Real-Time Surface Extraction and Visualization of Medical Images using OpenCL and GPUs

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

1) Norwegian University of Science and Technology

2) SINTEF Medical Technology. Trondheim, Norway

{smistad,elster,frankl}@idi.ntnu.no

Abstract

Marching Cubes (MC) is an algorithm that extracts surfaces from volumetric scalar data. It is used extensively in visualization and analysis of medical data from modalities like CT and MR, usually after a 3D segmentation of the structures of interest have been performed. Implementations of MC on CPUs are slow, using several seconds (even minutes) to extract the surface 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 used and time is crucial. Analysis of medical image data often entails changing different parameters, thus real-time implementations are very desirable. MC is a completely data-parallel algorithm, making it ideal for execution on GPUs. GPU processing enables 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 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 visualizations 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 [9], and has become the standard algorithm for generating surfaces from volumetric data. MC divides the 3D dataset into a set of cubes that can be processed independently. The original implementation processed each cube sequentially. Image analysis in medical imaging applications often requires experimentation of parameters 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

This paper was presented at the NIK-2012 conference; see http://www.nik.no/.

algorithm as each cube in the grid can be processed independently. A typical medical dataset can have from 2 to 200 million cubes. Thus parallel implementations of the algorithm has the potential of large speedups.

The Open Computing Language (OpenCL) is a new framework for writing programs that can execute on heterogeneous platforms. OpenCL enables execution, data transfer and synchronization on different devices, such as CPUs, GPUs and Cell Broadband Engines, without having to write device 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 data structure, presented by Ziegler *et al.* [18]. 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 datasets.

The next section describes the MC algorithm, GPU computing and related work. The methodology section describes our implementation in detail. In the results section, performance measures for our implementation on three different GPUs are presented and compared to the implementation of Dyken *et al.* [5]. The last two sections include discussion and conclusions based on the results.

2 Background

In this section, an introduction to the MC algorithm and GPU computing is given. This is followed up with a discussion on the challenges and related work of running MC on the GPU.

Marching Cubes

MC was introduced by Lorensen and Cline [9] 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 cubes, and each corner in each cube is represented by a data point in the dataset. By knowing which corners are outside and inside, triangles can be placed inside each cube to create the entire surface. In total, there are $2^8 = 256$ unique corner configurations of a cube. However, by considering symmetry this can be reduced to the 15 configurations depicted in Figure 1. It has been shown, that using only these 15 configurations can lead to topologically incorrect surfaces due to ambiguities. Chernyaev [3] 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 and approximate the surface normals, so that the surface becomes more smooth and represents the data better.

GPU Computing

GPUs were originally designed to help speed up the memory-intensive rendering calculations in demanding 3D applications. These devices are now increasingly used to accelerate the numerical computations in science and technology [15, 2]. The calculations the original GPUs were targeting 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

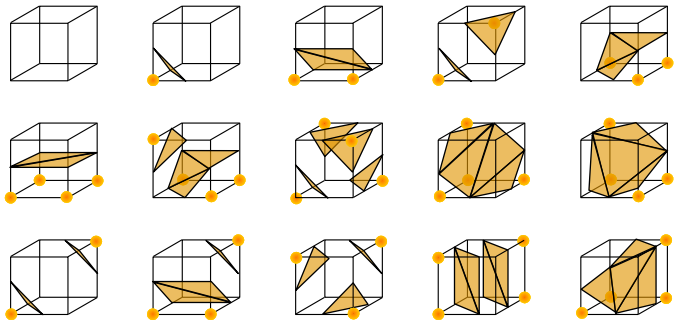

Figure 1: The 15 cube configurations from Lorensen and Cline [9], and the set of triangles that represents the surface. The marked corner points are considered to be inside the surface.

not referred to as ”cores” in the same sense as the multi-core CPUs. McCool [10] defined a core as a processing element with an independent flow of control. The functional units on a GPU do not have an independent flow of control. They are grouped together in a SIMD manner, so 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 above definition. Most current GPUs 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. However, if the code flow is divergent in a SIMD group, the GPU will run all the instructions, and 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 created for fast rendering. The introduction of programmable shaders in the pipeline 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.

