This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

P4: Portable Parallel Processing Pipelines for Interactive Information Visualization

Jianping Kelvin Li and Kwan-Liu Ma, *Fellow, IEEE*

Abstract—We present P4, an information visualization toolkit that combines declarative design specification and GPU computing for building high-performance interactive systems. Most of the existing information visualization toolkits do not harness the power of parallel processors in today's mainstream computers. P4 leverages GPU computing to accelerate both data processing and visualization rendering for interactive visualization applications. P4's programming interface offers a declarative visualization grammar for rapid specifications of data transformations, visual encodings, and interactions. By simplifying the development of GPU-accelerated visualization systems while supporting a high degree of flexibility and customization for design specification, P4 narrows the gap between expressiveness and scalability in information visualization toolkits. Through a range of examples and benchmark tests, we demonstrate that P4 provides high efficiency for creating interactive visualizations and offers drastic performance improvement over current state-of-the-art toolkits.

Index Terms—information visualization, interactive visualization, parallel data processing, GPU computing, data exploration

1 Introduction

Visualization proves to be effective for reasoning and exploring large and complex data, as it can effectively augment human cognition to gain insight from massive information [1]. Visualization systems must provide interactive performance for effective data exploration and analysis [2], [3]. As the trends of mainstream computer hardware are dominated by parallel and manycore processors, visualization software should effectively leverage parallel hardware to improve performance. In particular, Graphic Processor Units (GPUs) in mainstream computers provide massive computing power to boost the performance of visualization systems. However, exploiting the potential of GPU computing for interactive visualization is not a trivial task for visualization designers and programmers. Furthermore, interactive visualization requires both data processing and visualization rendering to be efficient, which needs not only parallel rendering but also general-purpose processing on the GPU (GPGPU).

Many visualization toolkits [4], [5], [6], [7] provide a high degree of expressiveness with declarative grammars for rapid specification of visualization design, which permits high productivity in developing visualization systems. However, an equally important quality of a visualization toolkit is its support for performance. Most existing information visualization toolkits lack parallel processing support to build high-performance interactive systems. Our work attempts to fill the gap between the expressiveness and high performance of visualization toolkits.

In this paper, we present P4, a visualization toolkit for building GPU-accelerated information visualization systems. Our design addresses four primary goals.

Performance. Nowadays, datasets with several million records are common. Modern GPUs typically available in commodity personal computers possess enough memory capacity and computing power for storing and processing such large data. P4 exploits GPU computing to provide interactive performance for exploratory visual analysis, accelerating both data processing and visualization rendering. P4 also efficiently allocates GPU resources to be shared between data processing and rendering, providing seamless interoperability to ensure high performance.

Productivity. An easy-to-use programming abstraction is necessary for productive development of high-performance visualization systems. P4's application programming interface (API) is based on the semantics of information visualization, allowing declarative specifications of common data transformations, visual encodings, and interactions. P4 makes it possible for visualization designers with little or no knowledge in GPU programming to build high-performance interactive systems.

Programmability. Providing a high-level programming abstraction for parallel processing and visualization design helps productivity, but it may restrict programmability as users are limited to the provided functionalities. For added programmability, P4 allows user-defined logics for customizing data transformation and visual encoding. At runtime, P4 synthesizes user-defined logics and design specifications to create effective GPU programs for data processing and rendering.

Portability. Web-based visualization systems are in wide use. To support the development of web-based visualization applications, P4 leverages a standardized graphics API (WebGL) to parallelize both data transformation and visualization, providing portable performance for different types of GPUs with widely varying numbers of processor cores. Applications made by P4 can be run on modern browsers without any plugin, allowing access across computing devices. P4 can be used seamlessly with different web technologies and other visualization libraries to build

• *J. K. Li and K-L. Ma are with the Department of Computer Science, University of California, Davis, CA, 95616. E-mail: {kelli,klma}@ucdavis.edu*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

interactive systems.

To achieve these four design goals, we combine GPU computing and declarative specification for interactive data visualization. P4 offers a programming interface with declarative grammar to leverage GPU computing for accelerating both data transformations and visualziations. We show how P4's system architecture and parallel processing framework allow declarative visualization grammar to be used with GPU computing. By combining runtime code generation and GPGPU techniques, we have made P4 a flexible, high-performance, and extensible visualization toolkit. In a range of examples, we demonstrate the usefulness and efficacy of P4 for creating interactive visualization and supporting exploratory visual analysis. We also describe three applications built with P4 to show the effectiveness of P4 for developing high performance visualization systems. Our benchmark testing shows that P4 provides more than an order of magnitude improvement in interactive performance comparing to the state-of-the-art systems.

2 Related Work

Our work aims to narrow the gap between expressiveness and performance in visualization toolkits by incorporating declarative specification and high-performance visualization techniques in P4.

2.1 Visualization and Graphics Toolkits

Many visualization libraries provide declarative grammars for designing information visualization applications. Declarative grammars can decouple specifications of visualization and interaction from execution details and defer control flow concerns to the runtime, which allow programmers to focus on application-specific design decisions [8]. The Grammar of Graphics by Wikinson [9] is applied as the foundation of declarative grammar for specifying visual encodings in many popular visualization toolkits, such as ggplot2 [5], Protovis [10], and D3 [4]. In addition to visual encoding, Vega [6] and Vega-Lite [7] provide declarative grammars for specifying interactions. However, these toolkits only use serial computation on the CPU, and thus cannot provide interactive performance for large data without applying additional high-performance visualization techniques.

As GPUs are prevalent in commodity computers, libraries and tools for GPU programming become useful for developing high-performance visualization systems. For scientific visualization, GPU computing has been broadly adopted to boost system performance. However, the adoption rate of GPU computing is much slower in the field of information visualization. This is because parallelizing the entire information visualization pipeline is challenging. In particular, GPGPU techniques [11], [12] are needed for data transformation in addition to parallel rendering for visualization. A common approach is to use a graphics API (e.g, OpenGL [13] and WebGL [14]), where the algorithms for both data transformation and visualization are embedded in shader programs. Using a graphics API for common data transformation requires expressing data in vertex and pixel space, which makes GPU programming more difficult. Newer graphics APIs may provide some support for embedding general purpose computations along with rendering (e.g., Compute Shader [15] in OpenGL 4.3+), but programmers are still required to be familiar with OpenGL's graphics pipeline. Another approach is to use a GPGPU API (e.g., OpenCL [16] or CUDA [17]) for data transformation, and then use a graphics API for visualization. In this case, application developers need to be familiar with not only two different APIs but also the inter-operation between them. Even with libraries that are designed for making GPGPU programming easier, such as Thrust [18] and Boost.Compute [19], a significant amount of software engineering effort is still required.

