Fast Surface Extraction and Visualization of Medical Images using OpenCL and GPUs

Erik Smistad[1], Anne C. Elster[1], Frank Lindseth[1,2]

[1] Dept. of Computer and Information Science
Norwegian Univerisity of Science and Technology
[2] SINTEF Medical Technology Trondheim, Norway
{smistad, anne.elster, frank.lindseth}@idi.ntnu.no

Abstract. Marching Cubes (MC) is an algorithm that extracts surfaces from volumetric data. It is used extensively in visualization and analysis of medical data from modalities like CT and MR, often after a 3D segmentation of the interesting structures is performed. Traditional implementations of MC on modern CPUs are slow, using several seconds (even minutes) to return the surface representation before sending it to the Graphics Processing Unit (GPU) for rendering. Fast surface extraction implementations are very beneficial in medical applications where large datasets are processed and time is crucial. Analysis of medical image data usually means changing different parameters so near real-time implementations are very desirable. MC is a completely data-parallel algorithm which makes it ideal for execution on GPUs allowing the result to be rendered on screens in a few milliseconds. In this paper, a MC implementation written in OpenCL that runs entirely on the GPU is presented. We show that our implementation uses a more efficient storage scheme than previous GPU implementations, and that this enables the real-time processing of large medical datasets. Our implementation also shows that GPU implementations written in OpenCL has the potential of being just as fast and efficient as CUDA or shader implementations.

1 Introduction

Creating 3D visualization of large medical datasets using serial processing on the Central Processing Unit (CPU) is very time consuming and inefficient. The Marching Cubes (MC) algorithm was introduced by Lorensen and Cline [7], and has become the standard algorithm for generating surfaces from volumetric data. The original implementation processed each cube sequentially. Image processing in medical imaging applications often requires experimentation of parameter values before a satisfactory result is achieved. For each trial of parameters the result has to be visualized. The total waiting time for creating the surface needed to visualize the result of many trials can become very long. The waiting time can be significantly reduced by exploiting the data parallel nature of the MC algorithm and running it on a Graphics Processing Unit (GPU). These processors have several hundred functional units that can each process a cube in parallel. MC is a completely data-parallel algorithm as each cube in the grid can be processed independently. Parallel implementations of the algorithm has the potential of large speedups as typical medical datasets can have 2 - 200 million cubes. Several parallel multi-chip, multi-threaded and multi-core CPU implementation have been proposed in the literature. A survey of these implementations was done by Newman and Yi [9]. Pascucci [10] accelerated a variation of MC called the Marching Tetrahedra algorithm by creating a quad per tetrahedra and letting a vertex shader program calculate the vertices on the GPU. Klein et al. [6] moved the calculations to the fragment shader by coding the data in textures. Reck et al. [11] improved on these methods by removing empty cells on the CPU using an interval tree. Goetz et al. [4] used a vertex shader program on the MC algorithm and Johansson and Carr [5] improved on this by removing empty cubes using a similar method to that of Reck et al. [11]. Dyken et al. [3] used an algorithm called a Histogram Pyramid to avoid processing empty cubes on the GPU and implemented a vertex shader, geometry shader and CUDA version of it.

Open Computing Language (OpenCL) is a new framework for writing programs that can execute on heterogeneous platforms. OpenCL allows for execution, data transfer and synchronization between different devices, such as CPUs, GPUs and Cell Broadband Engines, without having to write device type or vendor specific code.

In this paper we present a MC implementation written in OpenCL that runs entirely on the GPU. It uses the Histogram Pyramid algorithm, presented by Ziegler et al. [13] and used by Dyken et al. [3] previously on MC. We show that our implementation use a more efficient storage scheme for Histogram Pyramids than previous implementations and that this enables the real-time processing of large medical volumetric datasets.