Marching Cubes on the GPU

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 [12]. Pascucci [13] 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.* [8] moved the calculations to the fragment shader by coding the data in textures. Reck *et al.* [14] improved on these methods by removing empty cubes on the CPU using an interval tree. Goetz *et al.* [6] used a vertex shader program on the MC algorithm and Johansson and Carr [7] improved on this by removing empty cubes using a similar method to that of Reck *et al.* [14].

Each cube in the voxel grid can be processed independently of the other cubes. The main challenge with running MC on the GPU is how to store the triangles of each cube in memory in parallel. In the serial implementation, this is simple by using a stack and adding the triangles to the stack as each cube is processed. Two things are needed to store the triangle data in parallel on the GPU: 1) The number of triangles produced, so that the proper amount of memory can be allocated. 2) A unique index for each cube, so that the cubes can store their triangles in separate places.

It is not possible to assume that all the cubes produce triangles, because the device

memory is too small for allocating memory for the maximum number of triangles. For most medical datasets only a small amount of the cubes actually produce triangles.

NVIDIA has included a CUDA and an OpenCL GPU implementation of MC in their SDKs. Their implementations use the parallel algorithm 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 do not produce any triangles. Aksnes and Hesland [1] used this method to run MC on large porous rock datasets. Dyken *et al.* [5] used a data structure called Histogram Pyramid (HP), originally presented by Ziegler *et al.* [18], and implemented a vertex shader, geometry shader and CUDA version of it. This method was shown to be slightly better than NVIDIAs prefix sum scan approach in cases where the dataset was sparse. More recently, Ciznicki *et al.* [4] presented a multi GPU implementation of Marching Tetrahedra using the Histogram Pyramid data structure.

Our contribution

This paper builds on the work by Dyken *et al.* [5] and is a continuation of our previous workshop paper [16]. Similar to Dyken *et al.* and Ciznicki *et al.* we have used Histogram Pyramids. The main contributions of our paper are:

- OpenCL is used, which enables execution on GPUs from different vendors. This differs from CUDA which is for NVIDIA GPUs only.
- Dyken *et al.* packed the 3D data in 2D textures and used 2D Histogram Pyramids. In this work, we extend Histogram Pyramids to 3D. This increase cache locality and removes the need for address translations.
- An efficient storage scheme for Histogram Pyramids is presented. This scheme reduces the memory consumption allowing larger volumes to be processed with a single pass.

3 Methodology

This section starts with explaining and extending the data structure Histogram Pyramids to 3D. Finally, a detailed description of our implementation is presented.

Histogram Pyramids

The Histogram Pyramid (HP) data structure consists of a stack of textures. These textures can be either 2D or 3D. Figure 2 illustrates the construction of a HP in 2D. Let's 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 top 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 index the HP is traversed as shown in Figure 3. The traversal starts with the second level. The elements are scanned in a Z pattern as shown in the figure. When the sum of all scanned elements + the current element are above or equal to the index of the requested pixel, the procedure jumps to the

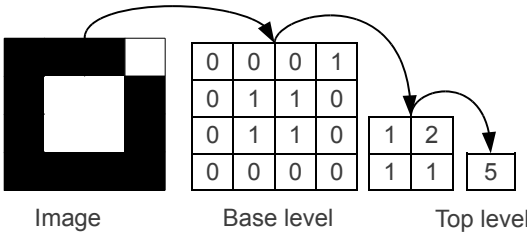

Figure 2: Construction of a HP

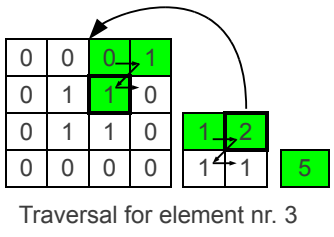

Figure 3: Traversal of a HP

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.

