [8], [9].

The key to effectively using GPU is to understand its memory hierarchy, which consists of three levels of memory. Programmers can explicitly manage data stored in them. *Device memory* is the global GPU memory which is accessible from all the threads. *Shared memory* is an on-chip memory. It's a low latency memory shared by all the threads within a block. *Texture and constant memory* are used to store explicitly declared read only data.[10].

NVIDIA's CUDA enables to divide the parallel program execution in tasks that can run across thousands of concurrent threads, over hundreds of processor cores. This programming model is known as a Single-Program Multiple-Data (SPMD) and it allows to program GPUs for general purpose. Tesla GPU, the NVIDIA device used for our experiences, manage efficiently a huge sum of threads employing a *Single-Instruction Multiple-Thread* (SIMT) parallel programming architecture. The task performed by every thread is managed writing special functions, called *kernels*. The kernel task is mapped over a set of threads, representing the work to be done at a simple point in the domain. We wrote the kernels using C and CUDA, an extended version of C; in this way, we mapped the kernels on the GPU manycore processors. For a more detailed description of CUDA, GPU architecture and Tesla architecture, you can refer to [10][11][12].

3. General features of sequential version

We first implemented a C++ sequential code of Distmesh algorithm. We took into account some important items in the original version of the mesh generator algorithm, as we describe below.

- A distance function *d* determines the domain geometry by means of a signed distance, which is negative inside the region. It was an essential decision, as authors remark. This function is calculated at a meshpoint set and also for calculating nodes distance to the closest boundary point.
- This implementation uses a linear function for repulsive forces, but it does not allow attractive forces.

$$f(l, l_0) = \begin{cases} k(l - l_0) & \text{if } l < l_0 \\ 0 & \text{if } l \geq l_0 \end{cases}$$

- The resultant force *FTot* is the sum of all force vectors meeting at a mesh node. Each bar exerts a force $f(l, l_0)$ depending on its actual length l and its relax length l_0.
- The relax length l_0 is constant for uniform meshes and it's required $f = 0$ for $l = l_0$. Distmesh authors choose l_0 slightly larger than the length desired -20% is a good rate- This calculation depends on the total sum of bars length.
- The time step for Euler method is Δt parameter. The parameter *geps* is used to calculate the tolerance in geometry evaluations, and it's used to decide whether a point is outside.
- All points going outside the domain during the update, p_n to p_{n+1}, are projected back to the boundary. The numerical gradient of *d* gives the direction of the point movement.

The following are the key steps of our sequential code. In general terms, these steps correspond to those of Distmesh. Although, we modify the original data structures for better performance of the sequential algorithm. We will refer to this later.

Data Input: As a first step, we create a uniform distribution of mesh points within the input desired geometry. These points are the mesh-nodes. The resulting mesh points are regularly placed at a distance h_0 from their closest neighbors.

Triangulation: An important step in this algorithm is Delaunay triangulation. At every iteration, we compare the actual points positions with that of the previous triangulation. When the maximum displacement is greater than a predefined tolerance, a Delaunay retriangulation determines the new meshpoints set replacing the old ones, in order to guarantee Delaunay properties.

Update: The bars lengths are used to calculate the bar forces components. The resultant node force is the sum of the force vectors, from all bars meeting at a node. This result contributes to update node positions.

Projection: The update process may place some points outside the geometry. Once these points are found, they are projected back to the boundary, in response to a normal force. We use the numerical gradient of the distance function to calculate their move direction to the closest boundary point.

Our first CPU sequential code was a translation of Matlab Distmesh algorithm. Distmesh is an efficient algorithm for Matlab, it's completely vectorized to avoid loops. The authors use a sparse matrix to compute mesh-nodes movement. The sparse matrix dimension is determined by total points (n) and bars (m) of the mesh. Distmesh code uses a Delaunay Matlab function in order to determine truss topology. We selected the Delaunay C-code written by Geoff Leach for triangulation step, an open-source program. The author improved the divide and conquer Guibas-Stolfi algorithm and he got a factor of 4-5 speedup. This is a $O(nlog(n))$ algorithm [13].

