State of the Art Parallel Computing in Visualization using CUDA and OpenCL

J. Waage

Abstract
In this state of the art report I discuss the newly released parallel computing APIs CUDA and OpenCL, and their adoption and current use in visualization. A brief introduction to each API and their approach to parallel computing is given. The main focus though, is on the application of OpenCL and CUDA in different areas of visualization. Each method and field will be given time according to the amount of literature on their use of parallel computing. Large parts will be devoted to volume rendering and image registration, smaller parts to parallel computing in scatterplots, flow and scientific simulation.

1. Introduction
Lately new parallel computing APIs has opened new opportunities in the field of visualization [RXTC07]. The most notable in this context are NVIDIA’s CUDA [Nvi09] and the Khronos group’s OpenCL [Khr09]. There are other new APIs like RapidMind [BB09] and AMD’s Stream [BB09] but it seems they are not in much in use in visualization. CUDA has been out for a while, and it is what one has adopted in visualization, while OpenCL is so brand new that there is little documentation on its use.

The interest for these APIs is motivated by the computational power they enable. In medical visualization one is in increasing need for quick access to medical imagery before, after, and during procedures [RXTC07] [MOOXS09]. The same is true in other fields that require real time updates, and real time interaction. In many cases even small delays are unacceptable and will render a method useless from a practical point of view [RXTC07] [MOOXS09].

There are a number of different methods of visualization that lend themselves to acceleration using parallel computing, including:

• Ray casting based volume rendering.
• Fourier volume rendering.
• Image registration.
• Flow visualization.
• Scatterplots.

This paper is organized as follows: Section 2 briefly explain parallel computing in general, OpenCL, CUDA, and GPU programming. Section 3 will provide an overview of research using CUDA and OpenCL to accelerate visualization and computation.

2. Parallel computing and APIs
In this section I will talk about GPU programming, parallel computing and the latest APIs to harness this.

2.1. GPU programming
Programmable GPUs have been around for a while, and has been harnessed for visualization using shader programming like GLSL, Cg or lately more computationally oriented solutions like CUDA. The GPU is attractive due to its incredibly fast floating point operations, and its parallel design [Goh09] [Khr09].

There are of course limits on the GPU, and the most important is the transfer from CPU to GPU is slow compared to transfer within the GPU, and the memory on the GPU is limited [Goh09] [SDK*08] [MOOXS09]. This needs to be taken into account to avoid unwanted bottlenecks.

2.2. Parallel computing
Parallel computing refers to a computation where many task are carried out simultaneously. The method works by splitting the problem into smaller parts and solving them at the same time. It can refer to parallelism on different levels, relevant for this paper is data parallelism. Data parallelism is when each processing unit does the same to different pieces.
of data, an example is matrix addition, where each element can be added in parallel [Goh09].

Parallelism has become more and more important, as computational performance has shifted from clock speed to cores, so one has multiple CPUs and programmable GPUs to utilize at the same time [Goh09] [MOOXS09]. Programming paradigms have not yet completely caught up in this regard [Goh09], so different APIs and interfaces are needed to deal with splitting up functions and algorithms and executing these on the CPU or GPU.

To be useful parallel computation needs tasks which can be done in parallel, it’s not going to be useful for sequential problems, or problems where you constantly permute data or that require a lot of communication and result updates. It is best for problems that can be split into parts that are then solved independently.

### 2.3. CUDA

CUDA is NVIDIA’s general purpose computing API. It’s only available for all the latest NVIDIA graphics cards, going back to the GeForce 8 series [BB09]. It uses a programming model based on C. The GPU is through CUDA seen as a set of multiprocessors where one can execute a substantial number of threads in parallel. CUDA offers a much easier interface and faster data exchange between threads than the shading languages mentioned earlier [SDK∗08].

**Figure 1:** Nvidia GeForce 8800 used in many articles on GPU acceleration [Nvi09].

CUDA treats the GPU as a co-processor to the CPU [BB09]. Functions with parallel behavior is written using the CUDA version of C. Before downloading these functions to the GPU, CUDA translates it to the instruction set of the device. Such a program is called a kernel [BB09].

### 2.4. OpenCL(Open Computing Language)

OpenCL is a specification maintained by the Khronos Group [Khr09]. Since it is a specification it needs to be implemented by someone, any implementation have to adhere to the specification to be OpenCL compliant. The specification is open, and anyone can develop an implementation. Differing from CUDA it supports general purpose parallel computations. Giving access to all the computational devices on your system (typically GPUs and CPUs), not just the GPU. It is also not bound to NVIDIA hardware like CUDA is [Goh09].

OpenCL is based on the C99 language and it can be seen thread management framework [Goh09], which frees you from thread management like locks, thread creation and destruction. OpenCL is designed to work together with OpenGL [Goh09].