2 Marching Cubes Implementation on the GPU

MC was introduced by Lorensen and Cline [7] as an algorithm for creating a 3D surface consisting of triangles from a volumetric dataset of scalars. The algorithm uses a parameter, called the iso-value, to classify points in the dataset as either inside or outside the surface. The dataset is divided into a grid such that a number of cubes are formed and each corner in the cubes is represented by a data point in the dataset. By knowing which corners are outside and inside we can determine were to place the triangles of the surface in that cube. In total there are $2^8 = 256$ unique configurations of a cube. But by considering symmetry this can be reduced down to the 15 configurations depicted in Fig. 1.

It has been shown that using only these 15 configurations can lead to topologically incorrect surfaces due to ambiguities. Chernyaev [2] showed how to deal with this by extending the number of unique configurations to 33.

Linear interpolation is often used to place the vertices of the triangles so that the surface becomes more smooth and represents the data better. This technique, as show by Lorensen and Cline [7], place each vertex on the cube's edge according to the difference between the data values of the two corners connected to the edge.

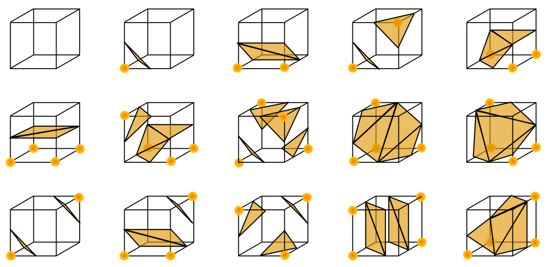

Fig. 1. The 15 cube configurations from [7]. The marked corner points are considered to be inside the surface.

2.1 GPU Computing

GPUs were originally made to speed up the memory-intensive calculations in demanding 3D computer games. These devices are now increasingly being used to accelerate the numerical computations in science. The calculations they were intended for was texture mapping, rendering polygons and transformation of coordinates. The GPU is a type of single instruction, multiple data (SIMD) processor. It can perform the same instruction on each element in a dataset in parallel. GPUs achieve this by having several hundred functional units. These are usually not referred to as ”cores” in the same sense as the multi-core CPUs. McCool [8] defined a core as a processing element with an independent flow of control. The functional units on a GPU does not have an independent flow of control as they are grouped together in a SIMD manner, which implies that the functional units in one group has to perform the same instruction in a clock cycle. These SIMD groups can thus be referred to as cores with the definition above. Most GPUs today also allow branching to avoid executing unnecessary instructions. If the code flow is convergent in a SIMD group, no special treatment is needed, and only the instructions needed are executed. But if the code flow is divergent in a SIMD group, the GPU will run all the instructions, meaning that no time is saved. The GPU use masking techniques to ensure the correct answer is produced by each processing element.

The GPU originally had a fixed pipeline that was constructed for fast rendering. The introduction of programmable shaders in the pipeline of GPUs enabled the possibility of running programs on the GPU. Programming shaders to solve arbitrary problems requires deep knowledge about the pipeline of the GPUs to be able to transform the problem into a rendering problem. General-purpose GPU (GPGPU) programming languages and frameworks like CUDA and OpenCL were created to ease the programming of the GPU.

2.2 Marching Cubes on the GPU

Each cube in the voxel grid can be processed independently of the other cubes. The main problem with running MC on the GPU is how to store the results of each cube in memory in parallel on the GPU. In the serial implementation this is simple by using a stack and add the triangles to the stack as each cube is processed. Two things are needed to store the triangle data in parallel on the GPU: The number of triangles produced so that the proper amount of memory can be allocated. And each cube need an index so that the cubes can all store their results in separate places. It is not possible to assume that all the cubes produce triangles. The device memory is to small for allocating memory for the maximum number of triangles possible. For most medical datasets only a small amount of the cubes actually produces triangles.

NVIDIA has included a CUDA and OpenCL GPU implementation of MC in their SDKs. Their implementations use prefix sum to calculate the sum of triangles and storage index to each cube. Stream compaction is performed on the prefix sum result to avoid processing cubes that does not yield any triangles. Aksnes and Hesland [1] used this method to run MC on large porous rock datasets. Dyken et al. [3] used a data structure called Histogram Pyramid (HP), originally presented by Ziegler et al. [13], in their shader and CUDA implementations of MC. This method was shown to be slightly better than NVIDIAs prefix sum scan approach in cases where the dataset was sparse.