The update step move the points to the new position, using the scalar force calculated for each bar. This is the main action taken by the algorithm and this is carried out using a large matrix of movements. Every matrix element ($M_{i,j}$) stores the movement of the i^{th} point which is one end of the k^{th} bar. The total move for n_i node is $P_i = \sum_{j=0}^{j=n-1}(M_{i,j})$. However, only a few bars converge at each mesh point.

This serial version is an easily implemented way to guarantee a correct execution and to facilitate the writing of a correct parallel version.

4. Our GPU-based Algorithm

As argued, Distmesh algorithm is highly suitable for GPUs architectures. Most of the operations, performed by the sequential mesh algorithm, were easily mapped on GPUs multiprocessors. The parallelism in Distmesh code is exploited by dividing the vector operations among the threads. Distmesh loop iterations are distributed to kernel blocks, so each data is fetched by a thread and every thread executes the same kernel.

Figure 1 presents a high-level overview of our parallel GPU-version of Distmesh algorithm. It outlines where the GPU acts as a parallel CPU co-processor in a collaborative way. The initial phase is executed on the CPU. CPU generates the first triangulation and copies points and bars from host (RAM) to device (GPU global memory). CPU launches the GPU kernels function to start the GPU mesh generation process. When GPU concludes, only final positions of points are copied from device to host. Data transference between host and device is performed at initial and final steps of the algorithm. During core operations, data remain at device memory. During kernels execution, bars length and data movements array remain resident in GPU device memory. We implemented our parallel mesh generator in this way, to exploit GPU threads concurrency. In next section, we describe the different steps we designed to run our parallel algorithm of Distmesh on a GPU.

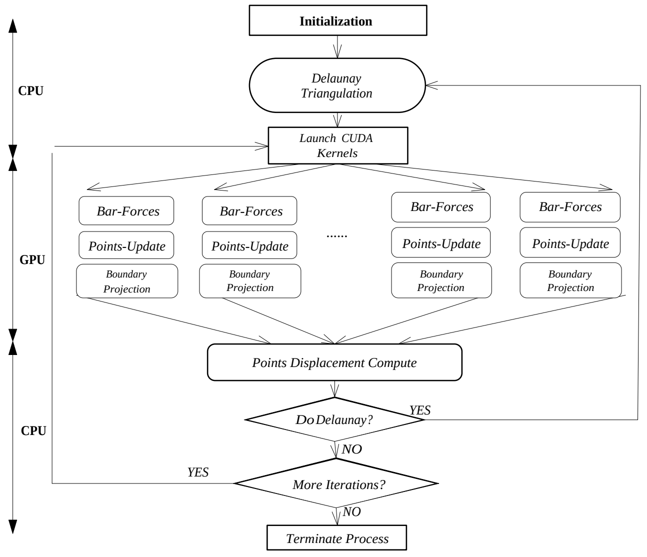

Fig. 1: Mesh Generator Algorithm in CPU-GPU.

4.1 GPU kernels

As a first step in writing the parallel CUDA program, we had to identify those code blocks we could separate from sequential program as a CUDA kernel. Here we explain each kernel written to exploit both data parallelism of the algorithm itself and the GPU architecture. Once written, data on the GPU is persistent unless it is deallocated or overwritten, remaining available for subsequent kernels.

Every node location remains fixed during its total force computing, that's why this task may be parallelized. Figure 2 shows data structures, kernels and the relationship between them. Data structures were selected to avoid the branch instructions in kernels code. This was possible by increasing the compute over the array elements, in order to minimize the use of *if-then-else* control instruction. The general description of each kernel is expressed bellow.

Bar length: We mapped a thread per bar, making every thread compute its corresponding bar length. Threads read every data stored in *Bars* array and store their results in *Length Bar* array. This was a suitable condition for thread computing. Subsequently, we had to do a sum over all elements of the lengths array, and so calculate relax length l_0. This was not a suitable condition for thread computing, we will refer to this later.

Scalar Forces: This kernel launch a thread per length bar. Data are supplied by length array, at global device memory. Every thread applies the same operation to every data, to calculate the resultant scalar force applied to each bar. Then stores the results in a new data structure, *Move.* Move array dimension is in correspondence with the number of bars.

