Improving Hard Drive Data Visualization

Therin Irwin
Cal Poly San Luis Obispo
tcirwin@calpoly.edu

Trevor DeVore
Cal Poly San Luis Obispo
tdevore@calpoly.edu

Wyatt Kessler
Cal Poly San Luis Obispo
wyatt.kessler@gmail.com

ABSTRACT

Effectively analyzing hard drive statistics is incredibly important to anyone in the industry. Unfortunately, there are few tools dedicated to this task, and the few that are out there don't make visualizing the data easy. Stephen Pungdumri's *Storage System Information Visualizer* is no exception. His tool is easy to use and effective when a small number of statistic points are used - 20 to 50 - but the usability of the software degrades as many more are added.

Users of the software desire to graph large quantities of data because of the sheer number of requests made to the hard drive every second. One solution explored in this paper is to graph *splines*, instead of straight lines, between statistics so that the user can easily follow a data point through each statistic.

1. Introduction

1.1 Goals

The primary goal of this project is to increase the readability of the Storage System Data Visualizer. This system takes in a list of hard drive statistics, with 14 data points each, and displays them graphically so the user can see trends in the hard drive's performance. Originally, the system simply plotted the 14 points and drew a straight line between each point. Although this displayed all the necessary information, the resulting graph was extremely confusing. To remedy this problem, we fit a curve to each hard drive statistic. This makes the graph much easier to read.

Using curves for the Data Visualizer makes the graph significantly slower, but unfortunately also makes the rendering time significantly slower. Currently, rendering the graphs with splines is about ten times slower than rendering the graphs with straight lines. Although, the slowdown is not trivial, it does drastically improve the readability of the graph.

1.2 Splines

In a nutshell, a spline is a piecewise-defined function that is smooth and continuous at each piece's endpoint, such that the entire curve looks like a single function. The pieces of a spline are usually quadratic or cubic functions whose 1st and 2nd derivatives match at the endpoints. In most cases, cubic splines are used because they look smoother and more natural than quadratic splines.

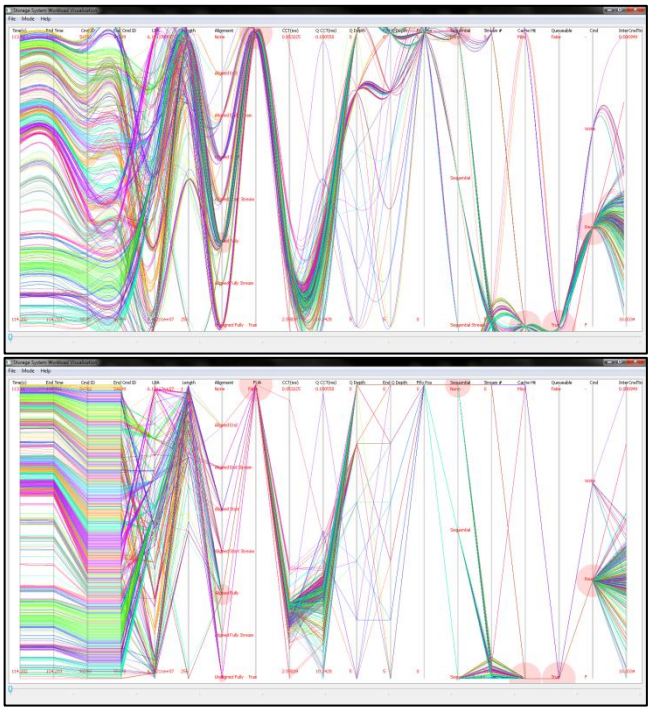

Fig. 1. The output of the Storage System Data Visualizer with splines (top) versus without splines (bottom)

Splines are often used for function interpolation: where we know several data points, but we want to be able to extrapolate as to where the function value might be if we sampled the function between data points. This is exactly what humans desire to do naturally when presented with a list of points on a graph - to draw some correlation between the set of points. In our case, with a possibility of 50,000 sets of data on the graph, it becomes necessary to help the mind out by plotting lines between the statistics to give some semblance of readability. Splines just let us do this task more effectively.

1.3 Geometry Shaders

A geometry shader is a relatively new type of shader that can transform sets of points into different shapes, like triangles, or in this case, more points. From the initial set of 19 points that are used in the data visualizer, the geometry

shader will output 19n points, where n is the granularity of the lines.

1.4 Who's Done This Before

As to be expected, there are many articles online which detail using splines to better visualize data sets. However, there are limited resources geared towards optimizing data visualization using CUDA.

