# Superconductor: A Language for Big Data Visualization

Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, Rastislav Bodík
University of California, Berkeley [*]
{lmeyerov,mtorok,ericatkinson,bodik}@eecs.berkeley.edu

## 1. Introduction

Increases in data availability is the force behind many recent innovations in society. However, visualization technology for exploring data is not keeping up. Designers must choose between scale and interactivity. They want big displays because of the ability to show an entire data set. However, viewers get lost in a sea of noise, so they also want interactivity. Performance constraints limit interactions to operating on a small data slice.

We present SUPERCONDUCTOR: a high-level visualization language for interacting with large data sets. It has three design goals that we distilled from popular visualization languages:

- **Scale.** Visualizations should support thousands or even millions of data points. For example, Matlab and Circos[1] are used for static visualizations of large data sets.
- **Interactivity.** Interactions should be within 100ms and animation should achieve 30fps. For example, JavaScript and its libraries such as D3 [2] are used for animating data and orchestrating interactions with users.
- **Productivity.** Programming should be at least as high-level as JavaScript. Designers invoke and customize visualizations in the above systems, and even create new ones from scratch. They less frequently do so with C++ and OpenGL.

Visualization languages support one or two of the above goals, but only SUPERCONDUCTOR seeks to address all three (Figure 1).

SUPERCONDUCTOR is split into three domain specific languages (DSLs). It refines our early design of a parallel browser [3] with lessons on how to combine the components [6, 7]. Each DSL is declarative and parallel:

| Name | Role | Tree traversal patterns |
|---|---|---|
| layout | define how to lay out a widget | preorder, postorder |
| rendering | define how to paint a widget | prefix sum, for-all |
| selectors | map data to stylized widgets | for-all |

By automatically optimizing these DSLs, SUPERCONDUCTOR supports the design of big, interactive visualizations.

[*] Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

[1] `http://www.circos.ca/`

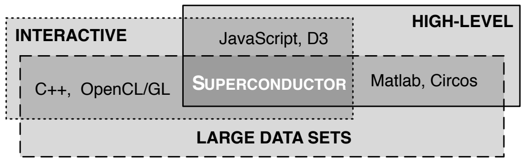

**Figure 1:** Design axes for visualization languages.

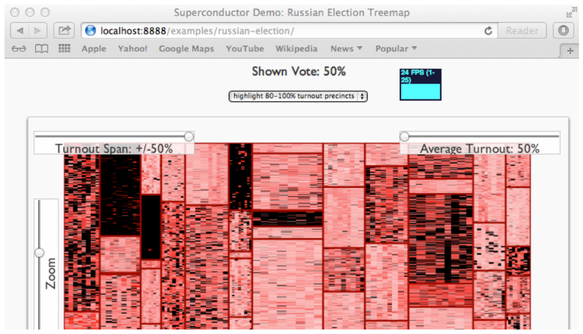

**Figure 2:** Interactive treemap of 94,000 voter polling stations.

## 2. Architecture & Example

We explain our DSLs through a treemap visualization of recent election results (Figure 2). The area of a small rectangle depicts voter count at one polling station, and the coloring depicts the average party voted for (red, white, or a hue in-between). Borders show how the 94,000 stations are grouped by districts. Black coloring highlights stations with a 80-100% voter turnout. Sliders filter stations by turnout and a dropdown menu toggles the highlighting.

The DSLs form a three-stage pipeline at runtime (Figure 3). First, CSS selectors [1] map voting data into a treemap widget and manipulate colors and filters. We implement selectors with a parallel for-loop [6]. Second, layout solves the size and position constraints of each treemap element. A sequence of parallel preorder and postorder tree traversals computes them. Offline, a designer declaratively specifies the treemap in our extension to attribute grammars [4, 7] and our synthesizer statically schedules as tree traversals. Finally, the runtime layout solver issues rendering commands. Our rendering API optimizes the use of GPU memory.

## 3. Selector DSL for parallel styling

CSS selectors [1] map the document tree into a tree of layout widgets and add style constraints. For example, the selectors in Figure 4 map voting data into three types of treemap nodes and customize the border style. Every rule left-hand side matches a set of nodes. Reading the expression "`.Nation *`" right-to-left, it

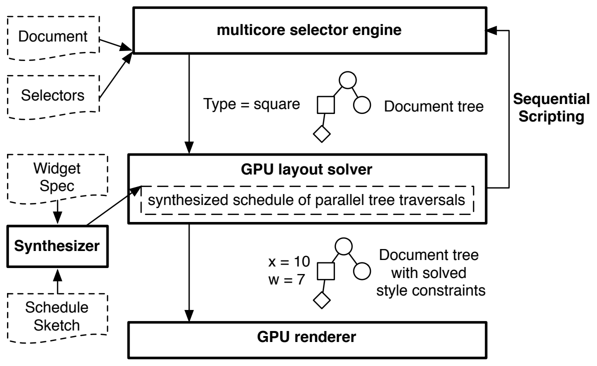

**Figure 3:** Parallel system architecture of SUPERCONDUCTOR.

```
1 .Nation { display: Root; width: 400px }
2 .Nation * { display: Intermediate; border: 2px red }
3 .Precinct { display: Leaf; min: 0.2; max: 0.4 }
```