Points Movements: This kernel reads data movement -

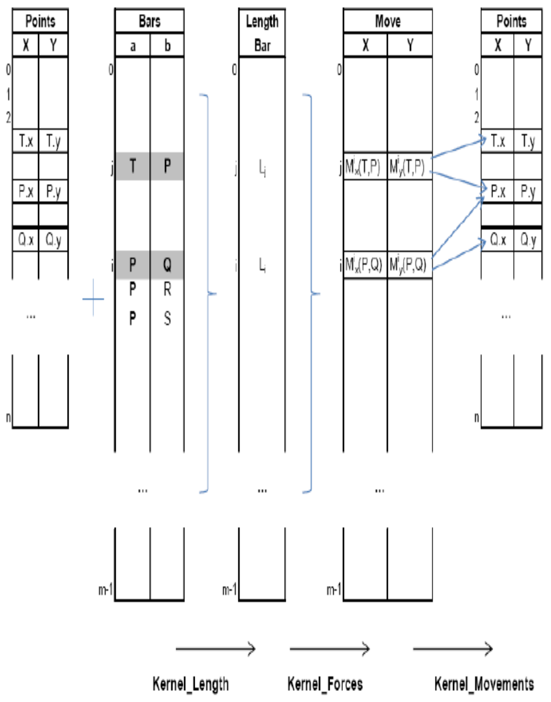

Fig. 2: Data Structures created by the kernels in GPU.

which was calculated with Scalar Forces kernel- in every bar. Each bar contributes to the total movement of all the meshpoints, affecting their extreme points. The final position of each point is obtained by summing all these movements, as shown in Figure 2.

Boundary Projection: Some points go outside the geometry after updating process. This kernel is in charge of projecting the points moved out from the domain, by relocating them to the nearest point in the border. This relocation is performed by calculating the numerical gradient.

The next step in writing a CUDA program, was to manage data transfer between RAM memory and the GPU global memory.

4.2 GPU kernels optimization

As we explained before, the data transfer from host to device is made only twice, before launching the kernels and when GPU process ends. We don't have the bottleneck memory transfer problem during GPU computation [14]. As shown in Figure 3, the data transfer time is 5.7% of GPU time, for the entire process. This graphic was obtained with CUDA Profiler tool. The kernels outlined above are limited by the rate at which the GPU can issue instructions; they are compute bound. To improve the performance, we optimized memory access using shared memory. CUDA uses share memory to help reducing overfetch [15]. To avoid multiple simultaneous accesses to memory, we could efficiently load data arrays from global to share device memory, thus ensuring coalesced readings [16]. CUDA *Atomic Add* functions are the core arithmetic operations to sum array elements and to calculate forces and points movement. These functions ensures that the readings will be done without any interference from other threads. Synchronization is guaranteed by CUDA if multiple threads, in different blocks, access to the same variable to perform *read-modify-write* operations [10][11].

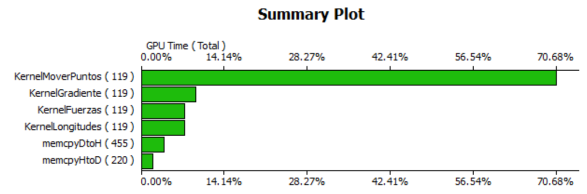

Fig. 3: Kernels and data transference.

Bars length kernel optimization: In a first version, relax length l_0 was performed by one thread. The other threads remain idle in the meantime. The run time was improved copying lengths bar structure to a new array in share memory and using atomic functions to sum lengths bar. Figure 3 shows its little kernel runtime compared to the total kernels execution time. This kernel uses a CUDA Atomic-Add function on share memory to sum bar lengths.

Points movements kernel optimization: We modified this kernel to avoid using a huge data array. It was possible performing atomic operations. This kernel reads movements stored in move array, as shown in Figure 2, launching a thread per row. Each row represents the scalar force at a bar and it contains *x* and *y* component of points movement, then each thread modify the position of two points. This action was optimized by using the CUDA *ATOMIC_FLOAT_ADD* function, obtaining a significant improvement in performance.