Researchers in the information visualization community have also investigated alternative approaches. McDonnel et al. [20] presented a visual programming environment with graphical user interfaces for composing GPU shaders to create data visualizations. However, graphical user interfaces can rarely provide the flexibility needed to create customized visualizations. Ren et al. [21] developed Stardust to provide a programming interface similar to D3 to render large data with WebGL. However, Stardust only focuses on rendering and provides no support for GPU-based data transformations.

While high-level visualization libraries (e.g., D3 and Vega) with declarative grammar cannot provide high performance for big data applications, low-level graphics libraries (e.g., WebGL) lack efficiency for designing interactive visualizations. We combine declarative visualization grammar and GPU computing in P4 to provide a high-performance toolkit for building information visualization systems.

2.2 High-Performance Visualization Techniques

To build high-performance systems for interactive analysis and exploration of big data, many methods have been developed to improve system performance. Data reduction methods, such as filtering [22], sampling [23], and data cubes [24], [25], are commonly used for analyzing large datasets. These methods require a priori knowledge of the dataset and may diminish opportunities for insight and knowledge discovery in data exploration. In addition, distributed systems and cloud computing can be employed for processing large datasets. Commercial products such as Spotfire [26] and Tableau [27] push data queries and transformations to cloud computing infrastructures to support interactive visualization. However, cloud computing causes overhead due to data transfer over the network, and it requires additional cost to the users. Especially for exploratory visual analysis, exchanging a large amount of data is often needed between server-side data processing and client-side rendering.

Other techniques can be used to hide system latency to support interactive performance. Multithreading and parallel processing in modern CPUs can be exploited to improve the performance of both data transformation and visualization [8], [28]. In addition, in-memory data caching [25], [29] can be used to cache data in system memory, avoiding the round trip to databases for every user interaction. For datasets that cannot fit into system memory, incremental visualization [30] can be applied to progressively visualize large data. P4 adapts both parallel processing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

and in-memory data caching techniques to leverage GPU computing for interactive visualization. Instead of caching data in system memory, P4 caches and manages data in GPU memory for parallel processing, similarly to image-based visualization techniques [31] that express data in pixel space.

Many methods leverage GPU computing to improve system performance. Fekete et al. [32] used GPU-based rendering techniques to improve the performance for visualizing a million items in treemaps and scatter plots. Govidara et al. [33] leveraged the graphics pipeline in GPUs for implementing common database operations. Hurter et al. [34] used GPU fragment shader prorgrams to visualize large sets of aircraft trajectories and support interactive queries. The imMens [35] system stores precomputed multidimensional data tiles as textures in GPU memory and use fragment shader programs for data filtering to support brushing and linking interactions between histograms and 2D heatmaps. These methods provided a basis for P4 to leverage GPU computing via shader programs for designing interactive information visualizations. To allow declarative grammar to be used with GPU computing, we combine techniques for GPU runtime code generation [36], [37], [38], [39] and data-parallel primitives [40] to create GPU programs based on user specifications.

3 Design

P4 is designed to make processing and visualizing large multidimensional data fast and easy. Figure 1 illustrates P4s architecture, which is constructed based on our system, data, and execution model designs.

3.1 System Model

The system model of P4 has the four primary components: 1) a translator for translating the JSON specifications into JavaScript function calls; 2) an API for JavaScript function calls; 3) a generator for converting user specifications into GPU programs at runtime; and 4) a controller for managing data input and output.

As a web-based toolkit, P4 provides a JavaScript API for application developers to build high-performance visualization systems. A portable JSON syntax is also supported for other programming languages and systems to easily generate design specifications. At runtime, P4 creates GPU programs based on user specifications of the visualization design. P4 effectively manage the execution and data flow of the GPU programs for utilizing the GPU to transform and visualize data.

3.2 Data Model

Internal data representation and management is the foundation of a visualization library. P4 uses a unified data model for representing multidimensional data in GPU memory. To efficiently access data in GPU memory, data is stored in column major order, where values of the same data dimension are adjacent to each other. This unified data model is used for input and output of all data transformation and visualization operations to ensure seamless interoperability between each operation. For example, the result of

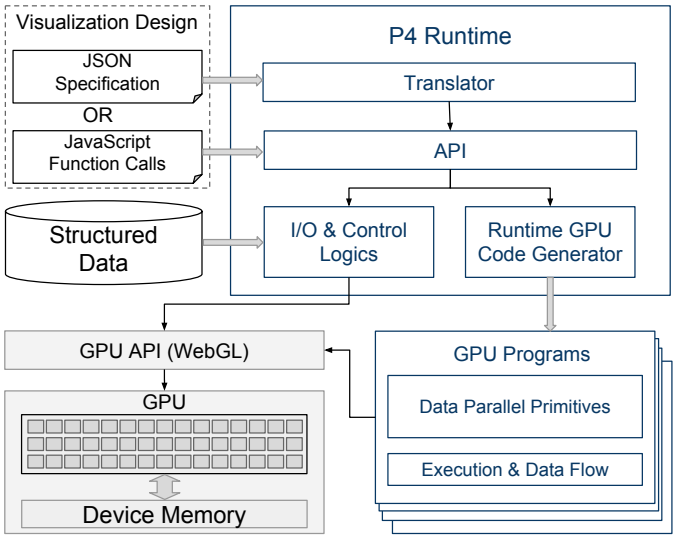


Fig. 1: P4 architecture and workflow.

data transformations can be directly mapped to different visual encoding channels for visualization, The data model supports both numerical and categorical data types. By default, a single-precision floating-point format is used for numerical attributes. For categorical attributes, P4 assigns integers to represent nominal categorical values, because it is easier and more efficient to manage integers than strings in GPU programs. Metadata, such as minimum and maximum values of each data attributes, is also stored in GPU memory for supporting operations such as normalization and aggregation.