2.3 Histogram Pyramids

The Histogram Pyramid (HP) data structure consists of a stack of textures. These textures can be either 2D or 3D. Fig. 2 illustrates the construction of a HP in 2D. Lets say we are interested in the white pixels in the 4x4 image to the left. The base level of the HP is created as a 2D texture of the same size as the original image. An element in the base level will have a 0 if the corresponding pixel in the image is black and 1 if it is white. The next level of the HP is created by summing 2x2 cells and storing it in another 2D texture with the size halved in both dimensions. This procedure is repeated until a 1x1 texture is left and no more reduction can be performed. The sum in the final level is the sum of the 1s in the base level. This sum can be used to allocate memory.

To retrieve a specific white pixel with a given ID the HP is traversed as shown in Fig. 3. The traversal starts with the second level. The elements are scanned in a Z pattern. When the sum of all scanned elements + the current element are above or equal to the ID of the requested pixel, the procedure jumps to the next level and scans the 2x2 cell that corresponds to the last element in the scan. This process is repeated until the base level is reached. The final element is the one requested.

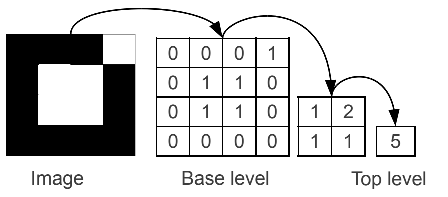

Fig. 2. Construction of a HistoPyramid

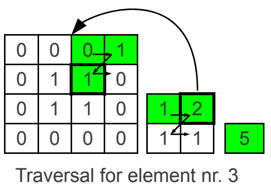

Fig. 3. Traversal of a HistoPyramid

MC is a 3D algorithm, hence the HP has to be designed so that it can be used for 3D. Dyken et al. [3] made a flat 3D layout, meaning that they packed the 3D volume onto a 2D texture and then used the HP in the same way as presented previously. The drawback of using the flat 3D layout is that it requires some extra computation for the address translation from 3D to 2D. It is also possible to extend the HP to 3D by using 3D textures in OpenCL. This was done in our implementation. Writing to a 3D texture requires an OpenCL extension called *cl_khr_3d_image_writes*. AMD currently supports this extension, but NVIDIA does not. In a 3D HP summing and traversal is performed on 2x2x2 instead of 2x2 cells. Also, in the example in the previous section the element in the base level had values of 0 or 1. In MC each cube can produce between 0 and 5 triangles, where each triangle consists of 3 vertices each. So each element in the base level of the HP for MC will have a number between 0 and 5 depending on how many triangles that cube produces.

2.4 Implementation

Our MC implementation consists of 6 main steps as depicted in Fig. 4. It is based on the original MC algorithm by Lorensen and Cline [7]. Hence it has the problem that incorrect surfaces can be produced, but we believe that this implementation can be extended with the method of Chernyaev [2] to deal with these ambiguities.

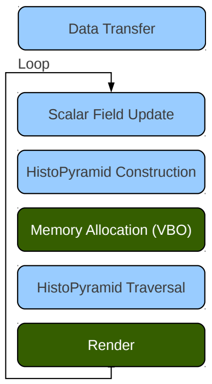

Fig. 4. Block diagram of the MC implementation

The bright/blue steps are performed using OpenCL, while the dark/green steps are performed using OpenGL. Synchronization is necessary for each switch between the two APIs. In this section each step will be explained in detail.

Data Transfer. The first step is to transfer the dataset to the device using the fast PCI express bus. The dataset is stored as a 3D texture on the device. Most GPUs today have a separate texture cache which allow for fast retrieval. This step is only performed once.