We present the experimental results in next section.

5. Experimental Results

In this section we evaluate the computational performance of our GPU parallel version of Distmesh on a platform consisting of a Intel Xeon dual-core processor with 4GB of main memory running at 3.2 GHz, connected to a NVIDIA Tesla C2070 with CUDA driver and runtime version 4.0. This GPU is comprised of 14 streaming multiprocessors (SMs) of 32 streaming processors (SPs) for a total of 448 SPs CUDA cores and its CUDA capability is 2.0.

Table 1 shows CPU and GPU execution time. These measurements of time consider the complete execution of

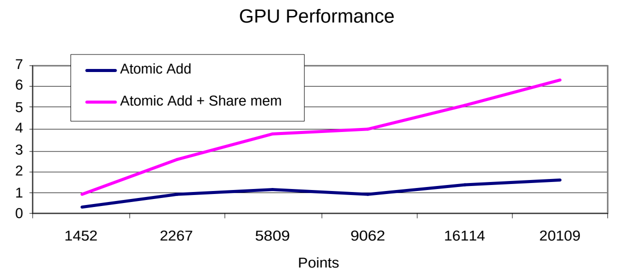

Fig. 4: Comparison of execution time.

Table 1: GPU Version: Execution Time (sg.)

Node Points	*CPU Time*	*GPU Version1*	*GPU Version2*	*Speedup Complete*
1452	0.035	0.130	0.037	0.95
2267	0.290	0.320	0.112	2.59
5809	0.687	0.580	0.183	3.75
9062	1.689	1.820	0.425	3.97
16114	3.388	2.510	0.666	5.09
20109	7.786	4.860	1.239	6.28

the algorithm, including both the executed phase by the CPU and the GPU, including memory transfer overhead. The maximum speedup achieved is: 6.28. GPU execution time was measured performing atomic add operations in *Version 1*, and performing atomic add operations at shared memory in *Version2*. We obtain a significant improvement in the second case. The execution time evolution, for a complete run, is presented in Figure 4.

To ensure consistent results when computing our sequential(CPU) and parallel(CPU+GPU) algorithms, we compared the number of iterations done and how many of those invoked to Delaunay. In Table 2, we present the relationship between Delaunay and Persson iteration, with the aim of showing consistency between CPU and GPU implementation. Rate values are close enough in both cases, allowing ensure satisfactory results, in accordance with original Distmesh results in Matlab. Moreover, we achieve a substantial improvement by measuring only core operations in the GPU version of mesh generator. The speedup reached 53X. We call core operations to GPU kernels outlined in section 4.

6. Related works

In this section we give a brief overview on mesh generator algorithms on GPU. Démian Nave et al. present their approach to parallelizing the Delaunay mesh generation, which can be parallelized in a natural way. This is a similar

Table 2: GPU-CPU execution: Delaunay and Persson iterations.

Node	*CPU*		*GPU*		*Speedup*
Points	*Delaunay*	*Persson*	*Delaunay*	*Persson*	*Kernels*
1261	25	265	23	253	4.77
2827	39	365	30	313	6.81
5025	45	381	35	381	4.94
7851	49	487	38	487	5.09
11313	65	658	44	616	12.08
15395	150	1056	134	983	12.10
20109	231	2083	212	2405	17.35
31409	728	7764	719	6980	53.38

situation to our work. They emphasize the importance of mesh generation algorithm and of Delaunay method in particular [17]. In 2008, Rong et al. present their approach to GPU computing. They enhance Delaunay triangulation using GPUs as a parallel co-processor in charge of the triangulation on a given set points in 2D. The best results are obtained for a large number of points, when they achieve a 53% improvement compared to *Triangle* Delaunay algorithm [18]. An interesting GPU mesh generator algorithm is presented in [19]. The authors propose a two-phase iterative GPU based method, that transforms any 2D planar triangulations and 3D triangular surface meshes into their respective Delaunay form. They used this algorithm to simulate sten deformation, where the geometry of triangulation changes dynamically and requires restore Delaunay conditions to interactive real time levels. This situation is similar to points retriangulation needed in our work, where we use Delaunay triangulation too. We are working on Delaunay parallelization to improve our work, and we are interested in the previous papers.