3.3 Execution Model

A P4 program specified as a *pipeline* that contains a data schema and a set of operations for transformation and visualization. When executing a *pipeline*, P4 transparently parallelizes all the specified data transformation and visualization operations to effectively run on the GPU. The operations are executed by the user-specified order, where the output of the previous operation becomes the input of the next one. Using our data model, P4 stores and manages all data in the GPU memory and performs all operations within the GPU, such that no data transfer between the CPU and the GPU is required within a P4 *pipeline*.

4 Programming Interface

P4's programming interface provides a set of customizable operations for common data transformations, visual encodings, and interactions in information visualization. These operations can be performed on multiple data attributes with customizable operands. In this section, all the examples for presenting P4's programming interface use a dataset that contains 200,000 records of babies born in 2015 and their parents, which is part of the publicly-available datasets from U.S. National Center for Health Statistics. Figure 2 shows an example *pipeline* for using P4's JavaScript API to process a dataset and visualize the result using a bar chart. To use P4's data model for managing data in GPU memory, data types for all data attributes need to be first specified in a *pipeline*. P4's declarative grammar also allows users

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics


```
var pipeline = P4.pipeline().data({
  method: "http",
  path: "data/Natality.csv",
  attributes : {
    BabyWeight: "float",
    BabyGender: "string",
    MotherAge : "int",
    …,
    FatherAge : "int"
  }
});
```

```
pipeline
.derive({
  AgeDifference: function(d) {
    return d.FatherAge - d.MotherAge;
  }
})
.match({
  AgeDifference: [-10, 10]
})
.aggregate({
  $group: "AgeDifference",
  $collect: {
    BabyCount: { $count: "*" },
    AvgBabyWeight: { $avg: "BabyWeight" }
  }
})
.visualize({
    mark: "bar",
    x: "AgeDifference",
    y: "BabyCount",
    color: {
      field: "AvgBabyWeight",
      scheme: "viridis"
    }
})
```

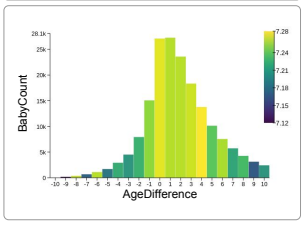


Fig. 2: An example for using P4's JavaScript API to perform analytical processing (deriving new variable, filtering, and aggregation) and visualize the analysis result as a bar chart.

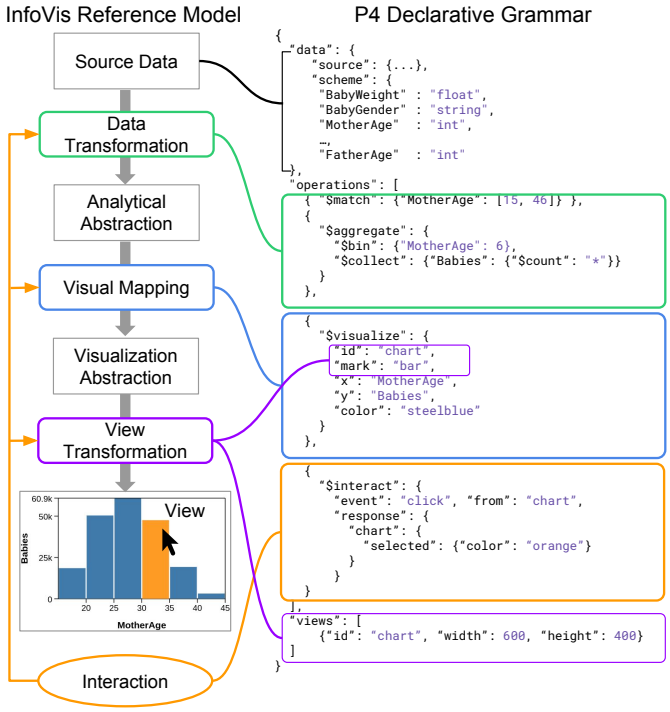


Fig. 3: An example showing P4's JSON syntax for designing interactive visualization, which follows the InfoVis Reference Model with separated specification for data transformation, visual mapping, and interaction.

to design interactive visualizations based on the InfoVis Reference Model [1], [41]. Figure 3 shows the declarative specification of a P4 *pipeline* in the JSON syntax for generating a histogram with a simple click interaction.

4.1 Data Transformation

To effectively specify data transformations, P4's declarative grammar uses a syntax similar to the JSONiq query language [42] and NoSQL document-oriented database [43]. Three operations for data transformations are provided:

Derive - calculate new values from existing attributes using user-defined logics.

Match - keep data records that meet all the conditions specified for the selected attributes.

Aggregate - group or bin data based on the specified attribute(s) and statistical methods.

```
.data({...})
.match({
  MotherAge: [12, 50],
  FatherAge: [12, 50]
})
.aggregate: ({
  $group: ["FatherAge", "MotherAge"],
  $collect: {
     Babies: { $count: "*" }
  }
})
.visualize({
  mark: "rect",
  x: "FatherAge",
  y: "MotherAge",
  color: "Babies"
})
```

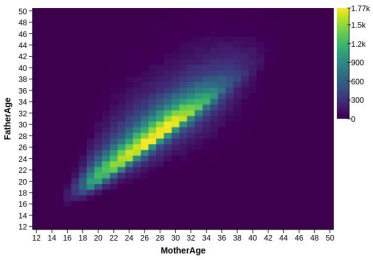


Fig. 4: Specification of a 2D heatmap visualization using the result from filtering and aggregation.

```
{
  "$visualize": {
    "mark": "line",
    "color": "steelblue",
    "opacity": 0.1,
    "y": [
      "BabyWeight",
      "MotherWeight",
      "MotherWgtGain",
      "MotherHeight"
    ],
    "brush": {
      "unselected": {
        "color": "lightgrey"
      }
    }
  }
}
```

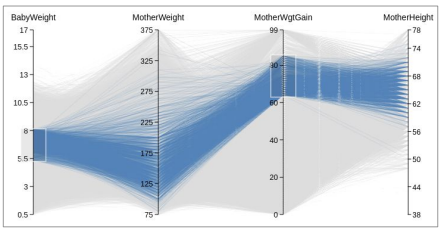


Fig. 5: Visualization specification of a parallel coordinates plot.