Scalar Field Update. In this step the scalar field is updated and the base level of the HP is created. In medical imaging the scalar field is usually constant, but the iso-value can be changed. All the levels in the HP are stored in textures on the GPU. This reduces the impact of the HP construction and traversal steps significantly with cache hits over 90%. A NDRange kernel is run with the same size of the dataset and the base level of the HP. This kernel creates a cube index which is an 8 bit number where each bit represents a corner in the cube of MC. If the corner has a value in the original dataset which is below the iso-value, that bit is set to 1 and if it is above it is set to 0. With this 8 bit index we can look up in a table how many triangles are needed for this specific cube. The number of triangles needed is stored in the base level of the HP.

HistoPyramid Construction. The entire HP can be constructed by a set of NDRange kernel calls in OpenCL. The number of calls needed is log_2 of the size of the base level. If the base level is 256x256x256, 128x128x128 work items are created to fill in the next level with the same size as the number of work items. In the next step 64x64x64 works items are created and so on until the 1x1x1 level is reached. This kernel simply sum all the elements in a 2x2x2 cell in the above level and store the results in the current level.

Memory Allocation. When the HP has been created, the sum of triangles is retrieved from the 1x1x1 top level of the HP, and sent to the CPU via the PCI-express. This sum is used to allocate memory on the graphics card for all the vertices and normals needed to store the surface. The memory is allocated in the form of a vertex buffer object (VBO) in OpenGL.

HistoPyramid Traversal. The memory is filled with the output of the MC algorithm by running a NDRange kernel of the same size as the total sum of triangles retrieved in the previous step. This kernel implements the HP Traversal procedure from section 2.3 using the global ID as the triangle element ID. When the 3D coordinate of the triangle's cube is located, the exact coordinates and normal of each vertex in the triangle can be calculated. The cube index is reused to look up in a table the edges which this triangle should have its vertices on. Linear interpolation is performed on each vertex with the data from the original dataset for each corner. The normals are calculated using forward differences as shown by Lorensen and Cline [7]. The vertices and normals are stored in the VBO made in the previous step.

Render. When the traversal procedure and creation of all the vertex and normal data is finished, the CPU is notified and the control is transfered to OpenGL which then renders the contents of the VBO on the screen.

3 Results and Discussion

We assessed the performance of our implementation by measuring the average frames per second (FPS) and execution time on the graphics device. The FPS and execution time was measured in the rendering loop. The execution time for each step in our implementation was also measured. For comparison, we also tested the OpenGL shader implementation of Dyken et al. [3] that also uses Histogram Pyramids (HPs). We call this implementation *HPMC Shader*. The dataset used for the measurements was a rotational angiography scan of a head with an aneurysm taken from [12] and depicted in Fig. 5. The algorithm was run with a constant iso-value of 0.2 for 4 different sizes of the original dataset with size 512^3. All of the tests in Table 1, 2 and 3 where run on the same computer with an Intel i5 750, 4GB RAM and ATI Radeon 5870 with 1GB device memory. The system was running the AMD Catalyst 11.2 graphics driver and APP SDK 2.3 with OpenCL 1.1.

3.1 OpenCL-OpenGL Synchroniztion

Comparing the performances of the HPMC Shader vs. our implementation, Tables 1 and 2 shows that the HPMC Shader implementation is almost twice as

Size	Execution Time	FPS	HP Size
512^3	3324 ms	0.3	1365 MB
256^3	5 ms	223	85 MB
128^3	3 ms	394	21 MB
64^3	2 ms	519	1 MB

Table 1. Performance of the shader implementation of Dyken et al. [3]

Size	Execution Time	FPS	HP Size
512^3	34 ms	30	148 MB
256^3	10 ms	105	18 MB
128^3	4 ms	233	2 MB
64^3	3 ms	319	$>$ 1 MB

Table 2. Performance of our OpenCL implementation

Update	HP Construction	Mem. Alloc. + Sync	Traversal	Render + Sync
2.4 ms	0.8 ms	5.4 ms	0.4 ms	1.8 ms

