GPU Optimized Marching Cubes Algorithm for Handling Very Large, Temporal Datasets

Henrik R. Nagel

NTNU
Trondheim, Norway
henrik.nagel@ntnu.no

Abstract. In this article is presented an efficient implementation of the Marching Cubes Algorithm using nVidia's CUDA technology, which can handle datasets that are so large that they cannot be loaded into the working memory of the used computer. Two kinds of data are considered: a single 3D grid that is too large to fit in memory and many, temporal 3D grids that individually fit in memory, but combined do not. The presented implementation makes use of the facilities in a software framework for exploring data called DXTK, which is also briefly presented, and the Marching Cubes algorithm is implemented as a module in this framework.

1 Introduction

The Marching Cubes algorithm [1] has been optimized using nVidia's Compute Unified Device Architecture (CUDA) programming technology before. The CUDA toolkit version 1.1, developed by nVidia, in fact comes with one such implementation. In this, a mathematical function is used to create a small 3D grid, which then is loaded into the memory of the used nVidia graphics card. After the isosurface triangles have been calculated, the graphics is rendered immediately thereafter. Although this implementation performs impressively, using it in praxis is difficult. A single volumetric dataset, as used by e.g. the oil industry, typically contain at least 9 GB with floating points and can therefore not be loaded into the working memory of any graphics card today. Volumetric data produced by scientific simulations is also difficult to handle, since these simulations are temporal and often produce a very large amount of data, distributed onto many separate files that each must be processed in sequence, so that an animation is created. An important consideration here is the amount of triangles that is produced. When rendering an animation consisting of multiple isosurfaces for each time step, a very large number of triangles must be kept in the working memory on the computer performing the rendering. It is therefore necessary to compress the vertex data as much as possible in the final steps of the algorithm. The Marching Cubes algorithm, as implemented in the CUDA SDK version 1.1, does not handle this problem either and can therefore not be used for handling the kinds of data that typically are used in the world of science.

This article presents an implementation that both is efficient and solves the problems mentioned above.

2 Motivation

In 2006 a new Data Exploration Tool Kit (DXTK) was being developed by the author of this article. The toolkit is modular and thus divides the task of processing data into many separate modules that send data between each other. To extend the toolkit with a Marching Cubes module, the 77 GB dataset for the visualization contest at the IEEE Visualization 2006 conference was used as a test case. The first implementation of the algorithm was sequential and precisely as described in literature, but two improvements were soon made: the calculation of vertex normals was corrected and the produced vertex data was compressed.

To calculate how much a 3D graphics surface should be lit, three vectors are used: the vector that points in the direction of the light, the triangle normals, and the vertex normals. Using the first two produces surfaces with abrupt transition between adjacent triangles, making the triangles clearly visible, while using the first and the third vector produces smoothly looking surfaces. The Marching Cubes algorithm, as described in literature, used the first approach. The reason for this is that using the second approach requires a search operation to take place of the triangles that are connected to a given vertex, so that the vertex normal can be calculated as the average triangle normal. In praxis, 3D grids are processed one layer at a time, so it is only necessary to search through the triangle normals in three triangle layers, when calculating a vertex normal.

The second problem was that too much triangle data was produced for the visualization to be possible on the used PC. The solve this problem, the vertex data was compressed by only keeping each vertex and its normal once and then referencing them as many times as required in an index array. However, this also requires a search operation to take place, since one, for each vertex, must search in a list of vertices for whether it occurred in the previous layer.

Having implemented this, as well as functionality for storing the produced triangle data in files, the used PC with an AMD64 3500+ CPU used between 4 and 7 hours to process the 77 GB dataset, depending upon the number of isosurfaces that should be produced for each time step. This implementation of the Marching Cubes algorithm was therefore used at nights, so that the results would be ready next morning. This article, however, describes recent work in optimizing the sequential implementation, so that near interactive execution times is obtained on a PC with an Intel Quad core Q6600 CPU, 4 GB ram and 2 CUDA capable 8800 GTX graphics cards with each 768 Mb ram.