2. How We Did It

Two different implementations were compared to determine which implementation would be faster, using CUDA or using a Geometry Shader to implement splines. Both implementations will output the same *granularity* of lines, in other words, if the granularity was 5, there will be five sub-lines plotted between each set of data points.

2.1 Computing a Spline

In order to turn a list of points

$$P = \{(d_0, y_0), (d_1, y_1), \dots, (d_n, y_n)\}$$

into a set of continuous piecewise functions, we need to have a certain set of conditions. First, we have a list of piecewise functions defined as follows:

$$f(x) = \begin{cases} f_1(x), & d_0 \le x \le d_1 \\ f_2(x), & d_1 \le x \le d_2 \\ \dots & \\ f_n(x), & d_{n-1} \le x \le d_n \end{cases}$$

Here, $f(x)$ is the entire spline. The set of functions f_k will be called F. Each piecewise function will be defined as:

$$f_k(x) = a_k x^3 + b_k x^2 + c_k x + e_k$$

We need to solve for $a, b, c,$ and d for each function. For n piecewise functions, this gives us $4n$ unknown variables. Notice how we only define n functions for $n + 1$ points; this is because we are defining each function as being *between* two points.

First, each of our piecewise functions will have to match up at each of the (d_k, y_k) predefined points. So for each piecewise functions, we have two equations:

$$a_k x_k^3 + b_k x_k^2 + c_k x_k + e_k = y_k$$
$$a_k x_{k+1}^3 + b_k x_{k+1}^2 + c_k x_{k+1} + e_k = y_{k+1}$$

Next, we must set several of the derivatives of each function equal to each other to give the appearance that the graph is smooth. For a cubic spline, the first and second derivatives must be equal. So for every adjacent set of function pieces, there are two more equations:

$$a_k x_k^2 + b_k x_k + c_k = a_{k+1} x_k^2 + b_{k+1} x_k + c_{k+1}$$

$$a_k x_k + b_k = a_{k+1} x_k + b_{k+1}$$

So, we have $(n-1)(n-1)(n)(n)$ equations and $4n$ unknowns, leaving us needing two more equations to be able to construct the spline. This is where the "natural" part of a natural cubic spline comes in. As there aren't two adjacent splines at the endpoints, we just set the 2nd derivatives at the endpoints to zero to get the last two equations:

$$a_0 x_0 + b_0 = 0$$
$$a_n x_n + b_n = 0$$

To make this system of equations easier to solve, the equations can be turned into a set of matrices involving z_i, where $i = 1..n$. The set of z values represents the list of 2nd derivative values at the predefined points P. Together with the list of points P, we can define concretely our set of functions F. With the z values, we can define each function in F now by this formula:

$$f_k(x) = \frac{z_{k+1}}{6h_k}(x - x_k)^3 + \frac{z_k}{6h_k}(x_{k+1} - x)^3 + \left(\frac{y_{k+1}}{h_k} - \frac{h_k}{6} z_{k+1}\right)(x - x_k) + \left(\frac{y_k}{h_k} - \frac{h_k}{6} z_k\right)(x_{k+1} - x)$$

Here, $h_k = x_{k+1} - x_k$. Then we have the next equation, which lets us solve for the z values:

$$h_{k-1} z_{k-1} + 2(h_{k-1} - h_k) z_k + h_k z_{k+1} = 6\left(\frac{y_{k+1} - y_k}{h_k} - \frac{y_k - y_{k-1}}{h_{k-1}}\right)$$

Finally, we can set up the linear system of equations. Here is an example of the matrices for a list of 5 points P, generating a list of 4 functions F:

$$\begin{bmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} y_2 - 2y_1 + y_0 \\ y_3 - 2y_2 + y_1 \\ y_4 - 2y_3 + y_2 \end{bmatrix}$$

Now we can solve this system of equations. To do so quickly on CUDA, the Jacobi method was used. This is an iterative method which can't be parallelized using CUDA, but each spline computed can be solved in parallel. After the z values are computed, they can be plugged into the previous formula to finally get the piecewise function representing the spline.

The Jacobi method, on which we won't go into much detail here, is an iterative method for solving linear systems of equations that have certain special properties. Luckily, the system of equations that is used will always fulfill these properties, so we know this method will always work for this application.

Essentially, the Jacobi method involves:

1. Guessing at an initial solution. A set of 0 solutions works here.
2. Computing the right hand side using the guess, which just involves a simple matrix multiply.
3. Using the right hand side to find the next guess.

After these three steps occur, the algorithm checks for convergence by comparing to the previous guess. If the change isn't small enough, another iteration is performed.

Upon receiving the piecewise functions to compute the spline, in the CUDA implementation we simply pass these to another function which generates a specified number of solution points along each spline and renders them.