Table 3. Execution time of each step in our implementation when run on 256^3 dataset

fast for the three smallest datasets. With the profiling tool gDEBugger it was discovered that this is due to an expensive synchronization between OpenCL and OpenGL. The total time used for synchronization was measured to be from 2 to 20 ms. This makes the GPU stay idle for approximately 70-90% of the execution time for the smallest dataset and 20-30% for the largest dataset. The synchronization cost is a major problem with the OpenCL-OpenGL interoperability. A possible solution to this problem is proposed by The Khronos Group through an extension in both APIs allowing them to share synchronization objects which should enable more efficient synchronization. The extensions are called *GL_ARB_cl_event* and *cl_khr_gl_event*. At the time of writing none of these extensions are implemented by any of the GPU vendors.

3.2 HistoPyramid Size

For the largest dataset the performance of the HPMC Shader implementation drops to below 1 FPS, while our implementation still runs in real-time. As Table 1 shows, the memory usage on the graphics device for HPMC Shader becomes very large, even larger than the 1GB memory available on the graphics card. Hence the program has to transfer large amounts of data back and forth from the GPU to the main memory over the PCI express. The excess use of memory is due to the way HPMC Shader stores the Histogram Pyramids. HPMC Shader stores the HP in a 2D texture with all the HP levels as Mipmap levels. The disadvantage of this is that the same texture format has to be used for all levels. 8 bit storage format for each pixel is sufficient for the base level because each cube can only produce a maximum of 5 triangles. And the maximum for the second level is $8 * 5 = 40$ which is also within the 8 bit limit. Our implementation has a single texture for each level allowing it to use 8 and 16 bit texture formats when it is sufficient.

If N is the size of the dataset in one dimension and N^3 is the total size. The memory consumption of the HP with our method is calculated in bytes in Eq. 1. The first two levels use only one byte per voxel, while levels 3 to 5 use 2 bytes and the rest use 4 bytes.

$$N^3 + \left(\frac{N}{2}\right)^3 + 2\left(\frac{N}{4}\right)^3 + 2\left(\frac{N}{8}\right)^3 + 2\left(\frac{N}{16}\right)^3 + 4\sum_{i=5}^{log_2(N)} \left(\frac{N}{2^i}\right)^3 \qquad (1)$$

HPMC Shader has to use 32 bit storage for all levels and this leads to a much higher memory usage. The HPMC Shader uses 4 channels in a 2D texture which results in $4 * 4 = 16$ bytes per pixel. The size of the texture M^2 is chosen so that the entire dataset fits into the top level, hence $4M^2 >= N^3$. The memory consumption of the HPMC Shader implementation is given by Eq. 2 which is always larger than the HP Size of our OpenCL implementation.

$$16 \sum_{i=0}^{log_2(M)} \left(\frac{M}{2^i}\right)^2 \qquad (2)$$

Table 4 shows the memory consumption for both methods for even larger hypothetical datasets. Note that in addition to the size of the HP also comes the size of the dataset itself and the geometry data. The GPUs that are commercially available with the largest amount of memory at the time of writing are NVIDIA's Tesla and Quadro GPUs, which have up to 6GB. The HPMC Shader implementation would not be able to fit all the data on this device when run on a 1024^3 dataset. Our implementation on the other hand could do so on a device with only 3GB of memory.

Dataset Size	**HP Size of HPMC Shader**	**HP Size of proposed**
2048^3	87 381	9 509
1024^3	5 461	1 188
512^3	1 365	148

