INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids 2012; **70**:1393–1404
Published online 26 January 2012 in Wiley Online Library (wileyonlinelibrary.com/journal/nmf). DOI: 10.1002/fld.2750

Efficient generation of adaptive Cartesian mesh for computational fluid dynamics using GPU

Seyoun Park and Hayong Shin*,†

Department of Industrial and Systems Engineering, KAIST, Daejeon, South Korea

SUMMARY

Mesh generation has been frequently the most time consuming step in typical CFD analysis studies. In the past two decades, adaptive Cartesian mesh methods have gained increasing popularity among CFD researches, mainly because of its simplicity and the possibility of automating mesh generation step. In contrast to body-fitted mesh, cells in Cartesian mesh are aligned with coordinate axes. In adaptive Cartesian mesh, cells near the objects' boundary are recursively refined using quad-tree (two-dimensional) or octree (three-dimensional). Then, cells intersecting the objects' boundary are clipped by the surfaces, leaving numerous small irregular shaped cells, called cut-cells. Most of the computational efforts required to generate adaptive Cartesian mesh is concentrated on the cut-cell clipping operation. To achieve the computational accuracy in the subsequent numerical solver, the number of cut-cells can be easily over millions, demanding substantial amount of computation time. Reducing mesh generation time matters more especially for unsteady flow simulation involving moving objects, which requires frequent regeneration of meshes for varied postures of the object. In this paper, we report an efficient novel approach to generating adaptive Cartesian mesh by parallelization using the graphics processing unit. The proposed method consists of the following three steps: (1) computing cross-sectional curves of object boundary, (2) octree refinement based on the section curves, and (3) cut-cell clipping. Because each step is designed to be highly parallelizable, we also implemented it on a graphics processing unit, showing orders of magnitude faster performance than the CPU version. Copyright © 2012 John Wiley & Sons, Ltd.

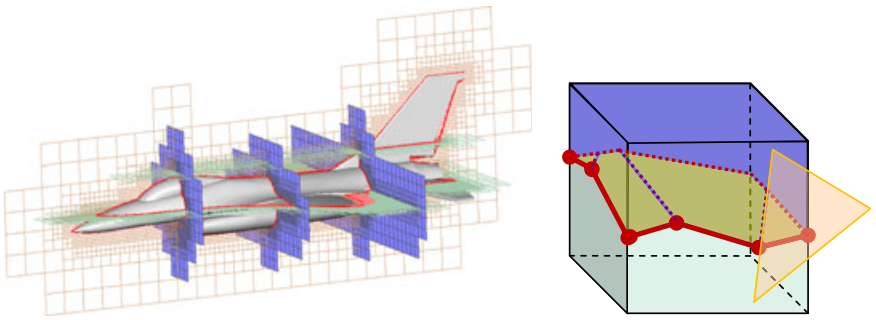
Received 26 June 2011; Revised 30 November 2011; Accepted 3 December 2011

KEY WORDS: adaptive Cartesian mesh; octree; CFD; GPU computing

1. INTRODUCTION

In the CFD dynamics process, mesh generation has been frequently the bottleneck step requiring human experts' intervention. Because of the simplicity of mesh structure and the possibility of fully automating mesh generation for objects with complex geometry, adaptive Cartesian mesh methods have gained increasing popularity among CFD researches in the past two decades [1, 2] Compared with body-fitted mesh in which elements are aligned with the objects' boundary surfaces, cells in Cartesian mesh are aligned with coordinate axes and hence the Cartesian mesh is inherently weak at describing the boundary geometry of objects. Although early works with Cartesian mesh used a stair-case representation of object boundary, in modern Cartesian meshbased CFD analysis, this drawback is alleviated by adaptive mesh refinement and using cut-cells as shown in Figure 1. Cells near the objects' boundary surface are recursively refined using quad-tree (two-dimensional) or octree (three-dimensional) structure. (For the concepts and algorithms of recursive refinement, readers are referred to computational geometry literatures such as [3].) Then, cells intersecting the objects' boundary are clipped by the surfaces, leaving numerous small irregularly shaped

*Correspondence to: Hayong Shin, Department of Industrial and Systems Engineering, KAIST, Daejeon, South Korea.
†E-mail: hyshin@kaist.ac.kr



(a) (b)

Figure 1. Adaptive Cartesian mesh with (a) octree and (b) cut-cell.

cells, called cut-cells To achieve the computational accuracy in the subsequent numerical solver, the number of cells can be easily over millions, among which 10∼15% are cut-cells.

There is a large amount of literature on numerical CFD solvers developed for use with adaptive Cartesian mesh. Recently, high performance solvers using parallel computing technique have also been reported in the literatures [5, 6]. As a result, there is an increasing demand on the efficient generation of Cartesian mesh which can handle very complex body geometry in a robust manner. A nice review on mesh generation for CFD can be found in [4]. Aftosmis *et al* [2] gently introduced the detailed steps in Cartesian mesh generation. In this paper, we focus on the efficient generation of adaptive Cartesian mesh using the graphics processing unit (GPU), and affordable parallel computing machine. Recently, parallel computing using GPU has attracted the attention of researchers in many fields, not only for high performance graphics such as in [7] but also for many general purpose computing areas as listed in [8]. Recently Schwarz and Seidel published a method to generate an octree structure using GPU in [9]. However, their work does not deal with cut-cells which take the most of computational efforts in generating Cartesian mesh for CFD.

