Contouring for Power Systems Using Graphical Processing Units

Joseph Euzebe Tate
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
jetate1@uiuc.edu

Thomas J. Overbye
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
overbye@ece.uiuc.edu

Abstract

To improve situational awareness in power systems, one useful tool used in control centers is bus (or substation) data contouring. Traditionally, the methods developed have used CPU processing, leading to long contour rendering times that reduce interactivity with the visualization. To improve interactivity and increase the data rate which can be handled, contouring methods utilizing graphical processing units (GPU's) show much promise. This paper proposes a GPU-based contouring algorithm which can easily outperform state-of-the-art CPU-based contouring algorithms. In addition, sample rendering times for a typical power system display, along with comments on the relative advantages and disadvantages of using the CPU and GPU to perform contouring, are provided.

1. Introduction

Based on the recommendations of the August 2003 blackout report [1], a heavy emphasis has been placed on improving situational awareness in control centers around the world. One visualization technique which has been used to increase situational awareness has been contouring [2]. This technique, also known as scattered data interpolation (see [3] for a survey of the state-of-the-art), aims to provide a weather map-like visualization, showing various bus- or substation-based data throughout the area of interest. This type of visualization has been used in many disparate fields, e.g., meteorology [4] and medical imagery [5], and has been shown to be particularly useful for power systems via human factors testing [6].

One key similarity between the methods used to generate power system contours in the past has been a reliance on the CPU to do the necessary contour calculations. CPU's, which are optimized for general purpose computing, are not well-suited for the parallel calculations that are inherent in some contouring algorithms. Several incremental improvements have been made, but CPU-based contouring is fundamentally constrained by the serial nature of CPU program execution (see section 6 for more detail). On the other hand, graphics processing units (GPU's), which are present in most modern computer systems, are ideally suited to the computations needed to generate contours because of the parallel nature of GPU program execution. Furthermore, some work has already been done in demonstrating the effectiveness of using GPU's to perform real-time interpolation of three- and four-dimensional data [7]. This work suggests that there is a potentially large benefit to using GPU's in two-dimensional power system contours.

In order to investigate the possible performance gains of using GPU- rather than CPU-based contouring for power system visualizations, we have developed a prototype GPU-based contouring algorithm. To begin, a quick overview of contouring is provided in section 2. Section 3 discusses some of the fundamental aspects of GPU programming, while section 4 provides the algorithm and implementation details. Section 5 provides some performance results for contouring a real power system area as parameters such as influence radius and contour resolution are varied. Section 6 discusses some of the key advantages and disadvantages of moving contouring to the GPU, and sections 7 and 8 provide conclusions and future areas of research that should be explored.

2. Contouring background

Formally, the purpose of contouring (or scattered data interpolation) is to find a function $F(x,y)$ such that for each bus k, located at position (x_k, y_k):

$$F(x_k, y_k) = f_k \quad (1)$$

, where f_k is the value to be contoured (e.g., voltage), known to be a specific value at each bus or substation k. This value is typically obtained from a state estimator or directly from a SCADA measurement. There are an infinite number of possible functions that can satisfy this constraint, so additional restrictions are useful in reducing the set of possible functions. One additional constraint we impose on F is that it be "smooth", i.e., continuous and once differentiable [8].

While there are many different methods used to determine the contouring function F (see [3]), our work

focuses on inverse distance weighted methods due to their intuitive nature and ease of parallelization. The basis of inverse distance weighted methods is that the function value at any point should be the weighted average of all points with known values. In this method of contouring, the function F is defined as follows:

$$F(x,y) = \frac{\sum_{k=1}^{N} w_k(x,y) f_k}{\sum_{k=1}^{N} w_k(x,y)} \quad (2)$$

, where $w_k(x,y)$ is known as the weighting function. The weighting function first proposed for this scheme, which gives rise to the contouring algorithm known as Shepherd's Method ([9]) is:

$$d_k = \sqrt{(x-x_k)^2 + (y-y_k)^2}$$
$$w_{k,SM}(x,y) = \frac{1}{d_k^2} \quad (3)$$

This weighting function gives a nonzero value of $w_k(x,y)$ for all values of x and y. Because each bus k has a non-zero contribution to every point on the screen, using the weighting function of equation (3) is categorized as a global contouring algorithm.

Global methods such as Shepherd's Method are computationally burdensome, due to the need to consider each bus' contribution at each pixel on the screen. As a result, several local methods have been proposed [8]. For our algorithm, we use the local weighting function developed by Franke and Little [10]:

$$w_{k,FL}(x,y) = \begin{cases} \left[\frac{(R-d_k)}{Rd_k}\right]^2 & , d_k < R \\ 0 & , d_k \geq R \end{cases} \quad (4)$$

, where d_k is defined in (3). For this local weighting function, only pixels within a distance R of bus k will be affected by the value f_k. This weighting function maintains continuity and first-order differentiability of the interpolating function F [8], and reduces to the weighting function of (3) as $R \to \infty$.