3 DXTK

DXTK programs are created by connecting modules. This has the advantage of making it easy for e.g. people without any knowledge of parallel computing or 3D OpenGL visualization to create advanced parallel programs that both analyzes data in parallel and visualizes the results with 3D graphics.

Parallel programs are described with tree structures. All internal nodes are job-schedulers and all leaf nodes are modules with the main part of the program

code. Multiple different job-schedulers exist, such as a sequential, a PThreads, and a TCP/IP job-scheduler. These can be combined freely, so that different combinations of job-schedulers can be used in different programs.

Communication between modules takes place through "data ports." These are structures that are specialized to handle communication of different kinds of data efficiently. This ensures that data is transferred as efficiently as possible, both when modules are executed with e.g. the sequential scheduler as well as when they are executed with e.g. the PThreads scheduler. Data ports are provided for all basic data types, as well as 3D grids and databases.

4 Handling Multiple, Very Large Datasets

To be able to process as large 3D grids as possible, only the strictly necessary, minimum amount of data is loaded at any time. This applies both for the load module, as well as for the Marching Cubes module. If a dataset that is processed where a single 3D grid is too big to fit in memory, then it is loaded and sent to the Marching Cubes module in chunks that are capable of fitting in memory. Notice, that since the Marching Cubes algorithm produces triangles in the spaces between the values in a 3D grid, it is necessary to repeat the data plane from the previous chunk of data, when a new chunk is sent. When multiple datasets are selected these will be processed in order.

The Marching Cubes module is also designed to use as little memory, as possible. This is achieved by dividing the 3D into a number of planes. There are two kinds of planes: data planes and triangle planes. The reason for this is that the triangles are created between data points in the 3D grid and there therefore is one less triangle plane in each direction than there are data planes. The minimum number of triangle planes required is three. This lower limit is dictated by the requirement that one for a given vertex must be able to find all the triangles that are connected to this point in 3D space, in order to calculate the vertex normal as the average of the triangle normals and these connected triangles can be in both the previous as well as the next triangle plane.

5 The Parallel Algorithm

With these requirements to the CUDA version of the Marching Cubes module, the parallel algorithm is as follows:

1. Initialization.
2. The first two data planes are loaded to the GPU.
3. n = 0
4. Vertices and triangle normals in triangle plane n are calculated on the GPU using the two data planes.
5. For every new data plane do:
   (a) The next data plane is loaded to the GPU.
   (b) n = n+1

(c) Vertices and triangle normals in triangle plane n are calculated on the GPU using the last two data planes.
(d) Vertex normals are calculated on the GPU for triangle plane n-1.
(e) Triangle plane n-2 is compressed on the GPU and sent back to the CPU.
6. Vertex normals are calculated on the GPU for triangle plane n.
7. Triangle plane n-1 is compressed on the GPU and sent back to the CPU.
8. Triangle plane n is compressed on the GPU and sent back to the CPU.

5.1 The Compression Algorithm

Initial requirements

1. During the calculation of triangle vertices and normals an index array was created as well as a "used indices" array with 1 indicating true and 0 indicating false.
2. During the search for identical vertices for calculating vertex normals a "used vertices" array was created with 1 indicating true and 0 indicating false.

Algorithm

1. The "used indices" array is, in parallel on the GPU, used to created a map for compating indices
2. The indices, vertices, vertex normal, and "used vertices" arrays are compacted in parallel on the GPU using the map.
3. The indices are updated in parallel on the GPU using the map.
4. The "used vertices" array is, in parallel on the GPU, used to create a map for compacting vertices.
5. The vertices and vertex normal arrays are compacted in parallel on the GPU using the map.
6. The indices are updated in parallel on the GPU using the map.

6 Results

The 77 Gb dataset will be used as a case study and the impact of each step in the optimization work will be explained: sequential, PThread, and CUDA optimization.

7 Conclusion

This article will show that it is possible to use a GPU optimized version of the Marching Cubes algorithm on very large, temporal datasets. The main technique for handling such large datasets is to process the 3D grids in planes that easily can fit in memory. Vertex compression is used to make rendering possible.

References

1. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. SIGGRAPH Comput. Graph. **21**(4) (1987) 163–169