In this paper, we propose an efficient and robust method for Cartesian mesh generation, designed to be massively parallelized using GPU. Section 2 describes the overall procedure for Cartesian mesh generation, followed by the detailed steps in the subsequence sections. Experimental results are reported in Section 7, and the conclusion in the last section.

2. OVERALL PROCEDURE

The objective is to generate Cartesian mesh represented by octree and cut-cells from the object boundary surfaces. Boundary surfaces of (possibly multiple) objects are typically given as triangular meshes. Let $\mathbb{B} = \{\Delta_1, \ldots, \Delta_k\}$ denote the set of triangles defining the objects' boundary. The spatial region of concern where we need to generate cells may be either the inside or the outside of the objects. Let $\mathbb{S} = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}]$ denote the spatial area of interest within which Cartesian mesh is to be made. $\mathbb{S}$ can be computed from the object boundary surface if fluid flow is inside of the objects, or it can be specified by the user when fluid flow is outside of the objects. Another parameter from the user is the maximum level of octree (or the size of the smallest cells). From these inputs, we would like to automatically generate Cartesian mesh for B. A Cartesian mesh $\mathbb{C}$ =<OT, CC>, where OT represents the octree structure and CC denotes the set of cut-cells which are the leaf cells of OT clipped by the object boundary surface $\mathbb{B}$. As shown in Figure 2, a cut-cell has detailed geometric information classifying the inside and outside of the body. CFD solvers often require metric properties like the volume fraction and the face areas of cut-cells Computing the detailed information of cut-cells is called *clipping*, which is the most computation intensive step in Cartesian mesh generation.

To ensure smooth transition of the mesh, two spatially adjacent octree cells are allowed to have 1 level of depth difference at most. This is called *'octree balance requirement'*.

DOI: 10.1002/fld

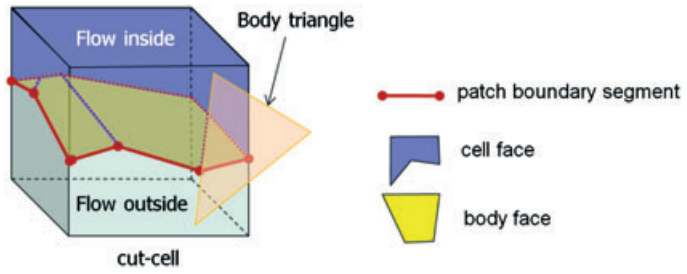


Figure 2. Cut-cell geometries.

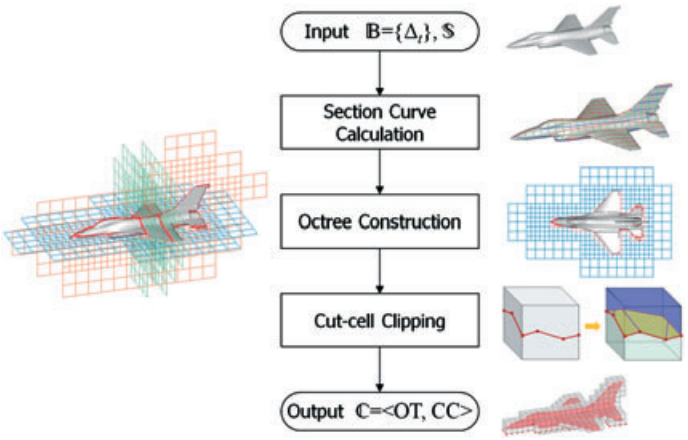


Figure 3. Overall procedure of adaptive Cartesian mesh generation.

Figure 3 shows the overall procedure for generating Cartesian mesh generation, proposed in this paper. The design of the algorithm is easily parallelizable so that we can implement the major steps in GPU, resulting in orders of magnitude faster than the single thread version using CPU.

- Section curve calculation: The first step is to compute the intersection between the objects' boundary surface $\mathbb{B}$ and a series of axis aligned section planes matching with finest level octree cell boundaries. The section curves obtained in this step are to be used in the subsequent steps of octree construction and cut-cell clipping.
- Octree construction: The next step is to construct octree using the section curves. Both top–down and bottom–up approaches are applicable. In the bottom–up approach, a full octree, where each leaf node is refined until the depth limit, is constructed first. Then leaf nodes intersecting with the object boundary are marked by traversing the section curves. Then, nodes are pruned if none of their children are marked. This pruning process continues while maintaining the tree balancing requirement. Although simple, the bottom–up approach suffers from the initial memory requirement, which can be an unnecessary bottleneck. On the other hand, in the top–down approach, the tree starts from the root node having only one cell covering $\mathbb{S}$. Then, each cell is subdivided if the boundary surface passes through the cell. This refining process continues until it reaches the depth limit. In this research, we take the top-down approach to refine the octree structure by tracing the section curves (obtained in the previous step) one by one. Later, the octree balancing requirement is taken care of by further refining cells showing more than 1 level difference with their neighbors.
- Cut-cell clipping: The third step is to compute the detailed geometric information of cut-cells, which are leaf nodes near the object boundary. For each cut-cell $\mathbf{c}$, we can collect the section curve segments lying on the boundary of $\mathbf{c}$. Then the body triangles in $\mathbb{B} = \{\Delta_t\}$ associated with the section curve segments are clipped by the cell $\mathbf{c}$. The clipped portion of body triangles

DOI: 10.1002/fld

$\{\Delta_t\}$ are called 'body face' and used to define the inside (or outside) of the cut-cell geometry together with cell faces, which are axis-aligned cell boundary faces clipped by the body triangles.