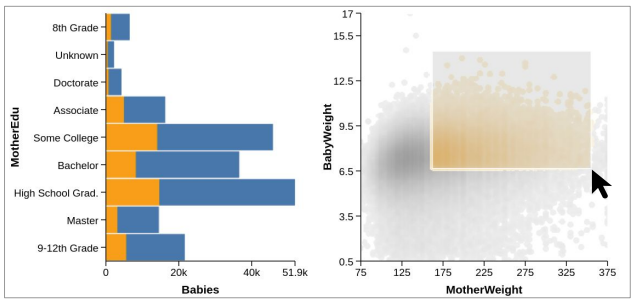
These operations can be used together in any arbitrary order for data querying and analytical processing. In the example shown in Figure 2, *Derive* is used to calculate the age difference between each baby's parents. This newly derived attribute is used in the next *Match* to filter out the data records with parents' age difference larger than 10 years. The matched data records are then grouped by the age differences to obtain the associated counts of babies and their average birth weights. The analysis result is shown in a bar chart for providing insights to the correlation of parent age difference to natality and the health of newborn babies (based on average birth weight).

As shown in the example in Figure 3, binning is used for numerical attributes instead of grouping in aggregation. The result is a histogram with 6 bins showing the baby counts based on the mother's age. In addition, two attributes can be used to group the data in aggregation as shown in Figure 4. The aggregation result can be visualized as a 2D heatmap for analyzing the natality respecting to the age of the parents.

4.2 Visual Mapping

The *Visualize* operation in the P4 grammar allows data attributes or transformation results to be easily mapped to visual encoding channels: color, opacity, width, height, and position x and y based on Cartesian coordinates. Currently, three types of visual marks are supported: circle, line, and rectangle. In addition, P4 allows the visual encoding of position x and y to be specified as an array of data attributes. This makes the GPU program to use an interleaved data fetching operation based on the order of the specified data attributes. For example, if we specify the mark to be line and position y to be a list of data attributes, the result is a Parallel Coordinates visualization shown in Figure 5. By default, linear scales are used for all the visual encodings. Power and log scales are allowed, and custom mapping can also be defined by providing a formula.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics




```
P4.pipeline()
.data({
  method: "http", path: "data/Natality.csv",
  scheme: {BabyWeight: "float", BabyGender: "string", MotherAge: "int",  ….,}
})
.aggregate({
    $group: "MotherEdu", $collect: { Babies: {"$count": "*"} }
})
.visualize({
    id: "chart1", mark: "bar", color: "steelblue",
    y: "MotherEdu", width: "Babies"
})
.reset()
.visualize({
    id: "chart2", mark: "circle", color: "orange", opacity: "auto",
    x: "MotherWeight", y: "BabyWeight"
})
.views([
    { id: "chart1", offset: [0, 0],   width: 360, height: 360 },
    { id: "chart2", offset: [360, 0], width: 360, height: 360 }
 ])
.interact({
    event: "brush", from: "chart1",
    response: {
        chart1: { selected: { color: "orange" } },
        chart2: { unselected: { color: "gray" } }
    }
})
```

Fig. 6: Specification of interaction for brushing a scatter plot to see the corresponding data highlighted in the bar chart.

4.3 Interaction

P4's declarative grammar allows a high-level specification of interaction design that covers a set of common tasks for direct manipulations of visualization. As shown in Figure 3, an interaction is specified as a pair of event and response, in which the event can be a click, hover, brush, zoom, or pan on any one of the visualization specified in the same *pipeline*. The response is specified as how each visualization (including the one that triggered the event) in the *pipeline* reacts to an event by changing its visual encoding. As an event is triggered, the associated data attributes and domains in the visualization are selected. For example, clicking on a bar chart selects the associated data subset, and brushing a scatter plot selects the data points projected within the brushed area. When specifying the response to an event, changes in the visual encoding for each visualization can be specified for both the selected or unselected portion of the data.

Figure 6 shows an example for specifying a brush interaction for the scatter plot, which changes the visual encoding in both a scatter plot and bar chart to highlight (in orange color) the brushed data. To keep the design specification simple when only one visualization is in the pipeline, the interaction can be embedded in the specification of visual mapping, as shown in Figure 5 (see "brush").

4.4 Control Flow

P4 provides control flow operations for managing complex work flows. In a P4 *pipeline*, intermediate results between operations are not saved in the GPU memory by default because this saves GPU memory for processing large data. A *Register* operation is provided for saving the states of a

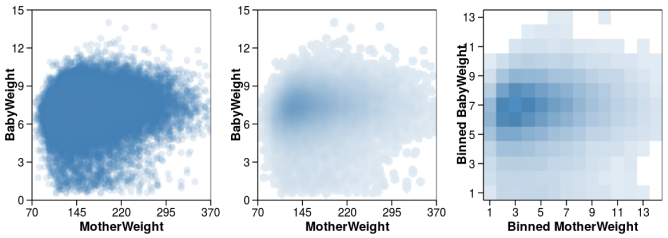


Fig. 7: Scatter plots with 200,000 dots: WebGL alpha blending (left), P4 opacity adjustment (center), and data binning technique (right) used in Liu et al. [35].