3. GPU programming [11]

3.1 The rendering pipeline

GPU programming is significantly different from CPU programming, and this must be taken into account when adapting any algorithm for GPU implementation. The basis GPU pipeline, illustrated in Figure 1, was originally constructed to efficiently render polygons, lines, and points to a graphical display. The basic steps in the GPU rendering pipeline are:

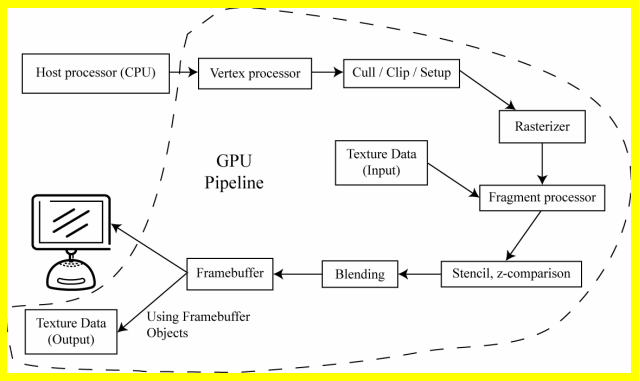

Figure 1. The GPU rendering pipeline

1. Polygon vertex coordinates, viewable area definitions, color values, texture coordinates, and textures are sent to the GPU via OpenGL calls from the CPU.
2. The vertex processor transforms coordinates defined via function calls from the CPU in step 0 into coordinates relative to the display. Also, vertices are grouped into primitives (points, lines, and triangles) for the later stages of the pipeline.
3. The Cull / Clip / Setup stage takes the primitives (polygons, lines, and points) that come out of the vertex processor and removes unviewable objects. Also, any primitives that are only partially inside the viewable area are clipped.
4. The rasterization stage determines which fragments are covered by each primitive. Fragments are the rasterized pieces of the polygon and correspond to a single pixel of the final output. Also, per-vertex values such as color values and texture coordinates are linearly interpolated across each element for each fragment within the polygon.
5. The fragment processor runs on each fragment, performing texture lookups and determining the final color that the fragment will be. For the contour implementation in this paper, two non-default fragment processors are used, as explained below in sections 4.2-4.4.
6. Once fragment colors have been assigned by the fragment processor, several tests (e.g., stencil and z-compare) are run on the fragments to see if they should be discarded. If the pixel passes all tests, then it is blended with the current framebuffer entry at its location and the final pixel value is written to the framebuffer. This write to the framebuffer is the final stage of the OpenGL pipeline. For traditional graphics applications, the framebuffer is essentially the pixels on the current video display. Framebuffer

objects (FBO's) make it is possible to draw onto a texture in video memory rather than to the display [12] . This texture can then be read back on a subsequent rendering pass. This technique is used extensively in our implementation, as discussed below.

3.2 Custom fragment processors

The only truly programmable units on the GPU are the vertex processor and the fragment processor, which allow for custom processing of vertices and fragments in stages 2 and 5. We use the default vertex processor, so it is not discussed here. On the other hand, we use two custom fragment processors for contour generation.

One very important aspect of fragment processor programming is that there can be fragments at different stages of the pipeline at any given time. As a result, the fragment processor is not allowed to write to any locations in video memory except for the framebuffer at the end of stage 5. Otherwise, each fragment might have to wait for another fragment to complete its processing before it could be sent out to the framebuffer. If we think of the framebuffer as an array of bytes, then the fragment processor can only output data to one particular array index in the framebuffer. Although it cannot write directly to texture memory, the fragment processor is capable of reading directly from texture memory. Because of these features, it is easiest to think of a fragment processor as having a large set of memory addresses it can read from, and only one set of memory addresses it can write to, with no overlap between these two address spaces. This is fundamentally different from CPU programs, where programs can easily read and write to the same memory addresses.

3.3 Computational power of fragment processors

Although fragment processors must be programmed with a rather restrictive set of operations, particularly when it comes to memory reads and writes, sheer processing power makes them ideal for speeding up computations that can be performed in parallel. For instance, modern GPU's such as the NVIDIA GeForce 7-series card used in our experiments are capable of performing over 165 billion floating point operations per second (gigaflops or Gflops), whereas a dual-core Pentium 4 is only capable of around 24.6 Gflops [13]. More advanced chips, such as the recently released GeForce 8800 GTX, have theoretical processing speeds above 500 Gflops. For cases where the GPU can be run near its processing limits, this difference in processing power can make GPU's easily outperform CPU's.

3.4 Using the computational power of the fragment processors [14]

There are two key issues involved in fully taking advantage of the GPU's computation power. First, textures should be read as sequentially as possible in order to take advantage of texture caches. We have designed our algorithm to make coherent texture reads, primarily by making all texture reads from rectangular textures, where each texel is accessed starting from the top-left and moving sequentially to the bottom-right.

The other key issue in programming for the GPU is to use algorithms which are limited by computation time. Contouring algorithms certainly fall into this category, because the bulk of the time spent in contouring is when (4) is evaluated for each pixel in the influence region of each bus. Also, because GPU's work best when performing parallel computations, the use of for…next loops in the algorithm definition allows for easy conversion to a fragment processor program.