This procedure was first implemented in single processor CPU (referred as CPU version). Table I shows the computation time taken in CPU version implementation of the proposed method when applied to a light aircraft model shown in Figure 1. The model has body triangles of 210,000 and the octree is refined to have total of 7,109,990 cells among which 1,367,460 are cut-cells. Maximum level of the tree is limited to 12. As shown in Table I, total mesh generation time is about 300 s most of which was taken in the cut-cell clipping step. If we increase the maximum level to increase the accuracy, the number of cells increases exponentially and so does the time for mesh generation, which makes the use of parallel computation more meaningful.

Because the section curve calculation takes only a small portion (less than 1%) of time of the total computation, we perform this step with more versatile CPU, while octree construction and cut-cell clipping steps are moved to the GPU version for better performance.

3. CALCULATING SECTION CURVE ON CPU

Because the object boundary surfaces are represented by a set of triangles, computing section curves is straightforward. Let us introduce some notations required to explain this step.

- P_i^A : the i-th section plane orthogonal to axis A, where A is either X, Y, or Z (see Figure 4). Section planes are aligned to the wall of octree cells (to be constructed later) at the finest level.
- $\mathrm{L}_{t,i}^A$: a line segment obtained by intersecting a triangle (Δ_t) with a section plane (P_i^A) intersection, stored as two end points and triangle.
- $\mathrm{C}_i^A = \{\mathrm{L}_{t,i}^A\}$: a curve loop consisting of consecutive line segments $\mathrm{L}_{t,i}^A$ on P_i^A. Line segments are oriented and ordered to form a clockwise loop when viewed from $+A$ axis direction.

The following Algorithm 1 explains the procedure to obtain curve loops for each section plane passing the objects. The plane–triangle intersection is straightforward and we will not go into detail. For the fast curve loop construction, we use geometric hashing technique, which promptly reports

Table I. Computation time for Cartesian mesh generation using single thread CPU.

# of body triangles	210,000
# of cells	7,109,990
# of cut-cells	1,367,460
Section curve (s)	3.150
Octree construction (s)	24.703
Cut-cell clipping (s)	275.000

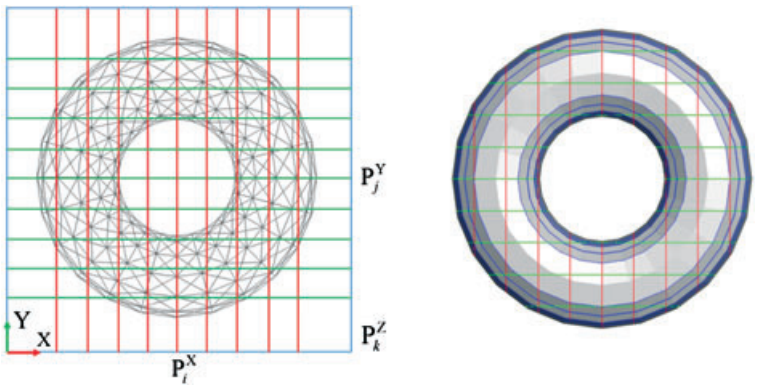


Figure 4. Section planes and section curves

DOI: 10.1002/fld

the line segment having an end point at a given position. With this technique, we can connect line segments sharing end points swiftly. Note that there can be multiple curve loops $\{C_i^A\}$ for each section plane P_i^A.

Although we used naïve implementation as explained above for computing section curve calculation, this step can be further improved by constructing topological information among triangles. One can also use the section curve on one plane to compute the section curve on the adjacent plane by considering only the incremental change of passing triangles. Figure 5 shows an example of section curves computed from a light aircraft model.

Algorithm 1. *Section curve calculation*

```
for each triangle Δ_t ∈ 𝔹
    for each axis A ∈{X, Y, Z}
        for each section plane P_i^A passing Δ_t (see Figure 4)
            compute L_{t,i}^A by intersecting Δ_t with P_i^A and store it on P_i^A

for each axis A ∈{X, Y, Z}
    for each plane P_i^A
        construct section curves loop C_i^A by connecting segments {L_{t,i}^A} stored on P_i^A
```

Algorithm 1. Section curve calculation

4. DESIGN PRICIPLES FOR PARALLELIZATION ON GPU

The GPU is a highly parallel computing machine made of many small cores (individual computing units), originally designed for high performance graphic operations. Because of its potentially massive computing power and very affordable price, there have been many attempts to utilize GPU in general purpose computation other than graphics. In [8], one can find a long list of projects that have successfully exploited the power of GPU in various areas like signal/image processing, bioinformatics, and computational finance. In the early stage, GPU programming involved reinterpretation of the application's computation work in terms of rendering a pipeline job. At the end of 2006, the GPU company nVidia (Santa Clara, California, USA) launched the compute unified device architecture (CUDA), which provided developers with a much comfortable environment when using GPU for general purpose computing. In this research, we used CUDA and Microsoft Visual C++ (Microsoft Corporation, Redmond, Washington, USA), and the target machine was an nVidia Tesla C2050