MC is a 3D algorithm, hence the HP has to be designed so that it can be used for 3D. Dyken *et al.* [5] used a flat 3D layout to pack the volume onto a 2D texture and then used the HP in the same way as in the example above. 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, as shown in Figure 4, 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. Due to this restriction on NVIDIA GPUs, a separate version was created for NVIDIA devices. This version uses regular buffers instead of textures, and Morton codes [11] to facilitate 3D caching. This is a bit slower than the 3D texture version used on AMD devices, due to a decrease in cache hits and additional processing. In a 3D HP, summing and traversal is performed on 2x2x2 cells instead of 2x2 cells. Also, in the example above, 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. Thus, each element in the base level of the HP for MC will have a number between 0 and 5, depending on how many triangles each cube produces.

Most modern GPUs have support for several texture formats of different data types. These include 8, 16 and 32 bit integer data types. In our 3D Histogram Pyramid implementation, each level is stored as one texture. This enables the use of different texture formats for each HP level. 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. The next three levels can use 16 bit data types. By using these different texture formats the memory required for the Histogram Pyramid is reduced significantly and this allows faster processing and larger volumes to be processed in a single pass. This differs from the implementation of Dyken *et al.* [5] where all levels are packed in a single texture with the 32 bit format.

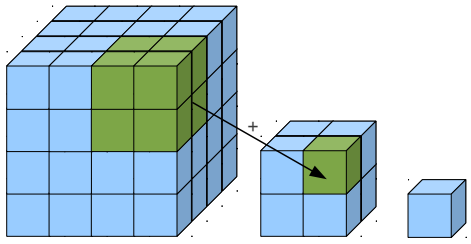

Figure 4: 3D Histogram Pyramid

Implementation

Our MC implementation consists of 6 main steps as depicted in Figure 5. It is based on the original MC algorithm by Lorensen and Cline [9].

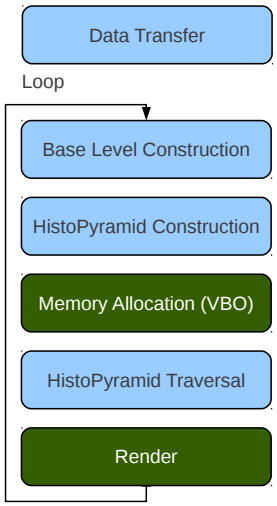

Figure 5: Block diagram of our MC implementation

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.

Base Level Construction. In this step, the base level of the HP is created. Recall that the base level contains the number of triangles necessary for each cube. In medical imaging the scalar field is usually constant. However, the iso-value can be changed, which can change the number of triangles needed. All levels of 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 size of the dataset and the base level. This kernel creates an 8 bit cube index, where each bit represents a corner in the cube. 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 and store it in the base level.

Histogram Pyramid 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 has the size 256x256x256, a NDRange kernel of size 128x128x128 is executed to fill the next level which has the size 128x128x128. In the next step, a NDRange kernel of size 64x64x64 is executed and so on until the 1x1x1 level is reached. This kernel simply sum all the elements in a 2x2x2 cell in the previous level and stores the sum 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. After the memory has been allocated, OpenGL has to synchronize and transfer control back to OpenCL.

Histogram Pyramid 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 3 using the global index as the triangle element index. 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 cube edges that 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 [9]. Finally, the vertices and

normals are stored in the VBO made in the previous step.

The total sum of triangles is not necessarily dividable by a multiple of the units of execution (32 on NVIDIA, 64 on AMD). This sum must also be dividable on the number of work-items in each work-group. If the number of work-items in a work-group is not a multiple of the units of execution, several threads in each work-group will be idle on the GPU. To deal with this, we add a set of dummy work-items so that the total number of work-items is dividable by 64 and set the work-group size to 64. This way, only work-items in the last work-group will be idle.