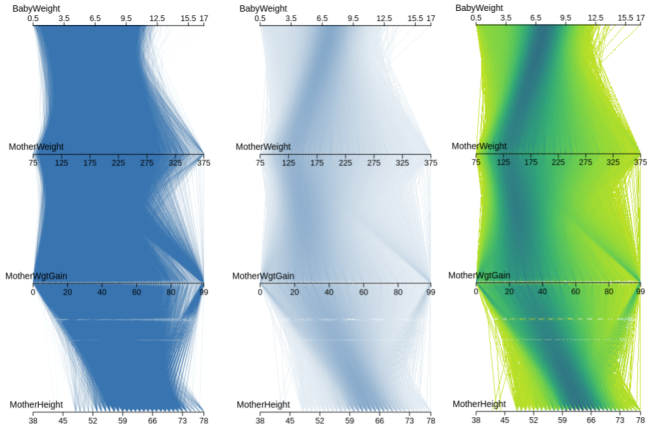


Fig. 8: Parallel coordinates plots with 200,000 lines: WebGL alpha blending (left), P4 opacity adjustment (center) , and P4 color mapping (right).

pipeline and caching intermediate results of data transformations. A *Resume* operation is provided for changing the current state and data input to the registered settings and values. As shown in Figure 6, the *Reset* operation (red line) changes the input of the second *Visualize* operation to use the original raw data, instead of the output of the aggregation. This is particularly useful for designing customized interactions that involve a set of data transformations and change the visualizations to show the updated results. In addition, a *Export* operation can be used to export the data processing results as JavaScript arrays or JSON to other libraries.

4.5 Perception Enhancement

Plotting a large number of marks often results in a visualization full of clutter. P4 provides an effective method to help better bring out patterns and trends in plots with large number of visual marks. The method is based on the perception enhancement technique used in Liu et al. [35], which maps data values or aggregation results to opacity. In P4, we modify this technique to reduce visual cluttering by adjusting either the opacity or color according to the normalized density and spatial distribution of visual marks. We first use the GPU to process all the pixels in the visualization to obtain the normalized density, and then we adjust the opacity or color based on the following formula:

$$\hat{\alpha} = (1 - \alpha_{min})(\frac{\rho}{\rho_{max}})^{\gamma} + \alpha_{min} \quad (1)$$

Here ρ_{max} denotes the maximum number of visual marks overlapped in the entire view, and ρ denotes the number of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

visual marks on the current pixel. As cubic roots can better approximate perceptual linearity [44], we set $\gamma = 1/3$ by default. Referring to prior experiments [45] on determining opacity for perception, the default value for α_{min} is set to 0.1 for clearly showing the pixels with minimum density. Figure 7 compares three different techniques for enhancing perception on scatter plots. As we can see, the alpha blending used by OpenGL or WebGL cannot reveal hot spots, and the data space technique based on data binning results in higher uncertainty. In P4's programming interface, setting opacity to "auto" applies our perception enhancement technique to the visualization. Another advantage of our view space perception enhancement technique is that it can be seamlessly applied to different type of visualizations. Figure 8 compares alpha blending to P4's techniques for enhancing perception on parallel coordinates plots. In addition to opacity, we can also use color to enhance the perception of cluttering visualizations.

5 Parallel Processing Framework

Most of the data transformations and visualization operations discussed in Section 4 are conceptually straightforward to implement. However, the complexity of GPU programming and the flexibility needed by declarative specifications make it challenging. Implementing each of the operations independently using ad-hoc methods requires significant programming effort. In addition, making these operations work effectively with one another is crucial to interactive performance. P4's parallel processing framework is designed to overcome these challenges. To support both high performance and flexibility, we combine the techniques of data-parallel primitives and runtime GPU code generation.

5.1 Data-Parallel Primitives

From our experience in developing interactive visualization applications, we have realized that most of the operations needed by information visualization, including both data transformation and visualization, can be implemented by functional programming primitives - map, filter, and reduce. Consequently, making these functions as data-parallel primitives for executing efficiently on the GPU can provide effective building blocks for implementing all the operations. In addition, an effective way for fetching data to the GPU's processing cores is needed, since P4 packages and stores data in GPU memory. The P4 framework currently has four data-parallel primitives:

Fetch is essential for each GPU program to retrieve the corresponding data records for parallel processing. Based on P4's storage mechanism and data attribute IDs, it calculates the data location in GPU memory for fetching data.

Map is particularly useful to compute intermediate values needed by operations. The intermediate values can be the derived data values based on user-defined logics or the memory location for storing the output values.

Filter provides a way for a GPU program to select data according to the specific criteria or condition. Instead of discarding the data after retrieving from GPU memory, it also reduces memory bandwidth by preventing unwanted data to be fetched.

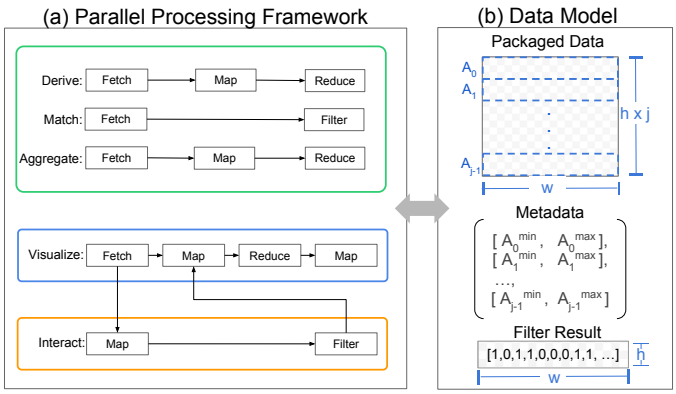


Fig. 9: P4's parallel processing framework(a) and data model (b).

Reduce allows a GPU program to process an array of data values in parallel to produce a single output value using a specific statistical method, such as counting, summing, or averaging.

These data-parallel primitives can be combined to construct different algorithms or high-level operations. Instead of making each data parallel primitive performs a fixed unit task, we use runtime GPU code generation to define the task based on the operation and user-defined logics.

5.2 Runtime GPU Code Generation

P4's declarative grammar provides flexibility to apply user-defined logics for data transformations and visual encodings at runtime. As in the example shown in Figure 2, the new attribute *"AgeDifference"* is derived from existing attributes with user-defined logics, and this new attribute can be used for other operations in the rest of the pipeline. To support such flexibility while parallelizing computations on the GPU, we adapt runtime GPU code generation techniques for embedding the user-defined logics in the GPU program. For each data transformation and visualization operation, we first construct the work flow and execution pattern using the data-parallel primitives. A default task is defined for each primitive, but the runtime GPU code generator can redefine the task to embed the user-defined logics if provided in the design specification. This methodology becomes clearer when we discuss our WebGL implementation in Section 6.