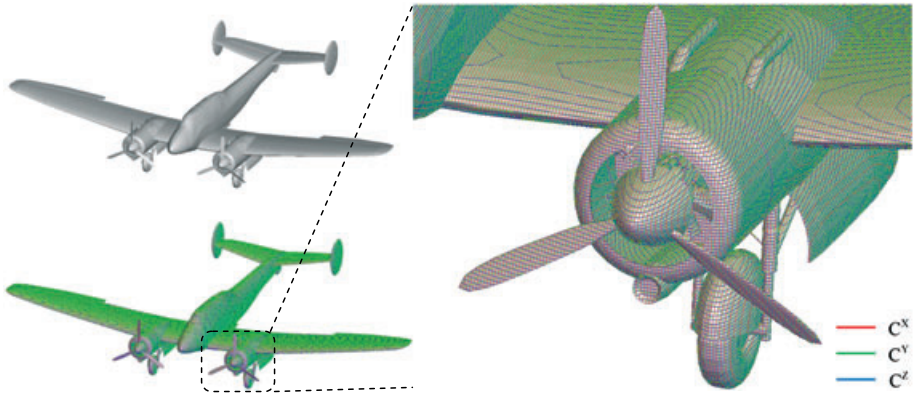


Figure 5. Section curves generation: red lines represent the curves sectioned by planes perpendicular to the x-axis, green lines and blue lines correspond to the y-axis and the z-axis.

DOI: 10.1002/fld

GPU. According to the official specifications from nVidia, the Tesla C2050 model has 448 cores, 3 GB random access memory, 515 GFLOPS peak performance for the double precision floating point computation and 1.03 TFLOPS for the single precision computing work Figure 6 shows a schematic diagram of the GPU run-time model. Although the GPU computation model can be organized in a more sophisticated manner, in this paper we take a simplistic approach consisting of threads, local memory, and global memory:

- Thread : a small set of instructions called kernel to be applied to different data
- Local memory : GPU memory allocated to each thread
- Global memory : GPU memory shared by all threads and transferred to and from main memory accessible by CPU

In GPU programming, special attention has to be paid on allocating computing resources (processing unit and memory) efficiently. Writing data on global memory causes dependency among parallel threads, which will slow down the overall performance significantly. Hence we need to design global memory usage to minimize the interdependency among threads.

The host program running on the CPU dumps the input data to the global memory of the GPU side, and transfers the kernel code to the GPU device. Then, a large number of threads running the same kernel code are created on the GPU. The number of threads can be much bigger than the number of available GPU cores. The GPU controller schedules the execution of threads utilizing its computing resources. Once all threads are completed, CPU can load the kernel code for the next step, and so on. After all jobs are completed, the resulting output stored in the global memory of GPU is transferred to the main memory accessible to the host program. Because the data transfer between main memory and GPU global memory is a major bottleneck, it is desirable to minimize the number of data transfer. Figure 6 shows the schematic model of CPU–GPU interaction we are using.

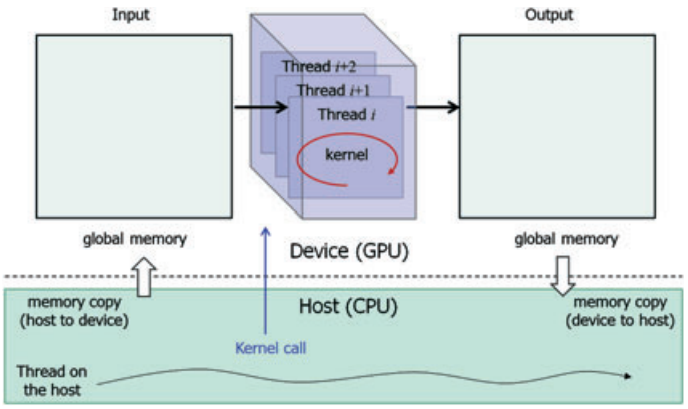


Figure 6. CPU–GPU program model with CUDA.

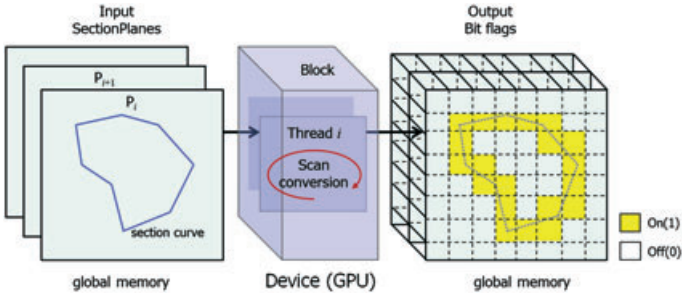


Figure 7. Scan conversion on GPU.

DOI: 10.1002/fld

5. OCTTREE CONSTRUCTION ON GPU AND CPU

This step starts by copying the section planes $\{P_i^A\}$, each of which contains curve loops $\{C_i^A\}$ obtained in the previous step to the global memory of the GPU. We assign one thread per section plane. Each thread processes curve loops $\{C_i^A\}$ stored on the corresponding section plane to refine the octree and store section curve segments to the octree cells. The first step of octree construction is to find out the octree cells requiring subdivision to the finest level. For this purpose, a bit plane to rasterize curve loops is allocated to each thread which processes a section plane. Each bit marked as yellow in Figure 7 represents an octree cell requiring subdivision to the finest level. We use scan conversion technique using a modified version of the Bresenham algorithm for rasterization of curve loops.

Marked bit planes are transferred to main memory for use in the octree refinement step on CPU. Initially, we use top–down construction starting from the initial octree OT with only one cell covering the entire S. Then each octree cell in OT is further subdivided if it contains a marked bit in it. Then, the resulting octree OT is examined to check the octree balancing requirement. If any two adjacent leaf cells in OT have level differing more than 1, the coarse cell is further subdivided.