Render. When the traversal step has created all the vertex and normal data, the CPU is notified and the control is transfered to OpenGL, which then renders the contents of the VBO on the screen. If the iso-value or scalar field has changed, all of these steps can be repeated to create a new surface or the program can continue rendering the next frame using the same surface.

4 Results and Discussion

The performance of our implementation was assessed by measuring the average number of frames per second (FPS) and execution time on the graphics device. For comparison, we also tested the OpenGL shader implementation of Dyken *et al.* [5] 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 [17] and depicted in Figure 7. The algorithm was run with a constant iso-value of 0.2 for 5 different sizes of the original dataset with size 512^3. Each dataset was processed on three different GPUs: AMD Radeon HD 5870 with 1GB memory, NVIDIA GTX 470 with 1280MB memory and NVIDIA Tesla C2070 with 6GB memory. The OpenCL implementations used were AMD APP 2.6 and NVIDIA CUDA 4.2. The FPS and execution time was measured in the rendering loop and gathered in Table 1 and 2. The two implementations were not able to process the largest dataset on all devices. This is indicated with a - in the tables. The execution time for each step in our implementation was also measured and is depicted in Figure 6.

Size	AMD HD5870	NVIDIA GTX 470	NVIDIA Tesla C2070
1024^3	-	-	-
512^3	3324 ms (0.3 FPS)	526 ms (1.9 FPS)	65 ms (15 FPS)
256^3	5 ms (223 FPS)	7 ms (149 FPS)	7 ms (140 FPS)
128^3	3 ms (394 FPS)	2 ms (556 FPS)	3 ms (389 FPS)
64^3	2 ms (519 FPS)	2 ms (524 FPS)	1 ms (1154 FPS)

Table 1: Performance of the shader implementation of Dyken *et al.* [5]. Their implementation was not able to process the largest dataset (1024^3) on any of the devices.

OpenCL-OpenGL Synchronization

Comparing the performances of the HPMC Shader versus our implementation, Tables 1 and 2 show that the HPMC Shader implementation is almost twice as fast for the three smallest datasets. With the profiling tool gDEBugger, it was discovered that the synchronization between OpenCL and OpenGL is very time consuming as shown in Figure 6. The total time used on synchronizing between these two APIs was measured to be from 2 to 20 ms. This makes the GPU stay idle for the major part of the total execution

Size	AMD HD5870	NVIDIA GTX 470	NVIDIA Tesla C2070
1024^3	-	-	1279 ms (0.8 FPS)
512^3	34 ms (30 FPS)	127 ms (7.9 FPS)	136 ms (7.3 FPS)
256^3	10 ms (105 FPS)	19 ms (52 FPS)	19 ms (50 FPS)
128^3	4 ms (223 FPS)	4 ms (241 FPS)	3 ms (276 FPS)
64^3	3 ms (319 FPS)	2 ms (524 FPS)	2 ms (498 FPS)

Table 2: Performance of our OpenCL implementation

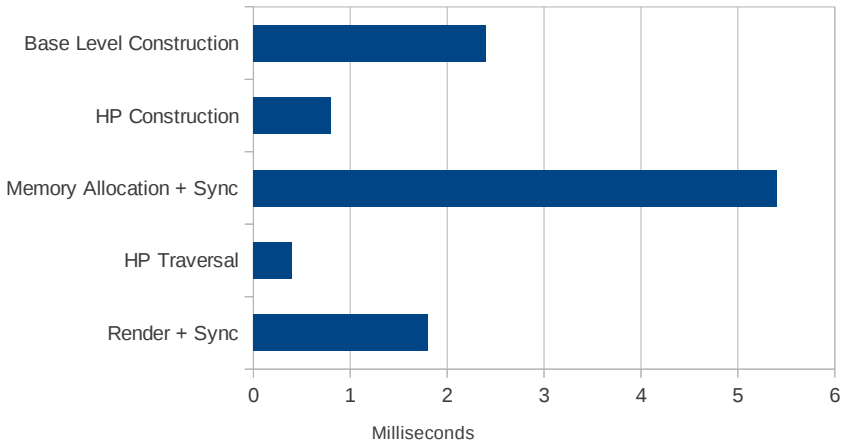

Figure 6: Execution time of each step of our implementation when run on the 256^3 dataset using an AMD Radeon HD5870

time for the smallest datasets. It is thus believed that this synchronization cost is the reason for the HPMC Shader implementation being faster than our OpenCL implementation on the smaller datasets.