### 3. Visualization and computation using CUDA and OpenCL

In this section I will present visualization research that uses CUDA and OpenCL to accelerate their computation or visualization.

#### 3.1. Accelerated volume rendering

Accelerated volume rendering has earlier been described in Real Time Volume Graphics [EHK∗06]. The methods adopted in my research were ray casting and fourier volume rendering.

##### 3.1.1. Ray casting

The basic idea of ray casting is to trace rays from the camera into the volume, computing the volume rendering integral along these rays. The main advantage is that these rays are handled independently from each other. This flexibility allows for several optimization strategies, such as early ray termination, adaptive sampling and empty space skipping [EHK∗06].

Since each ray is independent, ray casting can be parallelized on the GPU by associating one pixel with the calculations on one ray [EHK∗06]. To access data with the high internal bandwidth of the GPU, volume data could be store in textures [EHK∗06].

S. Frey and T. Ertl’s [FE09] propose a method to accelerate ray casting for industrial CT data, their focus is on interactive visual inspection for material deficiency. To achieve interactivity their approach is to reduce iteration of the ray casting loop, by skipping regions containing information of no interest, but also guaranteeing that critical parts are not missed. To do this, the entire data is split into regions in a segmentation step configured by the user, leap values within regions are then determined, denoting how many voxels one can leap without leaving the region. Figure 3 show the data structure after segmentation.

Leap voxels are classified by reserving the most significant bit. To avoid that trilinear interpolation is used with
leap voxels (since these now contain values of leaps instead of density values), which would be an undefined operation, S. Frey and T. Ertl use guard voxels around the leap voxels, these turn off trilinear interpolation when encountered, and it’s turned back on when the region is left, this can be seen in figure 2. The implementation uses an ray casting kernel and is accelerated using CUDA. The kernel itself has been modified to do leaps based on the data structure.

Figure 2: Showing one leaping and one not leaping ray in the volume [FE09].

Figure 3: The top left image shows the accelerated ray casting of a toy car, the other images are visualizations of the data structure, where green are density values, and blue is leap values. Intensity shows higher value. [MOOXS09].

D. Juba and A. Varshney [JV07] use a GPU based ray caster to render very large volumes which is represented implicitly by a radial basis function hierarchy (an octree). Such a representation makes it easier to make the tradeoff between memory and computation since one can vary the use of each. D. Juba and A. Varshney use a fitting algorithm to create this representation. The algorithm subdivides the data set into blocks until the fitting error (deviation from original data set) is below a certain threshold or one has reached the full original data resolution. The cells representing the data is then stored in the hierarchy.

During rendering, data and scalar values are calculated from the hierarchy, allowing one to not hold the original data set in memory. The rendering algorithm use CUDA and uses one GPU kernel for each ray following the rays in parallel.

Their implementation allows space skipping and early ray termination. Due to the RBF hierarchy, level of detail can easily be changed by rendering different levels of the hierarchy for different distances from the view. This is shown in figure 4.

3.1.2. Fourier volume rendering

Corrigan et al. [CWV08] has adapted fourier volume rendering to directly use meshless data, instead of extracting data into a grid, and then visualizing it. The implementation was done on NVIDIA hardware using CUDA. A CUDA kernel is used to sample from the fourier transform by executing it over a grid of threads, where each thread is responsible for a partial sum of each sample [CWV08]. These samples are then computed into an image using a complex-to-real FFT from the CUDA library. Figure 5 shows performance on different hardware and different data sets. The technique is capable of interactivity and scalable.

Figure 4: Level of detail rendering, showing different renderings from the left to the right. Where the rightmost have lowest rendering time, and renders less levels of the hierarchy based on distance from view [JV07].

Figure 5: Performance measured in frames per second when visualizing various data sets interactively. [CWV08].

3.2. Image registration

Image registration is the process of aligning images so that corresponding features can easily be related [MOOXS09]. There are two groups of image registration: rigid and deformable. Rigid only allow transformations like rotation and translation. Deformable registration allows other transformation like elastic ones. If both are used rigid ones should be applied first to minimize the time needed to do non rigid alignment of the images. [MOOXS09]
Sugiura et al. [SDK ’08] presents an accelerated method to track a bronchoscope using image registration. The goal is to achieve the speeds needed for real-time tracking. The registration is done between real bronchoscope images and virtual ones derived from CT scans. The virtual images are generated from a volume rendering (ray casting) of CT data. The flow of the method can be seen in figure 7. This volume rendering is accelerated using CUDA. The registration determining image similarity is accomplished using the mean squared error as similarity measure. The registration is partly done on the GPU through CUDA, with only the results being transferred to the CPU, this minimizes the amount of memory transfer between CPU and GPU which otherwise could be a bottleneck.