Once balanced octree is obtained, the next step is to classify the leaf cells into three types: inner cells, outer cells, and cut-cells. Inner cells are those completely contained in the objects, while outer cells are on the opposite side. Cut-cells are those on the objects' boundary. We use flood-fill technique starting from the cell at a far outside corner of $\mathbb{S}$, which is definitely an outer cell.

6. CUT-CELL CLIPPING ON GPU

Cut-cell clipping is to compute the exact division of cut-cells by the objects' boundary surface $\mathbb{B}$ in the following three steps: (1) *section curve extraction* which stores line segments for each cut-cell, (2) *body face clipping* which computes the portion of $\mathbb{B}$ contained in a cut-cell, and (3) *cell face clipping* which splits the side walls of cut-cell by $\mathbb{B}$.

In the section curve extraction step, we assign a thread per section plane P_i^A Each thread traverses the curve loops $\{C_i^A\}$ and stores the line segments to the GPU global memory allocated to each cut-cell as shown in Figure 8.

With this information localized for each cut-cell, we allocate a hread per cut-cell for body face clipping. In this step, each thread computes the intersection points (denoted by blue dots in the right side of Figure 8) of line segment $L_{t,i}^A$ and the cell face. Then, the portion of a body face Δ_i contained in a cut-cell is collected to form a body face polygon.

The next step (cell face clipping) is to clip cell side faces of square shape by the information obtained so far. Because each side face of a cut-cell is shared by two cells, we allocated a thread to a cell side face of a cut-cell, resulting in three threads per cut-cell. Figure 9 shows thread allocation scheme for cell face clipping.

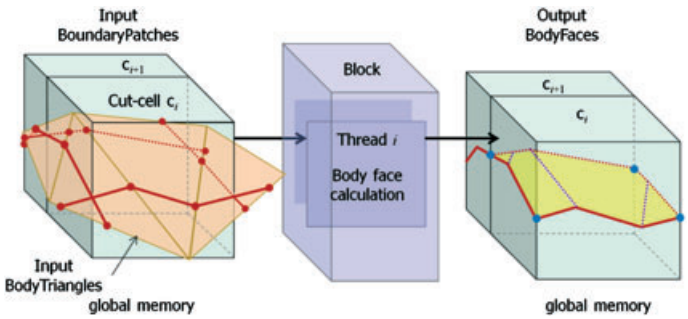


Figure 8. Body face clipping on GPU.

DOI: 10.1002/fld

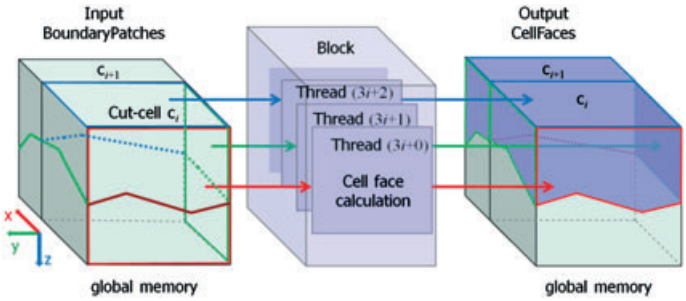
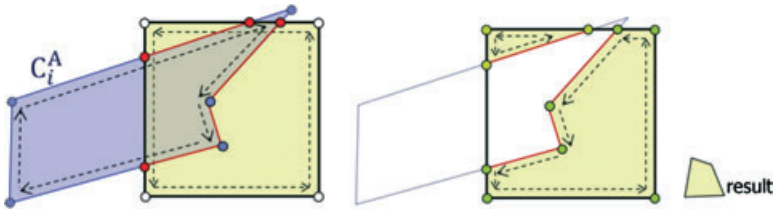


Figure 9. Cell face clipping on GPU.



(a) cell-side face and patch boundary (red edges)

(b) cell face clipping

Figure 10. Weiler–Atherton clipping algorithm.

The cell face clipping to find the red-polygon of cell side wall as shown in Figure 9 is a polygon–rectangle clipping problem in two dimensions, which is a fundamental problem in computer graphics research. Some efficient solutions to this problem are known and in this research we adopt and modify the Weiler–Atherton method [10] which is the most generalized and exact algorithm for the polygon–polygon clipping including holes. Figure 10 briefly shows the concept of the Weiler–Atherton clipping method. This algorithm is to cut a subject polygon (side face of a cell, yellow rectangle in Figure 10 (a)) by a cutter polygon (section curve from objects boundary, blue polygon in Figure 10 (a)) Both polygons (subject polygon and cutter polygon) should be oriented so that a valid portion of the resulting clipped polygon is on the left side of polygon edges. Because we assume that we are interested in the flow analysis on the outside of objects, the cutter polygon (section curve loop) is oriented in clockwise direction, while the subject polygon is oriented in counter-clockwise direction. (If CFD flow analysis is to be conducted on the inside of the object, the section curves should be oriented in counter-clockwise direction so that the valid portion is consistently on the left side of edges.) Figure 10(a) shows the input polygons (the subject polygon in yellow and the cutter polygon in blue). Note that red dots in Figure 10(a) are computed in the previous body face clipping step and reused here. Figure 10(b) shows the resulting clipped cell face, split into two separate polygons in this example.

Cut-cell clipping may result in very small cut-cells, which may cause harmful effects on the convergence speed of the CFD solver and the quality of the numerical solution. Hence, in the post-processing step (on CPU side) of cut-cell clipping, we merge cut-cells with their neighboring cells if the volume of cut-cells are smaller than 1/8 of the unclipped leaf cell cube in octree [11].