The synchronization cost is a major problem with the OpenCL-OpenGL interoperability. A possible solution to this problem has been 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.

Histogram Pyramid Memory Usage

The HPMC Shader implementation was not able to process the 1024^3 dataset on any of the GPUs. This is due to the fact that this implementation requires over 5GB just to store the Histogram Pyramid. Our implementation, on the other hand, use a compressed storage format that use only a little over 1GB to store the HP for the large 1024^3 dataset. The HPMC Shader implementation is able to process the 512^3 dataset. However, there is not enough memory on the HD5870 and GTX 470 GPUs to store the entire HP. This forces the application to move data back and forth from the host to the GPU, resulting in a large drop in speed as can be seen in Table 1. Our implementation use only 148MB to store the 512^3 dataset and can thus extract and visualize the surface quickly on all three GPUs.

The excess use of memory is due to the way HPMC Shader stores the Histogram

Pyramid. 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. Our implementation has a single texture for each level, enabling the use of 8 and 16 bit texture formats when it is sufficient.

N is the size of the HP in each dimension. N has to be larger than the dataset size in each dimension for it to fit, and it has to be a power of 2. The total memory requirements of the HP with our method is calculated in bytes as shown in equation 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.

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

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 $4M^2 >= N^3$. As with N, M also has to be a power of 2. The memory requirements of the HPMC Shader implementation is given by equation 2, and is always larger than the HP size of our OpenCL implementation.

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

Table 3 shows the memory requirements for both methods for datasets of different sizes.

N	M	HP Size of HPMC Shader	HP Size of proposed
2048	65536	**87 381**	**9 509**
1024	16384	**5 461**	1 188
512	8192	1 365	148
256	2048	85	18
128	1024	21	2
64	256	1	< 1

Table 3: Storage size in MBs of Histogram Pyramids of different sizes for both methods. Critical HP sizes are marked in bold/red.

Other GPU Implementations

NVIDIA's CUDA and OpenCL implementations that use prefix sum, were also tested on these datasets using an 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.

Improvements on other platforms

The improvement of storing data in a more efficient format should be applicable to OpenGL shader and CUDA implementations which would allow larger volumes to be processed with the same speeds as our OpenCL implementation. Due to the