Figure 6: Visualization of meshless astrophysical data sets [CWV08].

Figure 8: Performance of CPU vs GPU generation of virtual images from CT scans [SDK ’08].

Figure 7: The flow of their method [SDK ’08].

Figure 9: Control flow of kernels [MOOXS09].

Figure 10: Performance of GPU implementation versus CPU implementation [MOOXS09].

The running time of their algorithm can be seen in figure

Sugiyama et al. [SDK ’08] presents an accelerated method to track a bronchoscope using image registration. The goal is to achieve the speeds needed for real-time tracking. The registration is done between real bronchoscope images and virtual ones derived from CT scans. The virtual images are generated from a volume rendering (ray casting) of CT data. The flow of the method can be seen in figure 7. This volume rendering is accelerated using CUDA. The registration determining image similarity is accomplished using the mean squared error as similarity measure. The registration is partly done on the GPU through CUDA, with only the results being transferred to the CPU, this minimizes the amount of memory transfer between CPU and GPU which otherwise could be a bottleneck.

In figure 9 one can see how the kernels were applied and in which order they execute.

Muyan-Orçelik et al. [MOOXS09] use CUDA in their implementation of the demons algorithm, a deformable registration algorithm widely used to match medical volumes. The test data used was 3D CT scans acquired using 4DCT, which obtains sequences of 3D CT data. The implementation used CUDA for numerous operations needed for the demons algorithm. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In all implementations, the algorithm is run for 100 iterations. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In figure 9 one can see how the kernels were applied and in which order they execute.

Muyan-Orçelik et al. [MOOXS09] use CUDA in their implementation of the demons algorithm, a deformable registration algorithm widely used to match medical volumes. The test data used was 3D CT scans acquired using 4DCT, which obtains sequences of 3D CT data. The implementation used CUDA for numerous operations needed for the demons algorithm. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In all implementations, the algorithm is run for 100 iterations. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In figure 9 one can see how the kernels were applied and in which order they execute.

Muyan-Orçelik et al. [MOOXS09] use CUDA in their implementation of the demons algorithm, a deformable registration algorithm widely used to match medical volumes. The test data used was 3D CT scans acquired using 4DCT, which obtains sequences of 3D CT data. The implementation used CUDA for numerous operations needed for the demons algorithm. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In all implementations, the algorithm is run for 100 iterations. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In figure 9 one can see how the kernels were applied and in which order they execute.

Muyan-Orçelik et al. [MOOXS09] use CUDA in their implementation of the demons algorithm, a deformable registration algorithm widely used to match medical volumes. The test data used was 3D CT scans acquired using 4DCT, which obtains sequences of 3D CT data. The implementation used CUDA for numerous operations needed for the demons algorithm. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In all implementations, the algorithm is run for 100 iterations. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In figure 9 one can see how the kernels were applied and in which order they execute.

Muyan-Orçelik et al. [MOOXS09] use CUDA in their implementation of the demons algorithm, a deformable registration algorithm widely used to match medical volumes. The test data used was 3D CT scans acquired using 4DCT, which obtains sequences of 3D CT data. The implementation used CUDA for numerous operations needed for the demons algorithm. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In all implementations, the algorithm is run for 100 iterations. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In figure 9 one can see how the kernels were applied and in which order they execute.

Muyan-Orçelik et al. [MOOXS09] use CUDA in their implementation of the demons algorithm, a deformable registration algorithm widely used to match medical volumes. The test data used was 3D CT scans acquired using 4DCT, which obtains sequences of 3D CT data. The implementation used CUDA for numerous operations needed for the demons algorithm. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In all implementations, the algorithm is run for 100 iterations. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In figure 9 one can see how the kernels were applied and in which order they execute.

Muyan-Orçelik et al. [MOOXS09] use CUDA in their implementation of the demons algorithm, a deformable registration algorithm widely used to match medical volumes. The test data used was 3D CT scans acquired using 4DCT, which obtains sequences of 3D CT data. The implementation used CUDA for numerous operations needed for the demons algorithm. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In all implementations, the algorithm is run for 100 iterations. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In figure 9 one can see how the kernels were applied and in which order they execute.

Muyan-Orçelik et al. [MOOXS09] use CUDA in their implementation of the demons algorithm, a deformable registration algorithm widely used to match medical volumes. The test data used was 3D CT scans acquired using 4DCT, which obtains sequences of 3D CT data. The implementation used CUDA for numerous operations needed for the demons algorithm. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In all implementations, the algorithm is run for 100 iterations. Each category of operations got their own CUDA kernel. The categories were:

- Gradients from the static volume.
- Trilinear interpolation.
- Displacement: produces the current total displacement of each voxel.
- Smoothing using a 3D gaussian filter.
- Correlation: computes the correlation coefficient.