**Figure 4:** Selectors for mapping XML into a stylized treemap

```
1  class Root
2    children c : [ Node ]                    //layout tree schema
3    input width = 300;                       //default overridable by selectors
4    var height = width;
5    var render = paintSquare(0,0,votes[last],votes[last])
6    var votes[-1] = 0
7    votes[i] = votes[i - 1] + c[i].votes
8    ...
9  class Leaf : Node
10   input count = 0, min = 0.0, max = 1.0
11   var votes = turnout ∈ (min,max) ? count : 0
12   ...
13 schedSketch: parPreorder _ ; parPostorder _ ; _ ; _ ; _
```

**Figure 5:** Declarative specification of the election treemap. Line 13 is the parallel schedule sketch and the rest is functional behavior.

matches any node ("`*`") with an ancestor whose attribute "`class`" is "`Nation`". Right-hand side properties, such as "`border: 2px red`", are attached to matched nodes.

The DSL simplifies parallelization: it guarantees that rules can be matched independently. Previously [6], we built a native multicore engine that optimizes matching as a parallel for-all loop over nodes. We are now experimenting with two variants built on web standards: a multicore one via zero-copy message passing (*workers* with *transferable objects*), and a GPU variant via WebCL.

Selectors enable making sweeping changes to a visualization. For example, the `min`/`max` fields on line 3 control which polling stations to show based on voter turnout. A script binds UI sliders to the fields. JavaScript is fast enough for running the script because it modifies selector rules rather than nodes. The computations needing acceleration are downstream: selectors, layout , and rendering.

## 4. GPU layout through schedule synthesis

After selectors associate style constraints with nodes, layout solves all of the constraints. Widgets are *attribute grammars* [4, 7]. Every attribute of a widget either appears on the left-hand side of one widget constraint, as in variable *height* in line 4 of Figure 5, or is provided by matched selectors, such as input *width* being provided by line 1 of Figure 4. The right-hand side of a constraint may reference attributes of adjacent nodes, such as the tallying of child `vote` fields in lines 6-7.

Layout executes as a sequence of parallel tree traversals. The tree traversal schedule must observe data dependencies, such as the `paintSquare` call on line 5 occurring *after* vote tallying on lines 6-7. Offline, our synthesizer statically analyzes the grammar to compute the schedule. It returns a pattern for each pass (e.g., parallel preorder) and what computations to run in a node visit (e.g., the vote tally in the first pass and rendering calls in the last).

Our schedule synthesis algorithm [7] desugars a widget specification into an attribute grammar (AG) [4]. AGs simplify scheduling because a node's constraints are local to its neighborhood. Once scheduled, our compiler performs optimizations for trees such as tiling. This enabled multifactor speedups in our multicore backend, and we are now exploring optimizations for our GPU backend [5].

Our DSL's schedule *sketching* [7] construct enables lightweight partial specification of the automatic parallelization. For example, line 13 of Figure 5 specifies 5 traversals. It declares the patterns of the first two, and none of the attributes to compute within them. Our synthesizer finds schedules satisfying the sketch and grammar.

Sketching also improves optimization time. Our schedule synthesizer uses them to prunes its search space. Autotuning for the fastest schedule is also faster because fewer schedules are profiled.

## 5. Automatic memory management for rendering

Passing layout information to a GPU renderer is a bottleneck. SUPERCONDUCTOR uses three key techniques. First, as both layout and rendering occur on the GPU, the rendering API invoked by layout does not move data off of the GPU. Second, rendering API calls directly generate vertices that OpenGL renderers use.

Our third technique hides memory allocation. The layout solver automatically preallocates layout memory as soon as the tree size is known, but rendering memory must be dynamically allocated. For example, a large circle needs more vertices than a small circle, and the circle size might be computed as part of layout solving. The first rendering API pass precomputes the buffer space and position needed to render each node. It is a prefix sum over the tree [5] that, in essence, partially evaluates rendering calls for a given input. The full rendering buffer is then allocated, and a second rendering pass fills it in. Our synthesizer fuses the passes into the layout traversals.

## 6. Conclusion

We have shown how high-level programming abstractions support automatic parallelization. We examined three cases: selectors, layout, and rendering. In the case of layout, declarative constructs can further guide parallelization. Together, these ideas enabled our goal of high-level programming of big, interactive visualizations.

## References

[1] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2 CSS2 specification, 1998.

[2] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 2011.

[3] C. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodík. Parallelizing the web browser. HotPar'09, 2009.

[4] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 1980.

[5] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *PPOPP '12*, pages 117–128, 2012.

[6] L. A. Meyerovich and R. Bodík. Fast and parallel webpage layout. In *WWW'10*, pages 711–720, 2010.

[7] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodík. Synthesizing parallel schedules for attribute grammars. In *PPOPP '13*, 2013.