7. EXPERIMENTAL RESULTS

We have tested the proposed method of generating the adaptive Cartesian mesh with a few practical models. The testing environment is summarized below:

- Host: Desktop personal computer with Intel Xeon CPU 2.4 GHz (Intel Corp., Mountain View, CA, USA), Host program written in Microsoft Visual C++ (9.0)
- GPU: nVidia Tesla C2050, GPU program written in nVidia CUDA SDK 3.2 (64bit version)

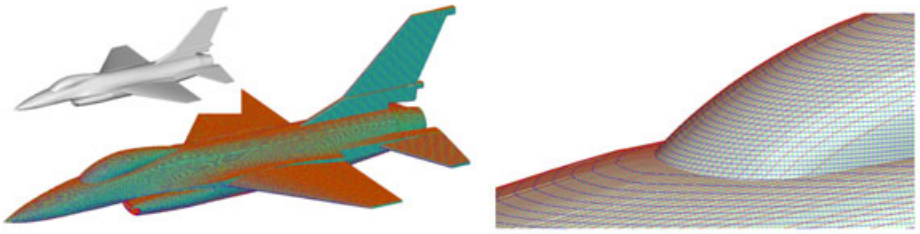
DOI: 10.1002/fld

The test models are:

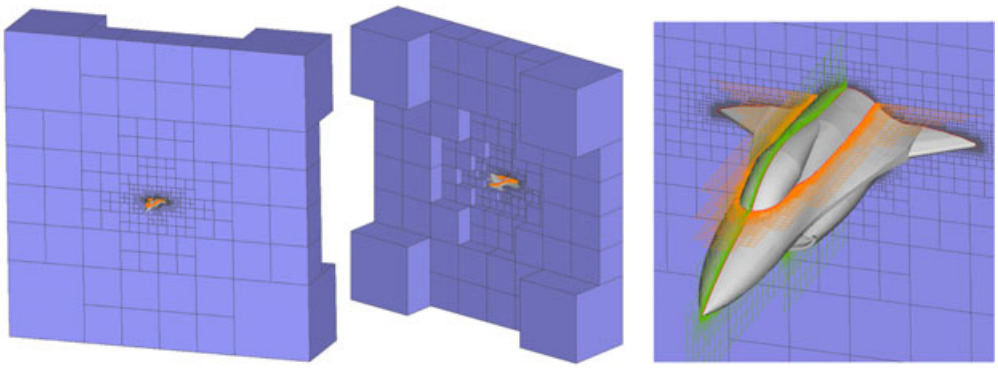
- Fighter aircraft (# of triangles : 360,318)
- Light aircraft (# of triangles : 240,046)
- Casting part (# of triangles : 298,770)

The input model and resulting Cartesian mesh for fighter aircraft is shown in Figure 11. Table II shows statistics on mesh size and computing time (with the maximum octree level 10∼13), comparing CPU version and GPU version Figure 12 plots the growth of computation time as the octree level increases (hence, so does the number of outer cells). As one can see in Figure 11, the growth rate of computation time of the GPU version is much smaller than that of the CPU version.

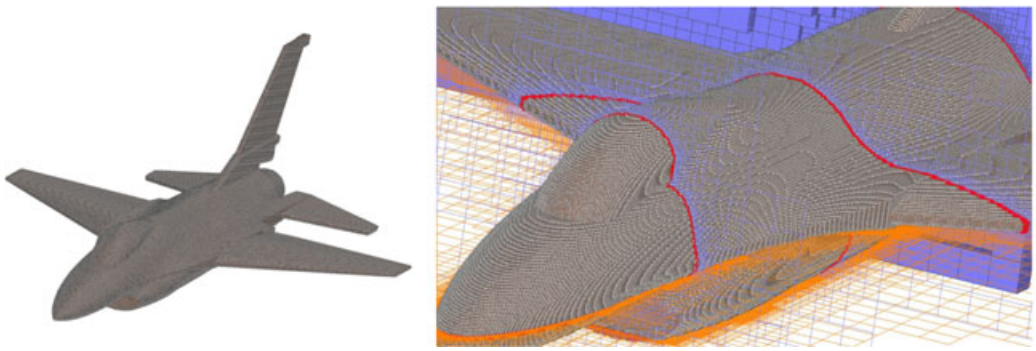
Table III shows the performance comparison of various models with similar number of cut-cells The resulting meshes are shown in Figure 13 (for the light aircraft model) and Figure 14 (for the cast part). In the cast part case, we set a smaller octree size than the aircraft models because the inside of the input model is the fluid flow field. The performance gain shown in Table III is approximately



(a) Fighter aircraft input STL model (left) and section curves



(b) Octree generation; sectional view of front size(left), back side(middle), the size of $\mathbb{S}$ is 5 times bigger than the maximal axial length of the object's bounding box



(c) cut cells only (left) and cut cells and octree sections(right) ; grey transparent faces are cell faces, light brown color represents body faces, and red color shows intersecting faces and nodes with octree sections

Figure 11. Adaptive Cartesian mesh for fighter aircraft with maximum octree level $=$ 12.

DOI: 10.1002/fld

Table II. The time performance of the fighter aircraft with CPU and GPU version.