2.2 Implementing Geometry Shaders

In terms of implementation, the geometry shader and CUDA implementations are essentially the same. The set of points will be transformed in the shader with the same linear system of equations, which is solved using the Jacobi method.

However, there are many more implementation limitations with the geometry shaders than with the CUDA implementation. Geometry shaders have a hard limit on the number of points that they can transform, so with the set of 19 data points, we can have a maximum granularity of 7.

Additionally, like the CUDA implementation, the Jacobi method was set to a fixed number of iterations to increase parallelization. Checking whether the Jacobi method converged for every iteration is tedious and unnecessary for the task at hand, so the number of iterations to do for each spline curve was simply set to 10.

3. Results

Our CUDA implementation has roughly the same performance as our geometry shader implementation. For the CUDA implementation, we have parallelized the computation of the spline curves. Without this parallelization, rendering spline curves would be too slow to be worthwhile.

Both implementations are significantly slower than the original, and for good reason. Generating spline curves requires significant effort, even on the GPU, to compute. In the amount of time the original was able to render about fifty thousand splines, we are only able to render five thousand splines. Our test GPU was not very powerful and substituting a faster GPU may significantly lower the time needed to render a larger dataset. On the same GPU, the geometry shader implementation can render all fifty thousand splines in almost the same amount of time as the original can render the same data.

Although the CUDA implementation is pretty fast right now, there are still many optimizations to be done. One idea that may be feasible is to keep the data points on the graphics card before we draw them, and display them using OpenGL. Right now a significant portion of the runtime of the CUDA system is focused on simply copying data back and forth from the graphics card, the slowest operation.

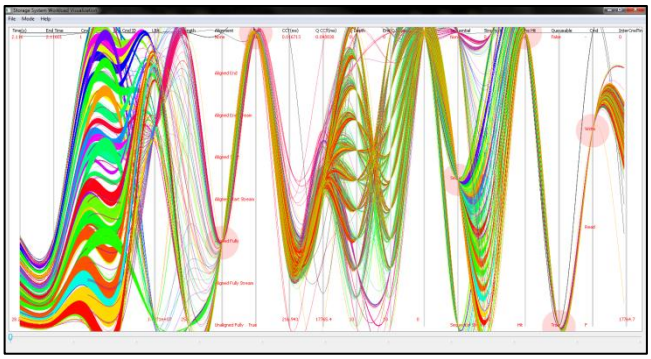

Fig. 2. Our Storage System Data Visualizer output with 5000 splines

4. Conclusion

Using spline curves to display the data has made the graph significantly easier to read. With all of the curves taking slightly different routes between the points, the graph looks much less cluttered. All previous graph options, such as color, repositioning, and zooming, have been maintained and no features have been lost. The increases in readability more than make up for the speed lost in the spline computation.

If we had to do this project over again, the main thing we would change would be our development environment. First, we would set up a Linux based CUDA machine. Our primary CUDA development machines were both running Windows. This became a difficult because Visual Studio turned out to be the only way to compile CUDA, and the CUDA objects had to be referenced from Qt Creator. Second, we would consider porting the application to a more CUDA friendly IDE. Qt Creator has little to no documentation for setting up CUDA. Simply getting the Qt Creator to run CUDA took a significant amount of setup, and we were unable to get Qt Creator to actually compile any CUDA code. Setting up CUDA on Qt Creator on Linux seemed to have more online resources, so if we were unable to port the application, then we would switch to a Linux environment.

Although we were able to make significant improvements to the look of the graphs, there is still more that can be done. There is room for significant speedup in both implementations.

5. Acknowledgements

We would like to thank Stephen Pungdumri, Chris Lupo, John Oliver, and Zoë Wood for their work on the original system.

6. References

[1] Landweber, Gregory D. "How to Compute Natural Cubic Splines." *Bard College Mathematics*. Bard College, 11 Feb. 2004. Web. 29 May 2012. <http://math.bard.edu/greg/math301/Splines.pdf>.

[2] Larson, Ron. "Iterative Methods for Solving Linear Systems." *Elementary Linear Algebra 5e*. CEngage. Web. 29 May 2012. <http://college.cengage.com/mathematics/larson/elementary_linear/5e/students/ch08-10/chap_10_2.pdf>.

[3] Pungdumri, Steven C., Chris Lupo, Zoe Wood, and John Oliver. "An Interactive Visualization Model for Analyzing Data Storage System Workloads." Thesis. Cal Poly San Luis Obispo, 2011. Print.

[4] Alfonse. "Geometry Shader." *OpenGL*. OpenGL, 31 May 2010. Web. 08 June 2012. <http://www.opengl.org/wiki/Geometry_Shader>.