5.3 Parallel Transformation and Visualization

In P4's parallel processing framework, data transformation and visualization operations are constructed by different combinations of the data-parallel primitives. Figure 9(a) illustrates how each operation discussed in Section 4 are constructed.

Derive takes a formula or a function for creating a new data attribute using user-defined logics with existing data attributes. *Fetch* is used to obtain the values for the existing data attributes, and *Map* is used to derive the new values using the user-defined logics. Since P4 needs to compute the metadata (e.g., minimum and maximum values) for the new data, all the derived data values for a new attribute are assigned with the same key for *Reduce* to obtain the metadata.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

Match takes a set of criteria to filter data. By parsing these criteria, *Fetch* is used to obtain the values of specified data attributes, and *Filter* is used to determine whether or not the values meet the criteria.

Aggregate takes one or more data attributes as the keys to group data and then processes the data in each group using the specified reduction methods. By parsing the specification for Aggregate, P4 sets the data attributes for *Fetch* and uses *Map* for assigning each data value to the corresponding group. Finally, *Reduce* is used to obtain the reduction results in each group.

Visualize takes a set of visual encodings for mapping data attributes to visual channels. Based on the visual encoding, the encoded data attributes are used for *Fetch*. By default, *Map* is used for interpolating the positions and other encodings of the visual marks based on data values. If a user-defined mapping is provided, *Map* is changed at runtime. For perception enhancement, *Reduce* is used to determine the minimum and maximum density values in the visualization, and another *Map* is used to calculate the output value for each pixel based on Equation 1.

Interact takes an event-response pair to change the visual encodings for the selected portion of data rendered on a visualization. It employs the same combination of the data-parallel primitives used by *Visualize* with an extra *Map* and a *Filter*. The extra *Map* obtains the extents of the data domains selected by the interaction. The *Filter* uses these extents to mark the data for changing the visual encoding according to the response of the interaction.

6 Implementation

Since P4 is designed to be a web-based toolkit, our implementation is based on WebGL 1.0, the only standardized GPU API currently available in most modern web browsers. Here we describe how P4 is implemented based on our parallel processing framework to enable GPU-based data transformation and visualization.

6.1 Data Management

P4 packs data as 2D WebGL textures for storing in GPU memory. Floating-point textures are used, which can support 24-bit precision with values ranging from 2^{-127} to 2^{127}. To pack multidimensional data in 2D textures, we use the data model of column arrays to arrange data based on the attribute orders, as shown Figure 9(b). Each pixel in a 2D texture stores a data value. A 2D texture is divided vertically into partitions with w width and h height, with each partition allocating for a data attribute A_i. Since a texture size of $8,192^2$ is typically supported in modern graphic cards, we set the default value of w to be 8,192. For data size exceeding $8,192^2$, the data are partitioned into multiple textures. The metadata of the minimum and maximum values for all data attributes are stored in GPU memory as WebGL Uniform Buffer objects. For data transformation operations, the results are written to the textures attached to the off-screen frame buffer, which can be used as input for the next operations. P4 manages the pointers to the input and output texture objects to ensure the correct data flows. As all the inputs and outputs of data transformation operations use a unified data model, the visualization operations can seamlessly inter-operate with the data transformation operations, leveraging the high memory bandwidth and massive computing power of the GPU to efficiently visualize a large number of visual marks.

6.2 Visualization Rendering

To generate visualizations on web browsers, P4 uses WebGL to render visual marks on HTML5 Canvas elements. Since the axes and labels do not need parallel rendering due to the small number of visual items, they are rendered as SVG elements. Therefore, a copy of the metadata is stored in the system memory, including the minimum and maximum values of the numerical attributes for rendering of the axes and labels in SVG. Since integers are used to represent categorical data (strings) in GPU memory, the metadata stored in the system memory also contains all the references to categorical attributes, where axes and labels can reference the metadata to show the string values of the categorical attributes. In addition, multiple visualizations share the same data source and data transformation results without duplicates. Ideally each visualization should be rendered to its own Canvas. However, resource sharing between different WebGL contexts and canvases is not supported in the current version of WebGL. Since each P4 *pipeline* can only be initialized with one data source, all visualizations for this data source and the results from data transformations are rendered to the same WebGL Canvas. P4 partitions the Canvas based on the view configuration provided by the users to render multiple visualizations for the same dataset. For multiple visualizations for multiple datasets, multiple P4 *pipelines* can be used.

6.3 GPU Shader Programs

P4 uses WebGL shader programs for both data processing and rendering. We leverage the WebGL pipeline to implement our parallel processing framework with the four data-parallel primitives discussed in Section 5. For *Fetch*, we take advantage of the efficient mechanism provided to the shader for accessing textures. For *Map*, we leverage the vertex shader for computing intermediate results, and use the fragment shader to write the results to corresponding locations. For *Filter*, we use the vertex shader to check if the data record matches the specified criteria and use the fragment shader to save the filter result. For *Reduce*, we leverage the capability of the blending unit to obtain the count, sum, maximum, and minimum values for each pixel on the output framebuffer. We can also obtain the average and variance using two passes, in which we get the count and sum in the first pass.

Figure 10 illustrates how P4 generates the GPU codes from user specifications for aggregating and visualizing data via the WebGL pipeline. The aggregation workflow (a) is composed of three data-parallel primitives (*Fetch*, *Map*, and *Reduce*) for processing the group-by transformation (b). At runtime, P4 generates GPU code (c) for the aggregation. The vertex shader program (c1) performs *Fetch* and *Map*, and the fragment shader program (c2) writes the fetched values to the corresponding output location based on the result of *Map*. To obtain the counts and sums for calculating

