Superconductor: GPU-accelerated Big Data Visualization for the Browser

Matthew E. Torok (mtorok@eecs.berkeley.edu)
Undergraduate, University of California, Berkeley. Advisor: Ras Bodík
565 Soda Hall, Berkeley, CA 94720-1776. ACM Member Number: 6534400

Abstract

There is need for data visualization tools which are both scalable and productive We show how previous work on parallel schedule synthesis for attribute grammars can be extended to this domain. Our results retain the flexibility of common tools in this area, while handling up to two magnitudes more data.

1. Motivation

Data is growing at an increasing pace. In 2003, it took 10 years to sequence a full human genome; today, the same can be done in a week, generating 700 MB of compressed data. Interactive data visualization is one technique for understanding and working with data.

Bostock and Heer [2] recognize the need for visualization tools which combine accessibility, expressiveness, and efficiency. Large datasets put an especially large emphasis on efficiency, yet traditionally tools have traded performance for accessibility and expressiveness

Browser-based interactive visualization tools have become popular in part because because of the accessibility and expressiveness they provide. Web browsers provide a high-level, widely known development environment, simple methods for user interaction, and a common platform for distribution. However, these tools often lack the performance required for big data.

In our tests, D3.js, a popular browser-based visualization tool, was able to animate just 1,000 points of data in a sunburst visualization while maintaining a speed of 30 frames-per-second. When the size of the data was increased to 10,000 points, animation speed was just 10 frames-per-second.

Our previous work has shown how to synthesize a schedule of parallel tree traversals from an attribute grammar [4]. As part of our previous position paper on this subject, we outlined how we might apply our work on parallel schedule synthesis to data visualization [3]. Here, we show how our visualization tool, Superconductor, implements those ideas.

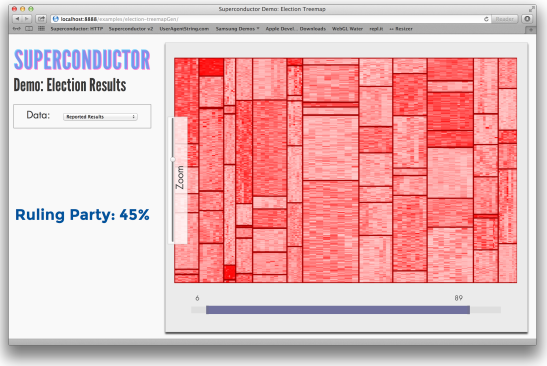

Figure 1: An interactive treemap visualization using Superconductor, displaying 94,000 points of data collected from a recent election.

2. Contributions

I contributed three major points to this project:

- To use the FTL synthesizer to find a parallel tree traversal schedule, I converted a collection of common visualizations to attribute grammar form.
- To enable scalable visualizations in the web browser, I wrote a code generator which takes a parallel traversal schedule, and generates GPU-based WebCL code targeting the browser.
- To enable efficient execution and an accessible interface, I created a runtime library for running these visualization engines within the browser. This includes support for data parsing, GPU device management, dynamic GPU memory allocation, and rendering support.

2.1 Synthesis

The FTL synthesizer uses attribute grammars as a declarative specification of layouts. From this, it is able to synthesize a schedule of parallel tree traversals which can be used to implement the program. These specifications only declare the behavior of the program, not the implementation. In this way, the programmer can focus on the behavior of the visualization, while the synthesizer can discover parallelization in the visualization.

For the Superconductor project, I worked on identifying a collection of common visualizations and creating attribute grammar specification for them. Figure 2 shows part of the attribute grammar used to specify the treemap visualization in figure 1. Superconductor is able to animate this visualization, containing 94,000 data points, at approximately 30 frames-per-second. I also created

```
1  class Container {
2    children c : [ Box ]                  //layout tree schema
3    input width = 300;                    //default overridable by selectors
4    var height = width;
5    width[i] = width * c[i].magnitude / totalMagnitude
6    var totalMagnitude = totalMagnitude + c[i].magnitude
7    ...
8  }
9  class Box {
10   input magnitude = 0
11   var x, y, width, height;
12   var render = drawSquare(x,y,width,height)
13   ...
14 }
```

Figure 2: Excerpt of attribute grammar representation of a treemap used to generate the visualization in figure 1.

a sunburst visualization with Superconductor, which was able to animate 1,000,000 data points at 30 frames-per-second, in comparison with D3.js animating 1,000 nodes at the same speed.

In accordance with the capabilities of the FTL synthesizer, we believe any tree-structured visualization, with computations linear in the number of children of a node, can be expressed in this way.

2.2 Code Generation

From a schedule of parallel tree traversals, we are able to directly code generate an implementation of the specified visualization. Superconductor targets the browser as its visualization platform, and makes use the GPU for parallelization. WebCL [1], an emerging web standard, allows our browser-based code access to GPU.

For Superconductor, I developed a GPU-based code generator. This work involved first creating a collection of parallel tree traversal patterns for the GPU. Efficient GPU data structures, which structure split tree data into flat arrays in a method similar to that in Blelloch and Greiner [1], were also created. The code generator uses these patterns and data structures to implement a parallel visualization engine. The schedule defines which pattern should be run, and in what order, as well as what computation should occur at each step.

The product of this code generation is a layout engine module implementing a particular visualization. A designer is able to choose how he or she wants to visualize their data, and load the layout engine implementing that visualization at runtime.

2.3 Runtime

The Superconductor visualization framework is used by designers via the `superconductor.js` runtime JavaScript this library. This library handles GPU device management, data parsing and transfer, and rendering.

One novel aspect I contributed to this library was our method of rendering. WebCL allows for directly sharing memory with WebGL, avoiding potentially slow transfers to-and-from the GPU. Our rendering API is implemented as a collection of functions which write WebGL-compatible vertex data into this shared memory, and WebGL is invoked to perform the actual rendering.

The layout specification is structured in such a way that visualization attributes are competed prior to calling rendering function. As a consequence, the number of vertices being rendered is unknown until after the layout engine has been executed on the input and these functions have been called. To avoid overflowing the allocated rendering buffer, we introduce an attribute to our attribute grammar which computes the size of the data to be rendered. The runtime library then ensures that the vertex buffer is sufficiently large to hold this data and, if not, reallocates the buffer with sufficient size.

[1] http://www.khronos.org/webcl/

3. Conclusion

I have shown my contributions in extending the FTL synthesizer for accessible, productive and efficient data visualization in order to support large data sets. My work has included expressing common visualizations as attribute grammars, creating a WebCL code generator for the creation of browser-based GPU-accelerated visualizations engines, and the creation of an efficient runtime library for the execution of these engines.

References

[1] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of nesl. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 213–225, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7. doi: 10.1145/232627.232650. URL `http://doi.acm.org/10.1145/232627.232650`.

[2] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2009. URL `http://vis.stanford.edu/papers/protovis`.

[3] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Superconductor: A language for big data visualization. LASH-C, 2013, 2013.

[4] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Parallel schedule synthesis for attribute grammars. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 187–196, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442535. URL `http://doi.acm.org/10.1145/2442516.2442535`.