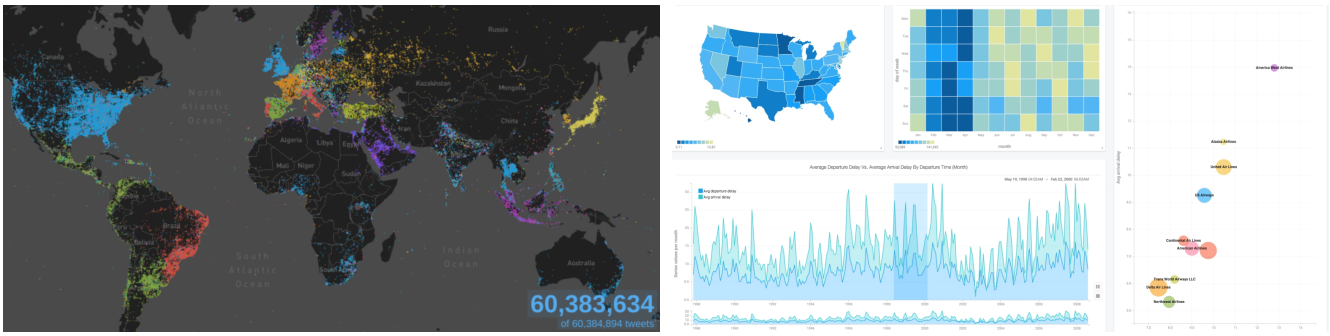
MapD: A GPU-powered Big Data Analytics and Visualization Platform

Christopher Root*
Senior Graphics Engineer

Todd Mostak†
CEO



(a) *A geospatial visualization of tweets colored by language. Basemap courtesy of © Mapbox, © OpenStreetMap.*

(b) *A dashboard of charts filtered by time span to explore tweet data.*

Figure 1: *Visualizations of a tweet dataset with over 60 million rows.*

Abstract

MapD, or "Massively Parallel Database", is a big data analytics platform that can query and visualize big data up to 100x faster than other systems. It leverages the massive parallelism of commodity GPUs to execute SQL queries over multi-billion row datasets with millisecond response times, and optionally render the results using the GPU's native graphics pipeline. Depending on the use case, MapD can be used as a standalone SQL database or as a data visualization suite by using its own visualization frontend (see Fig. 1) or by integrating it with other third-party toolkits.

Keywords: GPU, database, analytics, data visualization, CUDA, OpenGL

Concepts: •**Information systems** → **Data management systems;** •**Human-centered computing** → **Visualization systems and tools;** •**Computing methodologies** → **Graphics processors;**

1 Speed

MapD is an in-memory database that leverages both GPUs and CPUs to enable lag-free data exploration. While MapD can run on CPU-only machines and still achieve significant speedups over other in-memory systems, it is targeted at machines with GPUs.

Using a sample SQL query with a WHERE and GROUP clause over 3 billion rows, a leading in-memory database scanned 150 million rows per second (2 x 8 cores Xeon). By comparison, MapD, running CPU-only, scanned 6 billion rows per second, and running on GPU + CPU scanned 260 billion rows per second (8 x Nvidia K80s, 2 x 8 cores Xeon). For more performance benchmarks, please visit www.mapd.com/product.

MapD achieves its speed using a variety of novel techniques such as:

1. Compiles incoming queries to both CPU and GPU machine code using LLVM and executes queries simultaneously on both CPU and GPU.
2. Visualizes data and performs complicated analytics *in situ* on the GPU. Since relevant data is already cached on the GPUs, MapD does not need to copy the query result set before rendering it.
3. Vectorizes execution of queries when possible. Vectorized code allows the compute resources of a processor to process multiple data items simultaneously: a must for good performance on GPUs, and increasingly, CPUs as well.
4. Employs highly-optimized GPU routines for common database operations.
5. When compiled in a hybrid GPU/CPU system, data parsing and ingestion is handled by CPU, without slowing down GPU read access.
6. Deploys in a single node configuration avoiding the network overhead inherent in distributed systems.