4. GPU-based contouring

4.1 Algorithm definition

Based on equations (2) and (4), taking into account the comments made in section 3, an algorithm well-suited to GPU processing for constructing a contour is:

A-1. For each point (x, y)

A-1.1. $N(x, y) = 0$

A-1.2. $D(x, y) = 0$

A-1.3. $F(x, y) = 0$

A-2. For each bus k

A-2.1. For each point (x, y) s.t. $w_{k,FL}(x, y) \geq 0$

A-2.1.1. $N(x, y) += w_{k,FL}(x, y) f_k$

A-2.1.2. $D(x, y) += w_{k,FL}(x, y)$

A-3. For each point (x, y) s.t. $D(x, y) > 0$

A-3.1. $F(x, y) = \dfrac{N(x, y)}{D(x, y)}$

A-3.2. For display to the screen, each value of F is mapped to a color as discussed in [2] and stored as the color value for point (x, y).

Each of the algorithm steps given above can be translated into a CPU or GPU set of instructions, as shown in Table 1. For the GPU implementation we have developed, the C programming language was used with the GLUT library [15] for OpenGL programming and the Cg language was used for fragment processor programming [16]. The implementation of the algorithm was split into three steps, explained in detail below.

Table 1. Algorithm implementation on the CPU and the GPU

Algorithm Steps	CPU Implementation	GPU Implementation
A1	Allocate space for *N* and *D* in main system memory.	Allocate space for *AccumTexture* and *ContourTexture* on the GPU.
A2	For each bus, calculate and store the numerator and denominator values within each bus' influence region.	Draw a circle of radius *R* around each bus with the video output redirected to *AccumTexture*. Use a custom fragment shader to calculate and store the *N* values in the red channel and the *D* values in the green channel. Use additive blending to perform accumulation as each bus is drawn.
A3	Iterate through each element in the *N* and *D* arrays and divide the numerator by the denominator. Look up the corresponding color in the color map and write this color to a bitmap.	Draw a rectangle of size *Res* with its texture set to *AccumTexture* and the video output redirected to *ContourTexture*. Using a custom fragment shader, divide the red channel values of *AccumTexture* by the green channel values. For each pixel, look up the corresponding color in the color map and write this color to *ContourTexture*.

4.2 Implementation step one – allocation of textures and framebuffer objects

To begin, texture memory is allocated for two textures: *AccumTexture* and *ContourTexture*. *AccumTexture* is created as a GL_RGBA16F_ARB formatted texture, which allocates four components (red, green, blue, and alpha) for each texture element (texel). Each component stores a IEEE-formatted 16-bit floating point value (see [14] for more details on this format). *ContourTexture* is allocated as a standard 32-bit texture containing 8-bit red, green, blue, and alpha components for each texel. *AccumTexture* stores $N(x,y)$ in its red channel and $D(x,y)$ in its green channel. *ContourTexture* stores the finished, color-mapped contour as a texture which can then be rendered to the display. Both textures are created with the same dimensions, denoted as *Res*. This size is the resolution of the contour to be created (e.g., if a contour is to be created which exactly fits a 640x480 window, then the two textures will have dimensions 640x480). Because the algorithm requires rendering to these textures rather than simply reading from them, a framebuffer object is created which allow the GPU to write to the textures [12]. The framebuffer objects used to write to each texture are denoted as *AccumFBO* and *ContourFBO*. The allocation of texture memory occurs only when desired contour resolution changes (typically when a window is resized), which reduces the amount of time spent allocating memory on the GPU.

4.3 Implementation step two – accumulation of numerator and denominator values

Once the textures are allocated, *AccumFBO* is set as the render target for the GPU. By setting *AccumFBO* as the render target, all pixels drawn by the GPU are sent to *AccumTexture* rather than the display. After *AccumFBO* is attached, *glClear*() is called, which performs steps A-1.1 and A-1.2 by clearing the red and green color channels of *AccumTexture*. Next, a custom fragment processor is bound to the GPU that takes as arguments the position of the bus that is currently being drawn, (x_k, y_k), and writes out the values $w_{k,FL}(x,y)f_k$ to the red channel and $w_{k,FL}(x,y)$ to the green channel. The final setup step is to enable additive blending on the GPU to perform the assignment by sum ($+=$) operations needed in steps A-2.1.1 and A-2.1.2. Once all this setup has taken place, for each bus k, a circle of radius R is drawn around the point (x_k, y_k). After this step is completed, the red channel of *AccumTexture* contains N and the green channel contains D for all points in the contour.

4.4 Implementation step three – evaluation of interpolating function and color mapping

For the next stage of the contouring process, *ContourFBO* is attached as the render target for the GPU. A call to *glClear*() is then made to execute step A-1.3. Next, *AccumTexture* is set as the current texture to be read from in the fragment processor of the GPU. A custom fragment program is then bound to the GPU which performs steps A-3.1 and A-3.2. Finally, a quadrilateral of size *Res* is drawn with texture

coordinates such that each point in *AccumTexture* corresponds to one output fragment. Once this draw operation has completed, *ContourTexture* contains the color-mapped contour. This texture can then be drawn to the screen by binding *ContourTexture* as the current texture and drawing a quadrilateral with appropriate texture coordinates.