In figure 9 one can see how the kernels were applied and in which order they execute.
3.3. Continuous scatterplots

A typical 2D scatterplot is data presented as a collection of points, where the points position on the x axis determine one variable and the position on the y axis another variable. A continuous scatterplot uses geometry instead of points, to better represent continuous underlying data [BW08].

To accelerate continuous scatterplots Bachtal et al. [BFW09] use CUDA in their implementation. To accelerate it on the GPU they had to make substantial changes to their code, due to the difference in architecture between the CPU and GPU. The changes consisted of avoiding complicated data structures and execution paths.

The computation itself consists of a preprocessing step on the CPU determining classes of tetrahedra. Due to the preprocessing step, the input for each CUDA kernel is a single tetrahedron of the same class. The CUDA kernel has to compute a topology for the four input vertices in order to construct triangles [BFW09]. Results are stored directly in a vertex buffer to be handled by OpenGL for display. The speed compared to the same calculation on the CPU can be seen in figure 11. In figure 12 an example of a continuous scatterplot can be seen.

3.4. Flow visualization

Cheong et al. [CSD+05] accelerate tracking of colloidal spheres, offering near real time performance using CUDA. The tracking is accomplished using image analysis on images obtained using holographic video microscopy. Images are interference patterns from light scattered and not scattered by the sample under scrutiny shown in figure 13. The GPU trial implementation by Cheong et al. were 20 times faster than an equivalent CPU based calculation. Operating at 8 frames per second, the implementation also showed scalability, jumping to 15 frames per second when using two GPUs.

3.5. Computation using OpenCL and CUDA

This section includes more computationally oriented uses of CUDA and OpenCL. In these papers the visualization step is not the focus, but rather the use of parallel computing power which indirectly opens new possibilities in visualization.

Stantchev et al. [SJDV09] uses CUDA to compute plasma turbulence simulations, and then rendering their data to screen using CUDA’s OpenGL interoperability functions. They compared their CUDA implementation to an implementation in Fortran on a 3.0 Ghz Intel Xeon processor. Their observations were increasingly larger speedup depending on the size of the input array, culminating at a factor of 14 at 1024 x 1024. In their future work, they mention wanting to moving from two to multi dimensional data, and visualizing it using volume rendering. Figure 14 shows 2D images of turbulence simulations.

Riabkov et al. [RXTC07] use the GPU to reconstruct and render medical X-ray data, their focus was on the computationally expensive reconstruction step from raw data to something useful. The goal was to accelerate the entire process of acquisition and rendering to near real time. Riabkov et al. used two experimental setups to test their implemen-
tations. Their implementations consist of one slice based reconstruction, and one based on projectional slices.

The experimental setups:
1. Intel P4 XE, 3.4 GHz, 4GB RAM, 800 MHz FSB, Nvidia Quadro FX4500 w/ 512MB, SUSE Linux 10.0, 32-bit
2. Intel dual-core Xeon (Woodcrest), 3.0 GHz, 8GB memory, 1333 MHz FSB, Nvidia GeForce 8800 GTX w/ 768MB, SUSE Linux 10.1, 64-bit Utilizing CUDA.

In figure 15 one can see a comparison of allocation and execution time between the two setups using the slice based reconstruction. The projection based reconstruction took around 5 seconds on the GeForce 8800 GTX, I have omitted the table showing this, since this was never run on the Quadro FX4500(not enough video memory) so no comparison can be made. Figure 16 shows slice views and a volume rendering accomplished by Riabkov et al. [RXT07] using this method on generated knee data.

<table>
<thead>
<tr>
<th>GPU / CPU</th>
<th>GPU memory (MB)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadro FX4500 / P4</td>
<td>347</td>
<td>20.7</td>
</tr>
<tr>
<td>GeForce 8800 GTX / Xeon</td>
<td>2.7</td>
<td></td>
</tr>
</tbody>
</table>

Figure 15: Comparison of allocation and execution time of experimental setups of slice based reconstruction [RXT07].

4. Conclusion

In this state of the art report, I have discussed the APIs for parallel computation. Parallel computing using CUDA has been adopted across a wide spectrum of areas in visualization. Comparisons of implementations against their CPU counterparts show huge gains in speed. Many implementations reaching goals of real time updates. Many implementations show scalability too, which considering Moore’s law give a bright outlook on future performance.

On another note, It became early apparent that there is a lack of literature on utilization of OpenCL in visualization and image processing. This is surprising considering the adoption of CUDA. Implementation in OpenCL, and especially implementations taking advantage of multiple CPU cores as well as the GPU would be areas for future work.

5. Acknowledgements

I want to thank Paolo Angelelli for his guidance while writing this STAR.

References