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

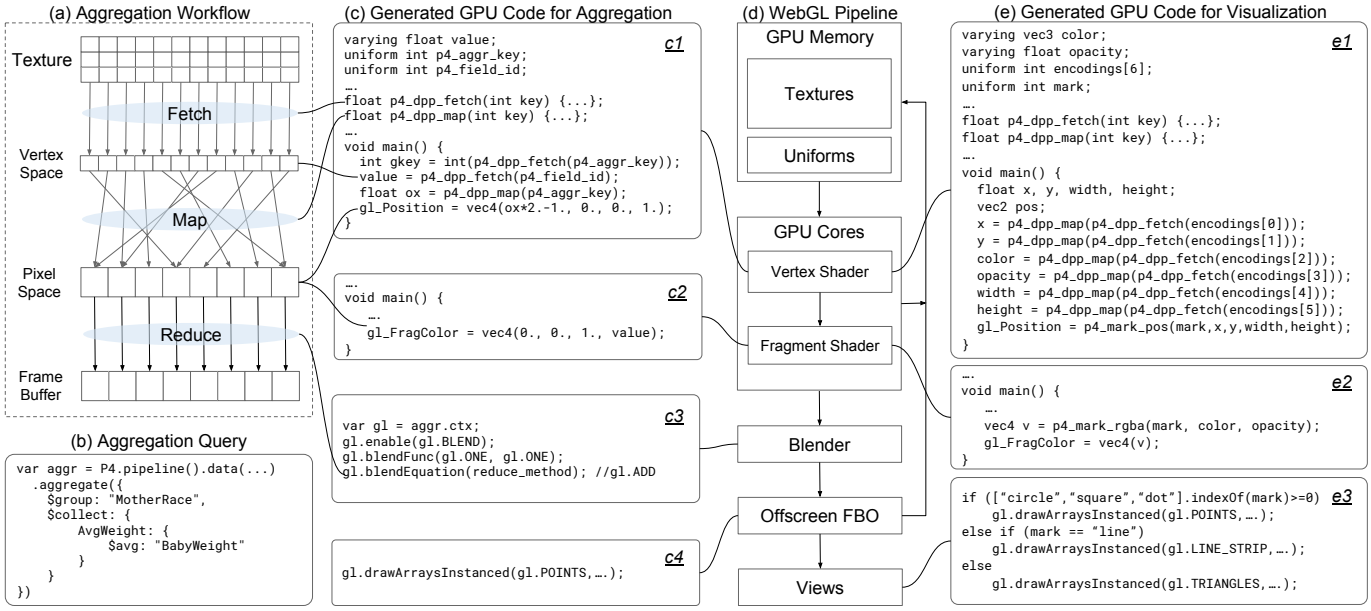


Fig. 10: Examples of runtime GPU code generation. The aggregation Workflow (a) for the, (aggregation query (b) generates the GPU codes (c) to effectively utilize the WebGL pipeline for grouping data and obtaining the average values in each group. In a similar way, the generated GPU code for visualization (d) renders visual marks based the visual encodings.

the average values with *Reduce*, the WebGL blending (c3) is set to use addition. When the WebGL draw call (c4) is executed, the vertex and fragment shaders and blender are utilized to write the aggregation result to the framebuffer. For the GPU code generated for visualization (e), the WebGL pipeline is utilized in a similar way. The vertex shader program (e1) uses *Map* for obtaining the normalized values based on the visual encodings and sets the position of visual marks. The fragment shader program (e2) sets the color, opacity, and shape of the visual marks. Base on the specified type of visual marks, P4 changes the visualization operation to use the corresponding WebGL draw call (e3).

In addition to being the building blocks for constructing different operations, data-parallel primitives also make runtime code generation easier. Figure 11 shows an example for generating the vertex shader program for a *Derive* operation. The user-defined logics for deriving the new data attribute are used to define the function for *Map* in the vertex shader program. When generating GPU codes using data parallel primitives, the main function in the shader program does not need to be changed, because the sub-functions of the data parallel primitives can be changed based on user specifications. Similarly in Figure 12, the user-defined criteria for *Match* is used to define the function for *Filter* in the vertex shader program. In this example, the criteria for *Match* depends on a categorical data attribute. Therefore, the indexes for the categorical data is uploaded to the GPU using WebGL Uniforms (Figure 12b).

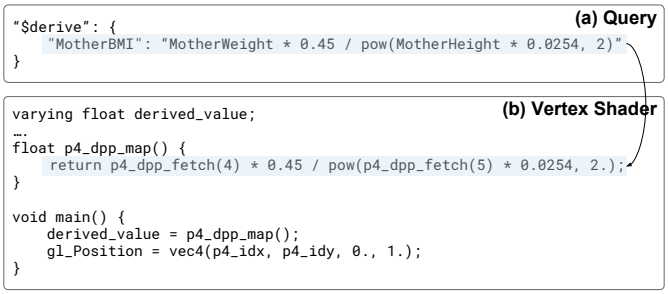


Fig. 11: P4's runtime GPU code generator embeds the user-defined logics of the Derive operation in the vertex shader (b).

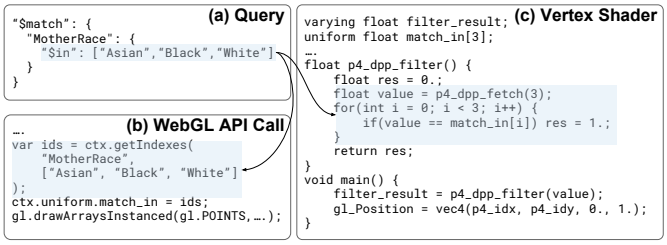


Fig. 12: Based on the user-defined criteria (a) for the Match operation, P4's runtime GPU code generator sets the criteria as WebGL uniforms (b) and define these criteria for the Filter primitive in the vertex shader(c).

7 Performance Benchmarks

To evaluate the performance of P4 for both data transformation and visualization, we compare P4 to four existing state-of-the-art libraries (Table 1). To evaluate portability, we benchmark P4 using four different grades of GPUs (Table 2).