	# of triangles in $\mathbb{B}$	360,318			
	Size of $\mathbb{S}$	5 times of object bounding box			
Input	Maximum octree level	10	11	12	13
	# of octree leaf cells	239,841	1,020104	4,286,577	17,889,481
	# of outer cells	186,253	710,564	2,651,729	10,338,412
Output	# of cut-cells	35,201	140,915	563,338	2,256,480
Computation time (CPU version)	Section curve	837 ms	1576 ms	2262 ms	3837 ms
	Octree construction	733 ms	2886 ms	10,827 ms	70,669 ms
	Clipping	11,482 ms	30,591 ms	103,880 ms	546,298 ms
	Total time	**13,052 ms**	**35,053 ms**	**116,969 ms**	**620,804 ms**
Computation time (GPU version)	Section curve	80 ms	375 ms	671 ms	968 ms
	Octree construction (GPU computing)	156 ms	343 ms	1341 ms	4836 ms
		76 ms	116 ms	172 ms	217 ms
	Clipping (GPU computing)	327 ms	1248 ms	3604 ms	11,341 ms
		36 ms	103 ms	223 ms	529 ms
	Total time	**563 ms**	**1964 ms**	**5616 ms**	**17,145 ms**

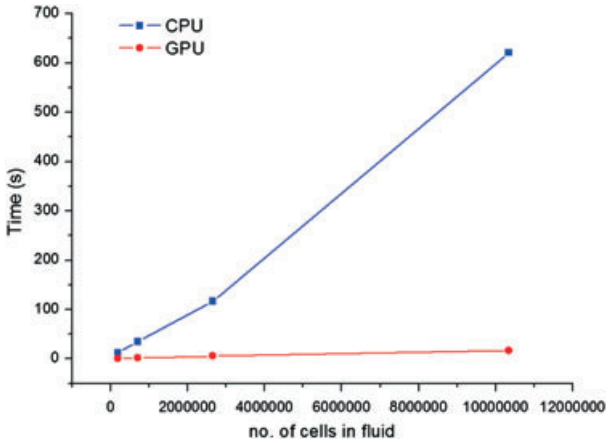


Figure 12. Computation time of the fighter aircraft model.

Table III. Various model tests on CPU

		Fighter aircraft	Light aircraft	Cast part
	# of triangles in $\mathbb{B}$	360,318	240,046	298,770
	Size of $\mathbb{S}$	5 times of the object's bounding box	5 times of the object's bounding box	1.2 times of the object's bounding box
Input	Maximum octree level	13	13	10
	# of outer cells	10,338,412	10,309,625	16,126,658
Output	# of cut-cells	2,256,480	2,361,677	3,849,112
Computation time (CPU version)	Section curve	3837 ms	1310 ms	4025 ms
	Octree construction	70,669 ms	46,597 ms	90,979 ms
	Clipping	546,298 ms	488,390 ms	775,946 ms
	Total time	**620,804** ms	**536,297** ms	**870,950** ms
Computation time (GPU version)	Section curve	968 ms	337 ms	1139 ms
	Octree construction (GPU computing)	4836 ms	3994 ms	3135 ms
		217 ms	58 ms	169 ms
	Clipping (GPU computing)	11,341 ms	10,110 ms	17,097 ms
		529 ms	487 ms	958 ms
	Total time	**17145** ms	**14,441** ms	**21,371** ms

DOI: 10.1002/fld

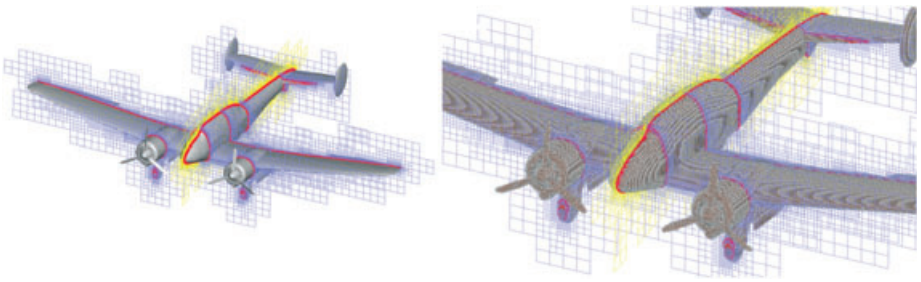


Figure 13. Light aircraft model; sectional view of cut cells(left) and full display (right).

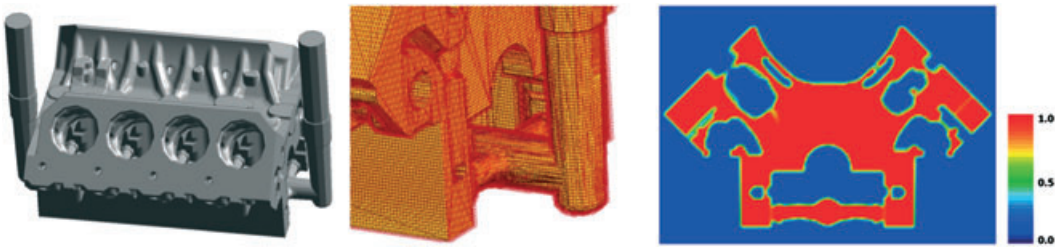
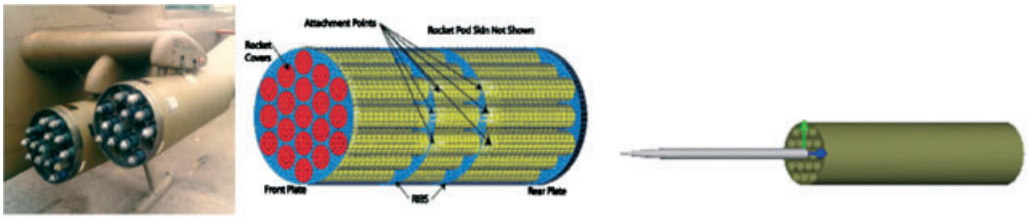
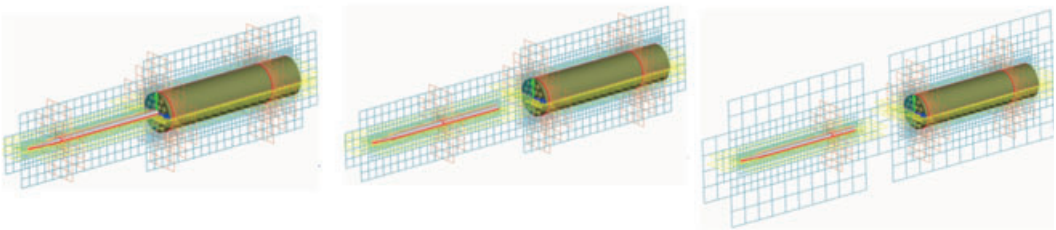