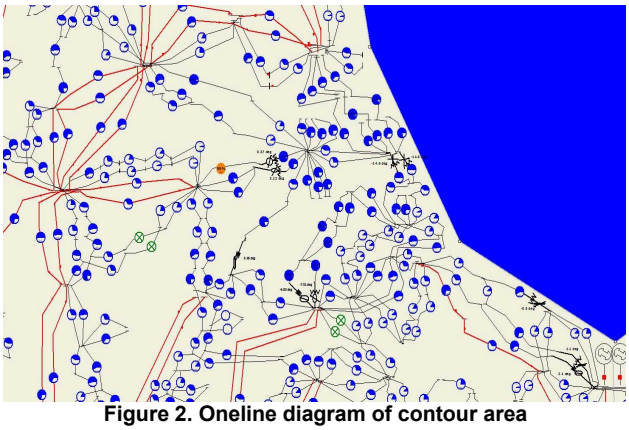

Figure 2. Oneline diagram of contour area

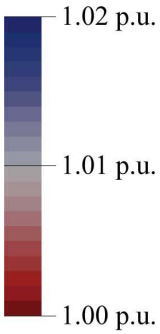

Figure 3. Contour color map relating voltage per unit magnitude to displayed color

5. GPU-based contouring algorithm performance results

For all of the results given below, a test system with an AMD Athlon 64 X2 Dual Core 3800+ CPU and an NVIDIA GeForce 7600 GT GPU was used. Timings for the GPU algorithm were determined using the high-resolution Windows performance counter [17], with an average taken over 100 contour renderings.

The power system one-line diagram used as the basis for the results and figures below is shown in Figure 2. The number of buses that influence the contour area range from 199 (for an influence radius of 1) to 818 (for an influence radius of 400) as shown in Table 3 below.

The data that is contoured is the per unit voltage level at each bus. The color map used to convert voltage levels to colors is shown in Figure 3.

5.1 The effect of changes in *Res*

For the algorithm defined in section 3, there are two key parameters which can potentially affect performance. The first parameter, *Res* , controls how large of a contour image to create. For the highest-quality contour, *Res* should be set to the same size as the screen area the contour covers upon final rendering to the display. However, reducing *Res* to a smaller value results in a decrease in rendering time, so a tradeoff must sometimes be made between contour resolution and rendering speed. To explore the effects of varying *Res* , the system was contoured with different resolution levels. In these tests, *R* , the radius of influence, was held at a constant value of 100.

Table 2 shows the timing results obtained for various contour resolutions. In order to gauge the performance of the algorithm in a typical industrial setting, we tested the most common resolutions in use today on both projectors and conventional displays.

Table 2. Timing results for several contour resolutions with radius of influence set to 100

Res			
Width	Height	Common Name	Time to render a contour (seconds)
3840	2400	WQUXGA	0.938
1920	1200	WUXGA	0.249
1600	1200	UXGA	0.208
1280	1024	XGA+	0.150
1024	768	XGA	0.100
800	600	SVGA	0.066
640	480	VGA	0.050

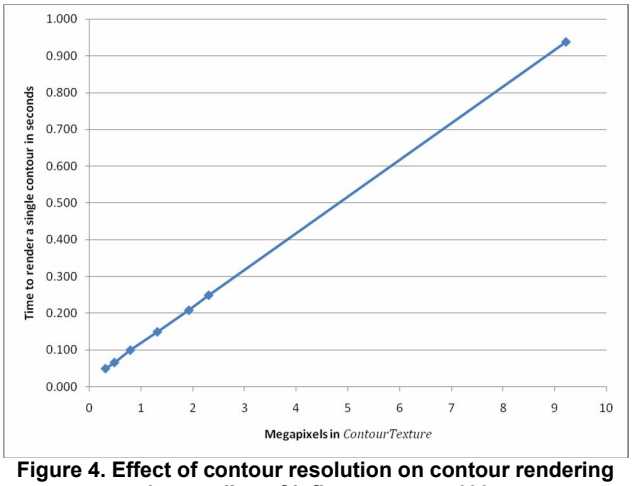

Figure 4. Effect of contour resolution on contour rendering time, radius of influence set to 100

Figure 4 illustrates the effect on rendering time as the contour resolution is varied. The x-axis is the number of megapixels associated with each tested resolution and the

y-axis is the average amount of time it took to render one contour. The linear relationship between resolution and rendering time indicates that the time required to render a particular contour is on the order of the number of pixels in the contour. The fact that this linear relationship holds for resolutions ranging from 640x480 all the way up to 3840x2400 indicates that the algorithm does not suffer from any scalability issues.