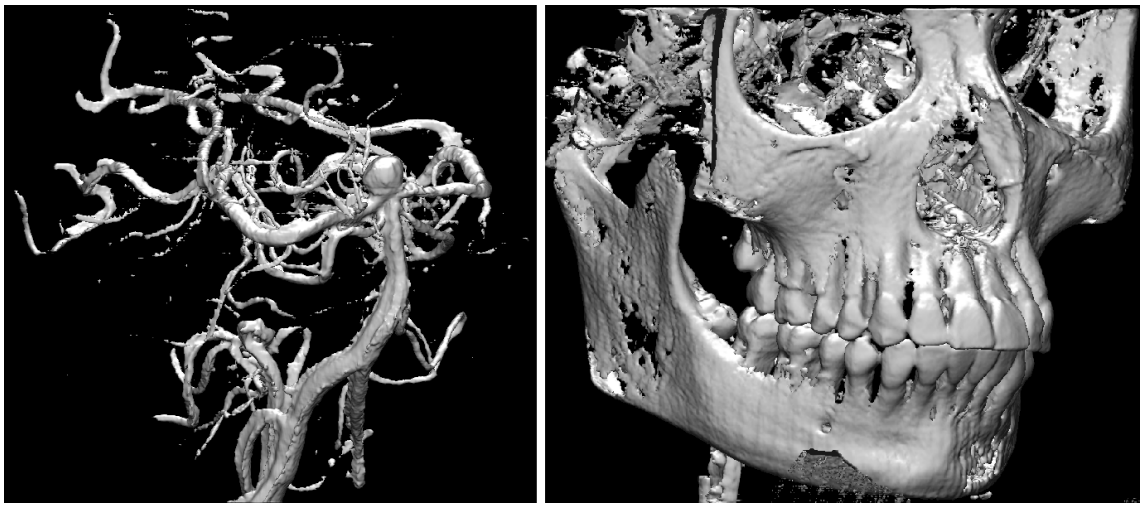

Figure 7: Two rendered results of the MC implementation

expensive OpenCL-OpenGL synchronization, an OpenGL shader implementation using this efficient storage format would probably be faster than our OpenCL implementation.

5 Conclusions

In this paper, an OpenCL implementation of Marching Cubes (MC) that uses Histogram Pyramids was presented. Our novel implementation is able to extract and visualize surfaces from large datasets (512^3 and 1024^3) faster than other implementations. This is achieved by using an efficient storage scheme that significantly reduces the memory usage of the Histogram Pyramid data structure. Our results revealed that the implementation lose some performance due to an expensive synchronization costs in the OpenCL-OpenGL interoperability. This issue will hopefully be overcome in future GPUs.

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

Acknowledgments

Great thanks goes to the people of Anne C. Elster's HPC-Lab at NTNU (`http://research.idi.ntnu.no/hpc-lab`) for all their assistance. The author would also like to convey thanks to the Dept. 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, Norwegian University of Science andTechnology, January 2009. IDI TR no. 02/10, ISSN 1503-416X.

[2] A. Aqrawi and A. Elster. Bandwidth reduction through multithreaded compression of seismic images. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1730 –1739, May 2011.

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

[3] E. V. Chernyaev. Marching Cubes 33: Construction of Topologically Correct Isosurfaces. Technical report, CERN, 1995.

[4] M. Cinicki, M. Kierzynka, K. Kurowski, B. Ludwiczak, K. Napieraa, and J. Palczyski. Efficient isosurface extraction using marching tetrahedra and histogram pyramids on multiple gpus. *Parallel Processing and Applied Mathematics*, 7204:343–352, 2012.

[5] 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.

[6] F. Goetz, T. Junklewitz, and G. Domik. Real-time marching cubes on the vertex shader. In *Proceedings of Eurographics*, volume 2005, page 2, 2005.

[7] 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.

[8] 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.

[9] 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.

[10] M. D. McCool. Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008.

[11] G. M. Morton. A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing. Technical report, IBM Ltd., Ottowa, Canada, 1966.

[12] T. Newman and H. Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, Oct. 2006.

[13] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2004.

[14] F. Reck, C. Dachsbacher, R. Grosso, G. Greiner, and M. Stamminger. Realtime isosurface extraction with graphics hardware. In *Proc. Eurographics*, pages 1–4, 2004.

[15] E. Smistad, A. C. Elster, and F. Lindseth. Real-time gradient vector flow on gpus using opencl. *Journal of Real-Time Image Processing*, pages 1–8. 10.1007/s11554-012-0257-6.

[16] E. Smistad, A. C. Elster, and F. Lindseth. Fast Surface Extraction and Visualization of Medical Images using OpenCL and GPUs. In *The Joint Workshop on High Performance and Distributed Computing for Medical Imaging 2011*, 2011. http://idi.ntnu.no/∼smistad/papers/Fast_Surface_Extraction_and_Visualization_of_Medical_Images_using_OpenCL_and_GPUs/article.pdf.

[17] Volvis. www.volvis.org. Accessed 20 Feb. 2011.

[18] 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.