Poster: CUDA-Accelerated Continuous 2D Scatterplots

Sven Bachthaler* Steffen Frey† Daniel Weiskopf‡

Visualization Research Center (VISUS), Universität Stuttgart

ABSTRACT

In this poster, we present how our previously published method of computing continuous 2D scatterplots can be performed with hardware acceleration on a GPU. By doing this, we exploit the parallel processing ability of current graphics hardware to improve the performance of continuous scatterplots by up to two orders of magnitude. For medium-sized tetrahedral data sets, we achieve interactive computation times.

1 INTRODUCTION

Data sets for scientific visualization are commonly defined continuously, e.g. by applying interpolation or reconstruction techniques to the data sampled on a grid. A helpful means for analyzing such scientific data is the well-known scatterplot. However, scatterplots only make use of the discrete data samples — they ignore the spatial relationship of neighboring data points and also the interpolated data between samples. This drawback was overcome by our recently published concept of continuous scatterplots [1]. Continuous scatterplots make use of continuously defined data by drawing the scatterplot in a dense way — instead of rendering discrete glyphs, the density of the data samples is drawn in the scatterplot domain. The original approach to compute a continuous scatterplot was implemented on the CPU. Depending on data set size, the time to compute a scatterplot may take up to several minutes — too long to efficiently use a continuous scatterplot for exploring a data set interactively. In this poster we present a modification of the original algorithm that enables the efficient construction of continuous scatterplots on the GPU. We process one tetrahedron per CUDA compute thread in a highly parallel fashion to perfectly suit the architecture of modern GPUs.

2 APPLICATION

A successful way of analyzing multi-dimensional data sets is to use a combination of scientific and information visualization techniques. Such a combined approach was presented by Doleisch et. al. [3] with their "SimVis" system. For the information visualization part, they use conventional scatterplots to visualize additional data dimensions and connect different views on the data set via brushing and linking. In such a system, continuous scatterplots can be easily integrated, since they are an extension to conventional scatterplots. In order to allow the user to interactively explore the data set, it is important that continuous scatterplots are recomputed rapidly — whenever the resolution or the focus within the scatterplot view is changed.

3 RELATED WORK

This poster is based on our original approach for computing continuous scatterplots [1]. In the original approach, the input data is provided on a tetrahedral grid. For each tetrahedron, a large number of computations is necessary — first, the topology of the resulting triangles is computed, then the maximum density of a tetrahedron is determined. These computations were performed on the CPU in a serial fashion, therefore consuming a considerable amount of time. In this original approach, the idea of Projected Tetrahedra [5] is used for rendering a continuous scatterplot. Therefore, a natural inspiration for our work is the paper by Wylie et. al. [7], which presents a way of projecting tetrahedra using vertex shaders. The main drawback of using a vertex shader for projecting tetrahedra lies in the fact that for each vertex the whole tetrahedron has to be processed. As shown by Wylie et. al. [7], each tetrahedron is represented by six vertices which leads to a six-fold computational overhead. Nowadays, graphic cards are much more flexible to program and we are no longer restricted to vertex shaders. Using CUDA, we get rid of the mentioned overhead since we can process tetrahedra in a step-by-step fashion on the GPU, creating the necessary vertices on the fly. Hardware-accelerated projection of tetrahedra was also implemented by Weiler et. al. [6]. However, the focus of Weiler's approach lies on volume rendering using shaders to perform ray casting. In addition, we proposed a different way of speeding up the computation of continuous scatterplots in [2]. In this previous work, a hierarchical approach was taken to approximate the correct continuous scatterplot. In contrast to the approach presented by our poster, our previous method was performed on the CPU and is restricted to uniform grids.

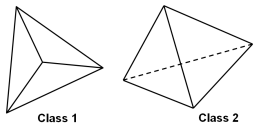

Figure 1: Two main classes of projected tetrahedra. Additional classes exist, but are only degenerate cases of these two classes.

4 ALGORITHM

In order to increase the compuational performance of continuous scatterplots, we use CUDA to execute a modified version of our original continuous scatterplot algorithm on the GPU. Directly porting the algorithm from CPU to GPU would be highly inefficient, due to the different architecture and programming model. On the GPU, we try to avoid complicated data structures and execution paths in order to achieve optimal performance. The modifications that are necessary to compute continuous scatterplots on the GPU are explained in the following paragraphs.

Our modified algorithm to compute continuous scatterplots is split into two parts: First, a preprocessing step is performed on the CPU. After this step, the actual computations that are necessary to render the continuous scatterplot are performed on the GPU.

To begin, we focus on the preprocessing step. For this, we introduce the notion of classes for tetrahedra. The original classification of tetrahedra was presented by Shirley and Tuchman [5]. However, for our approach, we use the simplification of this classification that was proposed by Wylie et. al. [7].

As in [7], we use only classes one and two for classification, since the remaining classes are only degenerate cases of these first two classes. In Fig. 1, the two main classes are shown for ref-

*e-mail: bachthaler@visus.uni-stuttgart.de

†e-mail: frey@visus.uni-stuttgart.de

‡e-mail: weiskopf@visus.uni-stuttgart.de

erence. To compute a continuous scatterplot, the density of each tetrahedron needs to be computed — which in turn requires a separate handling of tetrahedra based on their class. To achieve optimal performance of our density computations on the GPU, we sort the input tetrahedra in two lists which only contain tetrahedra of class one or class two respectively. This is necessary because on the GPU, diverging threads in a CUDA "warp" force the whole "warp" to serially execute each branch. In our case, this would result in significant computational overhead. Please note that sorting is a preprocessing step which is necessary only once per data set.