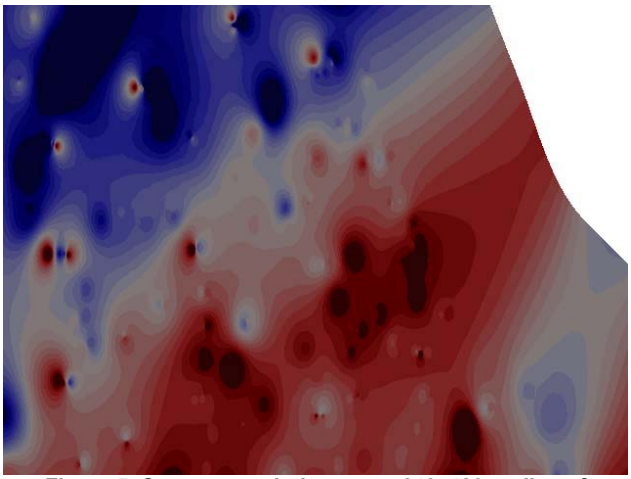

Figure 5. Contour resolution set to 640x480, radius of influence set to 100

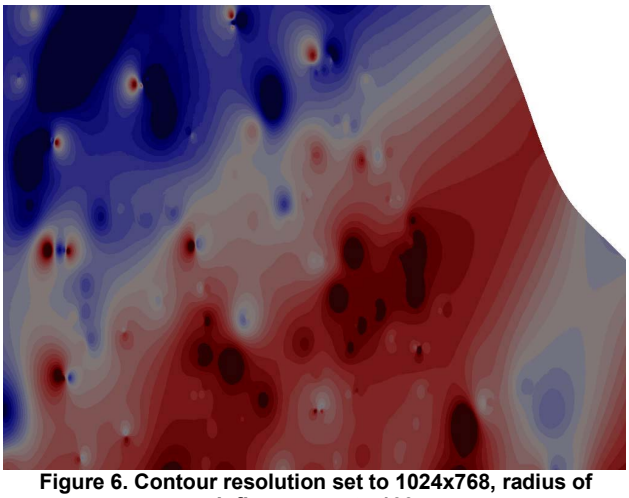

Figure 6. Contour resolution set to 1024x768, radius of influence set to 100

It is also worthwhile to consider the visual impact of changing the contour resolution. For a given screen size Res_{Screen}, the highest quality contour would be obtained by setting $Res = Res_{Screen}$, assuming the contour takes up the entire screen. However, by setting $Res < Res_{Screen}$, the speed at which the contour is rendered can be greatly accelerated. One obvious option, then, is to first render the contour at a lower resolution, e.g., 640x480, and then to gradually increase the resolution.

One other key benefit of being able to specify the contour resolution independently of the screen resolution is that using a lower resolution is not necessarily noticeable. Figure 5 and Figure 6 illustrate this fact—it is difficult to discern any differences between the images when rendered to the screen, even though Figure 6 takes over twice as long to render as Figure 5. The resulting appearance of a low resolution contour is also influenced by the method of texture filtering used when the contour is finally drawn to the screen. Ultimately, the tradeoffs between resolution and rendering time should be decided upon by the person viewing the contour; accordingly, we have made every effort to allow for easy modification of this key parameter within our implementation.

5.2 The effect of changes in R

The other key parameter which affects contour accuracy and timing is the radius of influence, R, of each bus. Because a circle of radius R is drawn around each bus (i.e., each bus affects a locus of points of radius R around its location), the contour rendering time should scale in proportion to the area of the circle drawn around each bus, i.e., R^2, assuming the contour area is infinite. However, for a contour with finite area, this relationship does not always hold.

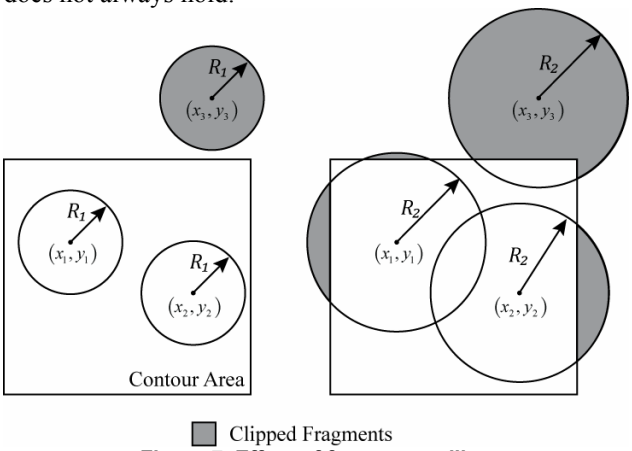

Figure 7. Effect of fragment culling

Figure 7 illustrates a simple three-bus contouring example where the relationship between R and the number of processed fragments is not quadratic. The left side of the figure shows the case where the circle of radius R_1 drawn around buses 1 and 2 fits entirely within the contour area, whereas the circle around bus 3 is entirely outside the contour area. Because the bounding box which encloses the circle drawn around bus 3 does not intersect the contour area, there is no need to draw the circle around bus 3. On the right side of the figure, as the radius is increased to R_2, the circles surrounding buses 1, 2, and 3 are partially inside of the contour area. In this

case, a circle is drawn around each bus, but the GPU is able to perform clipping at full speed [11] so that no time is wasted in calculating steps A-2.1.1 and A-2.1.2 for the shaded pixels outside of the contour area. Based on Figure 7, we see that a linear increase in the influence radius will cause a quadratic increase in computation time only if the set of buses influencing the contour area does not change and all bus influence regions are entirely enclosed in the contour area. Otherwise, the increase in computation time is specific to the characteristics of the area being contoured and the distribution of the buses.