Figure 14. Cast part; input triangular model (left), clipped cut-cells (middle), volume section (right).



(a) photograph of a MLRS and rocket pod structure of [12] (b) STL models for the unsteady simulation



(c) Cartesian mesh for the unsteady simulation of moving object

Figure 15. Rocket launching from a multiple launch rocket system.

35∼40 times faster when using GPU. Such computational advantage can be more significant for CFD analysis of unsteady (moving) objects as shown in Figure 15. In this example, we were able to generated Cartesian meshes at 50 different positions of moving rocket in 4.67 s using GPU.

8. CONCLUSIONS

In this paper, we proposed an efficient method to generate adaptive Cartesian mesh using massive parallelization with GPU, which can be easily implemented. The performance is 30∼40 times faster than the single threaded CPU version. The proposed method for Cartesian mesh generation can also be implemented as multithreaded code running on a CPU, which may take benefit of more flexible memory utilization compared with the restrictive memory structure in GPU implementation. Hence, the performance of the GPU version may not be directly translated to be equivalent with

DOI: 10.1002/fld

parallel CPU implementation requiring 30–40 cores. We will leave the parallel implementation on multicore CPU and the performance comparison with GPU version as a future study.

With the ability to generate adaptive Cartesian mesh swiftly, we could:

- Reduce the CFD analysis cycle
- Enable repeated generation of meshes for moving objects

Implementing parallel computing with GPU requires special attention paid to localized use of computing resources. In this paper, we proposed a simple memory model and thread allocation that can be utilized in Cartesian mesh generation. Because most of the computation time for Cartesian mesh generation is consumed in the cut-cell clipping step, we focused on localizing information required for cut-cell clipping. For CFD study of viscous fluid requiring high precision around the objects boundary, Cartesian mesh method alone may not provide adequate precision. It remains as a further study direction to use GPU parallelization for automatic generation of a body fitted layer of mesh around the object boundary embedded in a Cartesian mesh represented as octree. Although developing a CFD solver using GPU is beyond the scope of this paper, we believe GPU can greatly reduce the overall CFD study cycle if one can utilize GPU for both mesh generation and solver.

ACKNOWLEDGEMENTS

This work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract UD080042AD and UD090020CD.

REFERENCES

1. Hartmann D, Meinke M, Schroeder W. An adaptive multilevel multigrid formulation for Cartesian hierarchical grid method. *Computers & Fluids* 2008; **37**:1103–1124.
2. Aftosmis M, Berger M, Melton J. Adaptive Cartesian mesh generation, Handbook of Grid Generation. In *Handbook of Grid Generation*, Thompson J *et al.* (eds). CRC Press: Boca Raton, Florida, 1998.
3. de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. *Computational Geometry: Algorithms and Applications*. Springer: Berlin, Germany, 2000.
4. Baker TJ. Mesh generation: Art or science? *Progress in Aerospace Sciences* 2005; **41**:29–63.
5. Aftosmis M, Berger M, Adomavicius G. A parallel multilevel method for adaptively refined Cartesian grids with embedded boundaries. *AIAA Paper 2000-0808*, Jan. 2000.
6. Hunt JD. An Adaptive 3D Cartesian Approach for the Parallel Computation of Inviscid Flow About Static and Dynamic Configurations. *Ph.D. Thesis*, Department of Aerospace Engineering, University of Michigan, 2004.
7. Liu F, Harada T, Lee Y, Kim YJ. Real-time collision culling of a million bodies on graphics processing unit. *ACM Transactions on Graphics* 2010; **29**(6):1–8. Article:154.
8. www.gpgpu.org.
9. Schwarz M, Seidel H-P. Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics* 2010; **29**(6):1–9. Article:179.
10. Foley JD, van Dam A, Feiner SK, Hughes HF. *Computer Graphics: Principles and Practice*. Addison Wesley: Boston, 1996.
11. Bayyuk S, Powell K, van Leer B. A Simulation Technique for 2-D Unsteady Inviscid Flows Around Arbitrarily Moving and Deforming Bodies of Arbitrary Geometry. *AIAA paper 93-3391-CP*, 1993; 1013–1024.
12. Heavy K, Sahu J, Wilkerson S. *A Multidisciplinary Coupled Computational Fluid Dynamics(CFD) and Structural Dynamics (SD) Analysis of a 2.75-in Rocket Launcher*, Technical Report ARL-MR-529. US Army Research Laboratory: Adelphi, MD, 2002.

DOI: 10.1002/fld