After this step, the actual computations for the continuous scatterplot are performed. Depending on the resolution of the data set, not all tetrahedra can be uploaded to the GPU at once. If this is the case, we split the list of input tetrahedra into data subsets that fit into GPU memory. A serialized processing of these data subsets is trivial since individual tetrahedra are rendered using additive blending — therefore, the resulting projected tetrahedra are accumulated in the framebuffer until all tetrahedra are processed. 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 for output. This computation of topology is based on the algorithm by Wylie et. al. [7]. This algorithm computes the triangle topology of the projected tetrahedra based on a series of tests applied to the four input vertices of a tetrahedron. In addition, the density of the input tetrahedron is determined in order to simulate the attenuation of light for the volume rendering. The main difference lies in the computation of this density; here we apply the formula presented in [1]. The resulting density is encoded as a color value and assigned to the corresponding vertices of a projected tetrahedra. We store the resulting triangles of the currently processed chunk in a vertex buffer object, which can be directly rendered using OpenGL in combination with shaders. We use these shaders since we store the color value of the vertices in their z-coordinate — for a 2-D continuous scatterplot, only two coordinates are necessary to specify the location in the scatterplot domain. The shader removes the color value from the z-coordinate and uses it as an input value for a color-lookup table which determines the final appearance of the continuous scatterplot.

5 Results

The main goal of this work is to speed up the computation of continuous scatterplots. In this section, we compare the original CPU implementation with our new GPU implementation. Both approaches are comparable since exactly the same data is processed, and the same result is produced. For the time measurements we used a PC with an Intel CPU running at 2.4 GHz and an Nvidia GeForce 8800 GTX graphics card. All continuous scatterplots were created for a viewport size of 1024×768 pixels.

We performed our measurements for three different data sets and visualized the results in Fig. 2. The first data set is the "Bucky Ball" data set, an artificial volume data set with a resolution of $32\times32\times32$. The data set itself contains scalar values, which we mapped to the horizontal axis of the continuous scatterplot. The second data dimension represents the magnitude of the gradient of the scalar values which are mapped to the y-axis. This is a common approach for transfer function design as presented by Kniss et. al. [4], since material boundaries appear as arc-like structures in the scatterplot. The "Bucky Ball" data set is very small, resulting in short waiting times for the continuous scatterplot computation. The CPU implementation needs 1.25s to finish, whereas the GPU version takes only 0.03s. The pre-sorting step for the GPU version, which is performed only once for a data set, takes 0.86s for this data set. Taking this into account, the GPU version is $1.4\times$ faster, however, once the pre-sorting is done, the GPU version is approximately $41\times$ faster.

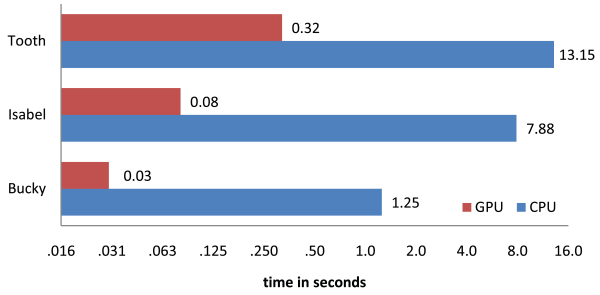

Figure 2: Comparison of CPU and GPU implementation for three data sets. We measured the time in seconds which is needed to compute a continuous scatterplot with identical parameters for both approaches. Please note that the horizontal axis uses a logarithmic scaling.

The second data set is "Hurricane Isabel", where several data dimensions are available at a resolution of $100\times100\times20$. We chose to map air temperature to the horizontal axis and air pressure to the vertical axis. For this data set, the CPU version takes 7.88s to compute a continuous scatterplot. The GPU version needs 0.08s to perform the same task, after a pre-computation step which takes 5.11s. Without the pre-computation step, the GPU version is approximately $98\times$ faster than the CPU version.

The third and last data set is a CT scan of a human tooth. This data set has a resolution of $64\times64\times80$. Again, we map the scalar values of the data set to the horizontal axis, the magnitude of the gradient of these scalar values to the vertical axis of the scatterplot. For this data set the CPU version needs 13.15s to produce the final result. The GPU version needs 8.46s for the pre-computation step and 0.32s for the actual computation of the continuous scatterplot. Without considering the pre-computation step, the GPU version is approximately $41\times$ faster than the CPU version.

6 Conclusion and Future Work

In this poster, we have presented a way of computing continuous scatterplots on the GPU using CUDA. The main benefit of this approach lies in the significant speed-up of the computation of a continuous scatterplot. For medium-sized data sets, we achieve interactive performance, which considerably increases the usefulness of continuous scatterplots in visualization systems. Since directly porting the original algorithm for continuous scatterplots is not feasible, we have proposed a modified version that takes the different architecture and programming model of GPUs into account.

References

[1] S. Bachthaler and D. Weiskopf. Continuous scatterplots. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1428–1435, 2008.

[2] S. Bachthaler and D. Weiskopf. Efficient and adaptive rendering of 2-d continuous scatterplots. *Computer Graphics Forum*, 28(3):743 – 750, 2009.

[3] H. Doleisch, M. Gasser, and H. Hauser. Interactive feature specification for focus+context visualization of complex simulation data. In *In Proc. IEEE Symposium on Visualization*, pages 239–248, 2003.

[4] J. Kniss, G. Kindlmann, and C. Hansen. Multi-dimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[5] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, 1990.

[6] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization '03*, pages 333–340, 2003.

[7] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *In Proc. IEEE Symposium on Volume Visualization and Graphics*, pages 7–12, 2002.