To examine the effect of changing the influence radius, several timing measurements were taken with a fixed contour resolution of 1024x768, letting R vary from 1 to 400. The results of these timing measurements are given in Table 3. In addition, Figure 8 illustrates the relationship between the radius of influence and the time to construct a contour.

For the first few values—1, 5, and 10—the time to render a contour is essentially constant. This indicates that for very low influence radii, the time spent rendering a contour is based more on other factors (such as texture lookup, OpenGL calls, etc.) than on the processing time needed to construct the contour. As the influence radius increases to values between 10 and 150, a quadratic relationship is seen. For radii above 150, a much more complicated relationship is demonstrated. This is due to the relationship between the bus influence regions and the contour area, as explained above and illustrated in Figure 7.

Table 3. Timing results for several values of radius of influence with a constant contour resolution of 1024x768

R	Number of buses influencing the contour area	Time to render a contour (seconds)
1	199	0.0225
5	206	0.0227
10	221	0.0229
25	274	0.0249
50	367	0.0361
75	438	0.0665
100	485	0.1001
125	514	0.1331
150	580	0.1838
200	653	0.2669
300	753	0.4251
400	818	0.4994

As with the other tunable parameter, *Res*, there is a tradeoff between how large of an influence radius is desired and how fast the contours can be rendered. Unlike the *Res* parameter, however, it is not quite as useful to start with a small value of R and work up to a specific value, because there is no obvious upper bound on what the radius of influence should be. Instead, the best value of R must be determined for each specific contour application—for dense one-line diagrams, a smaller value of R is probably more appropriate than for a sparse diagram. This is one justification for the use of dynamic influence regions when contouring power system data, as discussed in [18].

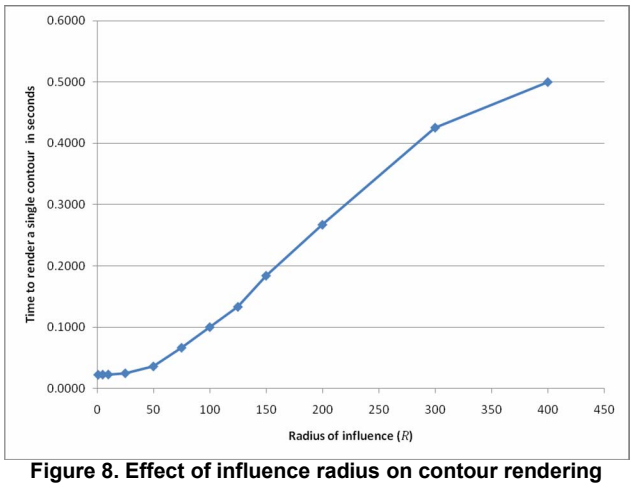

Figure 8. Effect of influence radius on contour rendering time, contour resolution set to 1024x768

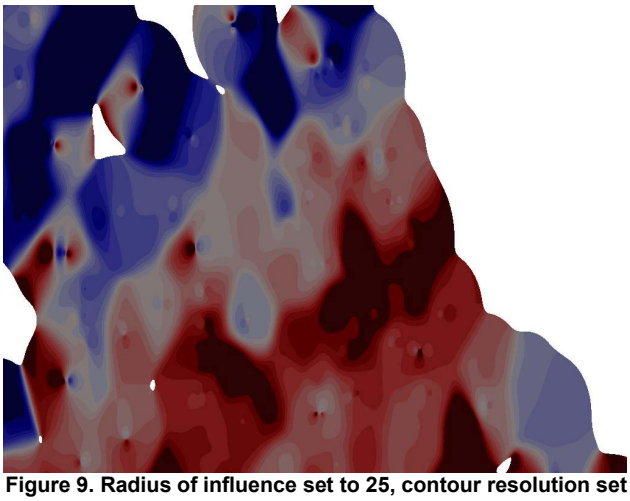

Figure 9. Radius of influence set to 25, contour resolution set to 1024x768

To illustrate the visual impact of changing the radius of influence, Figure 9 and Figure 10 show the contours obtained with the radius of influence set to 25 and 50, respectively. In Figure 9, it is very easy to distinguish the circles that surround each bus, and the silhouette of the contour is a series of arc segments. This is the typical silhouette obtained when the radius of influence is set to a relatively small value. On the other hand, in Figure 10, a much smoother contour is shown. No obvious circles are present in this contour, and further increases in R have a small impact on the contour (compare to Figure 6, which has R set to 100). Ultimately, the person using the

contour must decide on which value of R provides a good representation of the underlying data without sacrificing too much in terms of rendering speed.