7. Conclusions

We wish to highlight the GPUs technology suitability to improve performance of mesh generators algorithms. We showed how the efficient Matlab Distmesh algorithm can be parallelized by processing its mesh nodes concurrently and taking advantage of its data structures. Our results gives us an idea of the computing power offered by GPUs and the virtual machine defined by CUDA, which exhibit scalability to programmers. We initially ran our application in a NVIDIA G80 series card; despite being old devices, we obtained good results. Then, we could run the CUDA program in a TESLA card making minimal changes to the kernels code. This architecture enabled us to use Atomic functions in floating point.

We presented in this paper a developmental stage of our work and it shows our initial experiences, which resulted in a significant decrease of algorithm execution time. The Persson method generates high-quality meshes, which were perfectly reproduced for domains in 2D with our parallel algorithm. This approach requires improving the management of large amount of data when dealing with complex geometries and non-uniform meshes. Anyway, we intended to provide a contribution to this topic development, in the search of high performance in GPUs computing.

8. Acknowledgments

The authors thank to Gabriel Acosta for his help in making this work possible. The work was supported by the research project PICT2007-910-CONICET-ARGENTINA.

References

[1] N. Corporation, *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.3*, 2009. [Online]. Available: http://developer.nvidia.com/nvidia-gpu-programming-guide

[2] "Nvidia corporation," LO-tesla-brochure-12-lr.pdf, 2010. [Online]. Available: http://www.nvidia.es/object/LO-tesla-brochure-12-lr.html

[3] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, Mayo 2008.

[4] J. Nickolls and W. Dally, "The gpu computing era," *IEEE Micro*, pp. 56–69, Marzo 2010. [Online]. Available: http://www.computer.org/portal/web/guest/home

[5] U. B. Per-Olof Persson Department of Mathematics, "Distmesh - a simple mesh generator in matlab." [Online]. Available: http://persson.berkeley.edu/distmesh/

[6] J. R. Shewchuk, "Mesh generation for domains with small angles," in *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*. New York, NY, USA: ACM, 2000, pp. 1–10.

[7] P.-O. Persson and G. Strang, "A simple mesh generator in matlab." *SIAM Review*, vol. 46, pp. 329–345, 2004. [Online]. Available: http://persson.berkeley.edu/publications.html

[8] T. R. Halfhill, "Parallel processing with cuda. nvidia's high-performance computing platform uses massive multithreading." *Microprocessor Report*, January 2008.

[9] *Optimization. NVIDIA CUDA C Programming. Best Practice Guides.*, NVIDIA Corporation, Cuda Toolkit, July 2009.

[10] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable Parallel Programming*. ACM QUEUE, April 2008.

[11] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, E. Inc., Ed. Morgan Kaufmann, 2010.

[12] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, 2008.

[13] G. Leach, "Improving worst-case optimal delaunay triangulation algorithms." in *In 4th Canadian Conference on Computational Geometry*, 1992, p. 15.

[14] J. Stratton, S. Stone, and W.-m. Hwu, "Mcuda: An efficient implementation of cuda kernels on multi-cores." University of Illinois at urbana-Champaign - Center of Reliable and High-Performance Computing., Tech. Rep., April 2008.

[15] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, and S. S. Stone, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda abstract," in *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM*, 2008, pp. 73–82.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, 2008.

[17] D. Nave, N. Chrisochoides, and L. P. Chew, "Guaranteed-quality parallel delaunay refinement for restricted polyhedral domains," in *Computational Geometry (COMGEO)*, ser. 28, 2004, pp. 191–215.

[18] G. Rong, T.-S. Tan, T.-T. Cao, and Stephanus, "Computing two-dimensional delaunay triangulation using graphics hardware," in *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2008, pp. 89–97.

[19] C. Navarro, N. Hitschfeld, and E. Scheihing, "A parallel gpu-based algorithm for delaunay edge-flips," in *27th European Workshop on Computational Geometry (EuroCG)*, I. M. Hoffmann, Ed., 2011, pp. 75–78.