2 Database Size

MapDs largest server instance is for 8 Nvidia K80 GPUs, or 192GB of GPU RAM. In practice, datasets 1.5 - 3TB in raw size can typically be handled in GPU RAM, or 10 - 15TB in CPU RAM.

3 Rendering

MapD utilizes the CUDA/OpenGL Interoperability API to map query results into OpenGL vertex buffers for an optimal backend

*e-mail:chris@mapd.com
†e-mail:todd@mapd.com

SIGGRAPH '16, July 24-28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4282-7/16/07
DOI: http://dx.doi.org/10.1145/2897839.2927468

rendering pipeline. Results can be drawn as a variety of different primitives (points, lines, polygons, sprites), the properties of which can be linked to specific columns in the result set and mapped to a preferred output via a library of scaling functions performed in the shader. This mapping is configured in the frontend using a JSON format modeled after the visualization grammar format *Vega*. Fig. 2 illustrates one such configuration to render a large political donations dataset geospatially as points.

For interactive feedback of the data in the frontend, an auxiliary buffer storing the database row ids is appended to the render and cached for fast per-pixel lookup.

The final render is PNG-encoded and sent to the frontend for display. In the case of geospatial renders on a map, the png is layered atop a vector-tile map in the frontend using a third party map-rendering engine. We intend to explore other delivery methods such as hardware-accelerated video encoding in the future to improve this pipeline.

3.1 Multi-GPU

In the case where the database requires multiple GPUs for storage, a subset of the query results are generated per-GPU and rendered separately. The final image is generated by copying the resulting color, depth, and any auxiliary buffers to a primary GPU for compositing. The copy method used is determined by the windowing library configuration. For example, in an EGL-configured system, we take advantage of the *KHR_image_base* extension to create a composite framebuffer on the primary GPU and use the *OES_EGL_image* extension to create siblings on the others to handle the copy automatically. In a GLX-configured system, we employ the *NV_copy_image extension* to copy the buffers across contexts.

4 Demo

Please visit www.mapd.com/demos to experiment with several live demonstrations, including the 60 million+ tweet dataset seen in Figure 1a and the 25 year political contributions dataset seen in Figure 2b.

Acknowledgements

We'd like to acknowledge the rest of the MapD engineering team for their excellent work - backend engineers: Alex Suhan, Andrew Seidl, Michael Thomson, & Minggang Yu, frontend engineers: Marc Balaban, Danny Delott, Du Hoang, Jonathan Huang, & Mike Luby, & product management: Ed O'Donnell.

References

BOLZ, J., JONES, J., AND OTHERS, 2012. Nv-copy-image extension. https://www.opengl.org/registry/specs/NV/copy_image.txt.

HEER, J., AND OTHERS. Vega: A visualization grammar. http://vega.github.io/vega/.

JULIANO, J., KING, G., AND OTHERS, 2014. Khr-image-base egl extension. https://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image_base.txt.

KING, G., LEECH, J., AND POOLEY, A., 2015. Oes-egl-image gl extension. https://www.khronos.org/registry/gles/extensions/OES/OES_EGL_image.txt.

Figure 2: *Render configuration example for a political donation dataset.*

(a) *A json config describing scales for point size and fill color.*

```
{
  "data":[
    {
      "name": "table",
      "sql":  <SQL>
    }
  ],
  "scales":[
    ...,
    {  // a linear scale
      "name": "size",
      "type": "linear",
      "domain": [   0, 5000 ],
      "range":  [ 2.0, 12.0 ],
    },
    {  // an ordinal scale
      "name":"color",
      "type":"ordinal",
      "domain":["Rep", "Dem", "Ind"],
      "range": ["red", "blue", "green"],
    }
  ],
  "marks":[
    {  // points colored by party
       // and sized by donation amount
      "type": "points",
      "from": { "data": "table" },
      "properties": {
        ...,
        "fillColor": {
          "scale": "color",
          "field": "party"
        },
        "size": {
          "scale": "size",
          "field": "amt"
        }
      }
    }
  ]
}
```

(b) *The resulting render using donation amt for size, and party for color. Basemap courtesy of © Mapbox, © OpenStreetMap.*