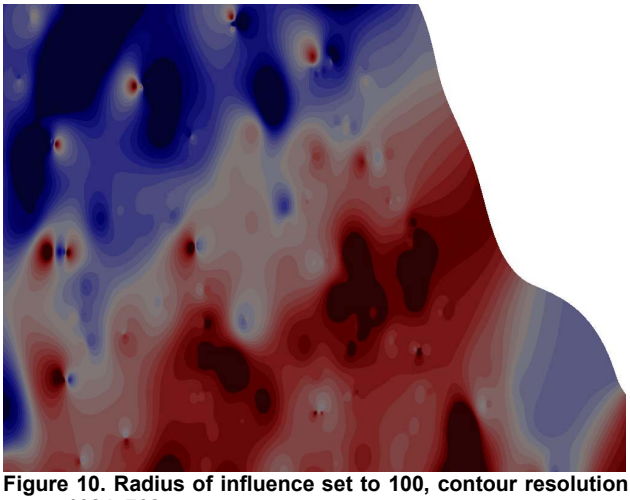

Figure 10. Radius of influence set to 100, contour resolution set to 1024x768

6. Advantages and disadvantages of GPU-based contouring versus CPU-based contouring

6.1 Advantages of GPU-based contouring

Because implementation of a GPU-based contouring algorithm can require significant expenditure of time and resources, it must be justified as an alternative to current CPU-based contouring methods. The greatest benefit of moving contouring to the GPU is the decrease in rendering time for contours. For example, contouring the power system area from Figure 2 using a resolution of 640x640, with a radius of influence of 150, the GPU-based contour takes 0.102 seconds to render; rendering the same contour using the CPU (with the method outlined in [18]) takes 5.10 seconds.

The rendering speeds of CPU-based contouring methods are significantly slower than the GPU-based algorithms for several reasons. First, CPU's are designed to process instructions in a serial fashion, which makes them ill-suited for parallel computations. GPU's, on the other hand, have been built from the start to process multiple vertices and fragments in parallel in order to quickly render polygons to the screen. For instance, the NVIDIA GeForce 7600 GT GPU used in the above tests has 12 fragment processors, each capable of performing steps A-2.1.1 and A-2.1.2 independently. On a CPU, each of these operations must be performed serially. Even with a multi-core CPU, each core still processes all instructions serially. Second, CPU's typically only have a 6.4 GB/sec memory bandwidth, compared to the 32 GB/sec (and higher) memory bandwidth available to modern GPU's [11]. As a result, reading and writing to memory can occur at a much faster rate on the GPU. Finally, the clipping operations built into GPU's are able to efficiently eliminate calculations which do not influence the contour plot (e.g., calculations on the shaded pixels in Figure 7). On the other hand, when using a CPU to perform contouring, the set of contour points influenced by each bus must be explicitly calculated.

Another key benefit to using GPU's for contouring is that it frees up the CPU to perform tasks more appropriate to the CPU. For instance, while the CPU is used to calculate load flows or perform other types of system analysis, the GPU can independently construct and render a contour without any CPU interaction.

Finally, because the rendering times for contours using GPU's are very short, it should be possible to display data that arrives at a much faster rate than if CPU-based contouring is used. Using the aforementioned rendering times of 0.102 seconds and 5.10 seconds for the GPU- and CPU-based contouring, a CPU-based contouring algorithm with a 640x640 resolution and influence radius of 150 could only keep up with a data rate of 0.2 samples per second, whereas the GPU-based method could keep up with a data rate of 10 samples per second. CPU-based methods have been used for state estimator visualization, because the incoming data rate is (at most) once per minute, or 0.02 samples per second. On the other hand, CPU-based contouring of SCADA data, which is typically sampled every two or three seconds, can require significant reduction in influence radius and resolution in order to keep up with the data rate. Moving forward, the realtime visualization of phasor measurement unit data, which comes in at a rate of 30 samples per second, is outside the realm of CPU-based contouring and is a clear case where GPU-based contouring should be used instead.

6.2 Disadvantages of GPU-based contouring

GPU-based contouring has several disadvantages when compared to CPU-based contouring. The greatest disadvantage to using GPU-based contouring is the potential loss in accuracy due to floating point errors. Although many of the latest GPU's support 32-bit operations on the chip, using 32-bit floating point typically incurs a performance penalty relative to 16-bit operations. Another potential difficulty with using 32-bit floating point textures is that blending of 32-bit floating point values, needed to implement steps A-2.1.1 and A-2.1.2, is only possible on the very latest video cards (e.g., the NVIDIA GeForce 8-series video cards). As a result, all of the calculations in our implementation are performed using 16-bit floating point numbers. This can lead to numerical inaccuracies due to the limits on 16-bit

precision, which is a problem that CPU's, which work on 32- or 64-bit values, do not have.