Table 4. Storage size in MBs of HistoPyramids of larger sizes for both methods

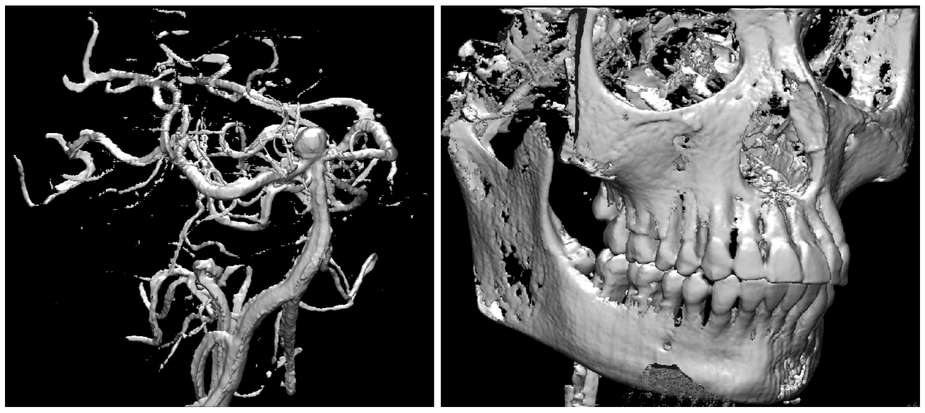

Fig. 5. Two rendered results of the MC implementation

3.3 Other GPU Implementations

NVIDIA's CUDA and OpenCL implementations that use prefix sum were also tested on this dataset using a NVIDIA Geforce GTX460 with 2GB device memory. Their OpenCL implementation was excluded from the comparisons above because the largest dataset it could process was 64^3, running with an average FPS of 452 and memory usage of 23 MB. Also NVIDIA's CUDA implementation failed for the largest dataset due to memory exhaustion.

3.4 Improvements on other platforms

The improvement of storing data in a more efficient format should be applicable to OpenGL shader and CUDA implementations also which would allow larger volumes to be processed with the same speeds as our OpenCL implementation. Due to the expensive OpenCL-OpenGL synchronization, an OpenGL shader implementation using this efficient storage format would probably be faster than our OpenCL implementation.

4 Conclusions

Our proposed OpenCL implementation of Marching Cubes (MC) has shown that efficient GPU implementations written in OpenCL are possible. Since our implementation stores data in a more efficient format, our implementation can also process large datasets faster than previous implementations, including the HPMC Shader [3] which also uses Histogram Pyramids. It can be concluded that GPU implementations of MC, and other image processing methods, written in OpenCL, may be just as fast and efficient as shader and CUDA implementations, but may lose performance to an expensive synchronization costs in the OpenCL-OpenGL interoperability.

The source code of this implementation is available online[1].

Acknowledgments

Great thanks goes to the people of the High Performance Computing Lab at NTNU for all their assistance. The author would also like to convey thanks to the Dep. of Computer and Information Science at NTNU, NVIDIA and AMD. Without their hardware contributions to the HPC Lab, this project would not have been possible.

References

1. E. O. Aksnes, H. Hesland, and A. C. Elster. GPU Techniques for Porous Rock Visualization. Technical report, 2009.

[1] http://github.com/smistad/GPU-Marching-Cubes

2. E. V. Chernyaev. Marching Cubes 33: Construction of Topologically Correct Isosurfaces. Technical report, CERN, 1995.
3. C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, Dec. 2008.
4. F. Goetz, T. Junklewitz, and G. Domik. Real-time marching cubes on the vertex shader. In *Proceedings of Eurographics*, volume 2005, page 2, 2005.
5. G. Johansson and H. Carr. Accelerating marching cubes with graphics hardware. *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research - CASCON '06*, page 39, 2006.
6. T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, pages 186–195.
7. W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 21, pages 163–169. ACM, 1987.
8. M. D. McCool. Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008.
9. T. Newman and H. Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, Oct. 2006.
10. V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2004.
11. F. Reck, C. Dachsbacher, R. Grosso, G. Greiner, and M. Stamminger. Realtime isosurface extraction with graphics hardware. In *Proc. Eurographics*, pages 1–4, 2004.
12. Volvis. www.volvis.org. Accessed 20 Feb. 2011.
13. G. Ziegler, A. Tevs, C. Theobalt, and H. Seidel. On-the-fly point clouds through histogram pyramids. In *Vision, modeling, and visualization 2006: proceedings, November 22-24, 2006, Aachen, Germany*, page 137. IOS Press, 2006.