Another potential problem with using GPU-based contouring is the wide variety of capabilities of GPU's. Although there are some standards for GPU capabilities as defined by the OpenGL Architecture Review Board, it is nontrivial to write a GPU contouring program which takes advantage of each type of GPU equally. For instance, writing to textures with an NVIDIA GPU is optimized for a different set of instructions than an ATI GPU. CPU's do not typically suffer from this difficulty—it is generally safe to assume that an AMD and an Intel chip will do the same thing when sent the same instructions, and compilers optimized for one typically work well for the other.

7. Future work

There is substantial future work to be explored in the area of GPU-based contouring. Different forms of interpolating functions may provide a faster or more accurate data representation than that obtained using this algorithm. Also, there are likely several ways to further optimize the fragment processor code used to perform the algorithm steps. The use of advanced techniques such as dynamic influence regions ([18]) could also have a beneficial impact on GPU-based contouring. Another avenue to explore would be the human factors impacts of having much faster contour rendering times. Finally, measuring the performance of this algorithm with newer cards supporting 32-bit blending would also be worthwhile, as the only possible drawback of using the GPU currently is the potential for graphical artifacts to occur due to low numerical precision.

8. Conclusions

The fundamental role of data visualization in promoting situational awareness is to provide timely, accurate information. By performing contouring calculations on the GPU rather than the CPU, data can be visualized in at a much faster rate, leading to more timely data visualization and the potential for greater interactivity with the data display. Although each application will require a careful weighing of the advantages and disadvantages of a CPU or GPU implementation, it is clear that GPU-based contouring is a viable alternative that can greatly benefit situational awareness in power systems.

9. References

[1] (2003) Final Report on the August 14, 2003 Blackout in the United States and Canada. U.S.-Canada Power System Outage Task Force. [Online]. Available: https://reports.energy.gov.

[2] J. D. Weber and T. J. Overbye, "Voltage contours for power system visualization," IEEE Trans. Power Systems, vol. 15, no. 1, Feb. 2000, pp. 404-409.

[3] I. Amidror, "Scattered data interpolation methods for electronic imaging systems: a survey," Journal of Electronic Imaging, vol. 11, no. 2, Apr. 2002, pp. 157-176.

[4] W. A. Nuss and D. W. Titley, "Use of Multiquadric Interpolation for Meteorological Objective Analysis", Monthly Weather Review, vol. 122, Jul. 1994, pp. 1611-1631.

[5] T. M. Lehmann, C. Gonner, K. Spitzer, "Survey: interpolation methods in medical image processing," IEEE Trans. Medical Imaging, vol. 18, no. 11, Nov. 1999, pp. 1049-75.

[6] T. J. Overbye, D. A. Wiegmann, A. M. Rich, Y. Sun, "Human factors aspects of power system voltage contour visualizations," IEEE Trans. Power Systems, vol. 18, no. 1, Feb. 2003, pp. 76-82.

[7] S. W. Park, L. Linsen, O. Kreylos, J. Owens, B. Hamann, "A Framework for Real-time Volume Visualization of Streaming Scattered Data," Proc. of 10th International Fall Workshop on Vision, Modeling, and Visualization 2005, Erlangen, Germany, pp. 225-232.

[8] R. J. Renka, "Multivariate Interpolation of Large Sets of Scattered Data," ACM Trans. on Mathematical Software, vol. 14, no. 2, Jun. 1988, pp. 139-148.

[9] D. Shepard, "A two-dimensional interpolation function for irregularly-spaced data," Proc. of the 23rd ACM National Conference, New York, NY, 1969, pp. 517-523.

[10] R. E. Barnhill, "Representation and approximation of surfaces," Mathematical Software III (J. R. Rice, Ed.), Academic Press, New York, 1977, pp. 69-120.

[11] E. Kilgariff, R. Fernando, "The GeForce 6 Series GPU Architecture," GPU Gems 2 (M. Pharr, Ed.). Addison-Wesley, Upper Saddle River, NJ, 2005, pp. 471-491.

[12] S. Green, "The OpenGL Framebuffer Object Extension," Game Developers Conference 2005. [Online]. Available: http://http.download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf.

[13] D. Geer, "Taking the graphics processor beyond graphics," Computer, vol. 38, no. 9, Sept. 2005, pp. 14-16.

[14] I. Buck, "Taking the Plunge into GPU Computing," GPU Gems 2 (M. Pharr, Ed.). Addison-Wesley, Upper Saddle River, NJ, 2005, pp. 509-519.

[15] M. Kilgard, The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3. [Online]. Available: http://www.opengl.org/documentation/specs/glut/spec3/spec3.html.

[16] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like

language," ACM Trans. on Graphics, vol. 22, no. 3, pp. 896-907, Jul. 2003.

[17] J. D. Meier, S. Vasireddy, A. Babbar, and A. Mackman, "How To: Time Managed Code Using QueryPerformanceCounter and QueryPerformanceFrequency," Microsoft Developer Network, May 2004. [Online]. Available: http://msdn2.microsoft.com/en-us/library/ms979201.aspx.

[18] D. Savageau, T. J. Overbye, "Adaptive Influence Distance Algorithm for Contouring Bus-Based Power System Data," 40th Annual Hawaii International Conference on System Sciences, Jan. 2007, pp. 117-117.