7.1 Benchmark Methods

Modern web browsers commonly use asynchronous, non-blocking API calls to access the GPU. Since the execution of GPU-based computations is managed by WebGL, a method is needed to synchronize between CPU and GPU in order measure the actual execution time of a GPU program. To do this, we intentionally read a random byte of the result from the GPU memory after each operation, which ensures the GPU program has completed writing the result. Similarly

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

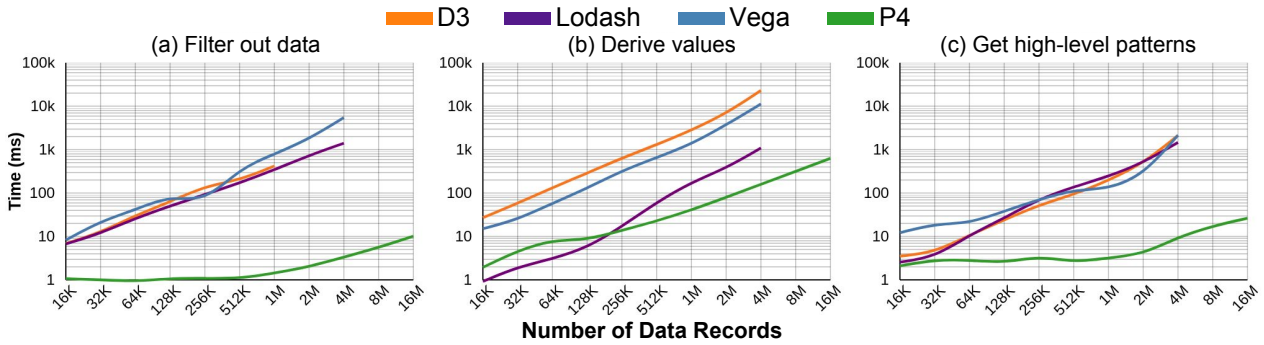


Fig. 13: Average time for P4 and other three stat-of-the-art libraries to perform three different data transformations.

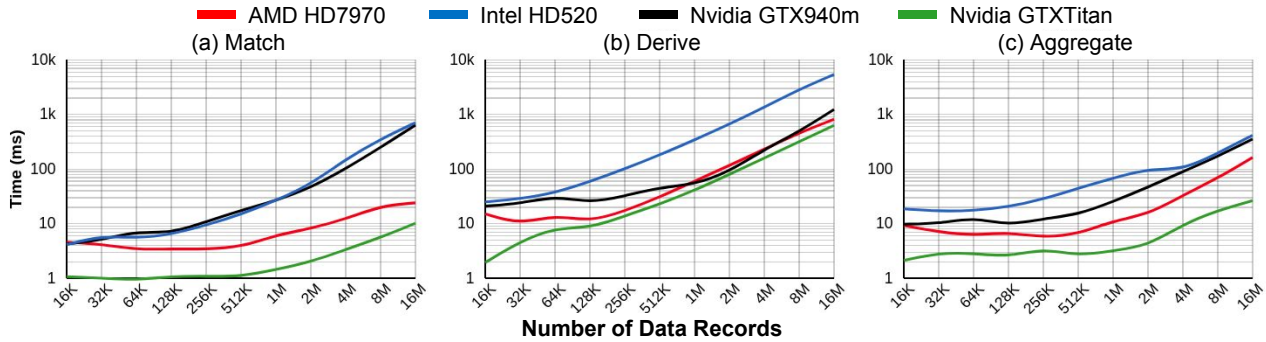


Fig. 14: Average time for P4 running on four different grades of GPU to perform three different data transformations.

Library	Version	D.T.	Vis.	Website
D3 [4]	3.5.17	Yes	Yes	d3js.org
Vega [6]	3.0.2	Yes	Yes	vega.github.io/vega/
Lodash	4.17.4	Yes	No	lodash.com
Stardust [21]	0.1.1	No	Yes	stardustjs.github.io

TABLE 1: P4 is compared to these four libraries for data transformation (D.T.) and visualization (Vis.) performance.

Vendor	Model	Cores	GPU Clock	Memory
Intel	HD520	192	1050 MHz	1GB DDR3
Nvidia	GTX940m	384	1070 MHz	2GB DDR3
AMD	HD7970	2048	925 MHz	3GB DDR5
Nvidia	GTXTitan	2688	836 MHz	6GB DDR5

TABLE 2: P4 is evaluated on these four graphics processors.

for WebGL-based visualizations, we randomly read a pixel from the view port after rendering. Normally, P4 does not transfer the result from the GPU memory to the CPU side after completing an operation, but this provides a useful bound for evaluating the performance of P4.

Each benchmark is tested with 10 trials per record size, with the size varying from 16K to 16M records with each record having four data attributes. All our benchmarks were performed in Google Chrome 61.0.3163.100 (64-bit). When comparing P4 against other libraries, we ran our benchmarks on a desktop computer with a quad-core 3.6 GHz Intel i7-4790 CPU and 32 GB RAM. The computer has a PCI Express Nvidia GTX Titan graphic card with 6GB video RAM.

7.2 Data Transformation Performance

For data transformation, we compared P4 against D3, Vega, and Lodash. D3 and Vega are two popular libraries for information visualization. D3 provides a set of helper functions that can be used for implementing common data transformations. Vega explicitly supports data transformations where users can specify operations using its declarative grammar. Lodash is a widely used data processing library for developing web-based applications, including information visualization and visual analytics systems.

To compare performance, we selected three data transformation tasks from Heer and Shneiderman's taxonomy of interactive dynamics for visual analysis [46]: (1) filter out data to focus on relevant items, (2) derive values from source data, and (3) navigate to examine high-level patterns. For D3, Vega, and Lodash, we follow the documentation and examples in their websites to implement these three tasks. For P4, these three tasks can be easily implemented using the *Match*, *Derive*, and *Aggregate* operations, respectively. The results in Figure 13 show the performance of P4 compared to D3, Vega, and Lodash. P4 performs significantly faster, especially for data size larger than one million records, where P4 is at least 10X faster than the other three libraries. Due to browser time limitation, D3, Vega, and Loadash cannot handle data with more than 4M records, while P4 can handle up to 16M records. For filtering out data, the performance of P4 is two order of magnitude higher. For deriving new values, Lodash is slightly faster for 128K data records or less, but P4 is much faster for larger data sizes. For showing high-level patterns (aggregation), P4 performs 10X to 100X faster as data size increases.

Figure 14 shows P4's performance running on four different grades of GPUs. The results show that P4 can achieve higher performance on more powerful GPUs. Even for integrated GPUs inside the CPU chip (Intel HD520) and GPUs for mobiles and laptops (Nvidia GTX940m), P4 performs 5X to 10X faster than other libraries for large data sizes. For *Match* and *Aggregate*, P4's performance is relatively constant for small data sizes. The computation time starts to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

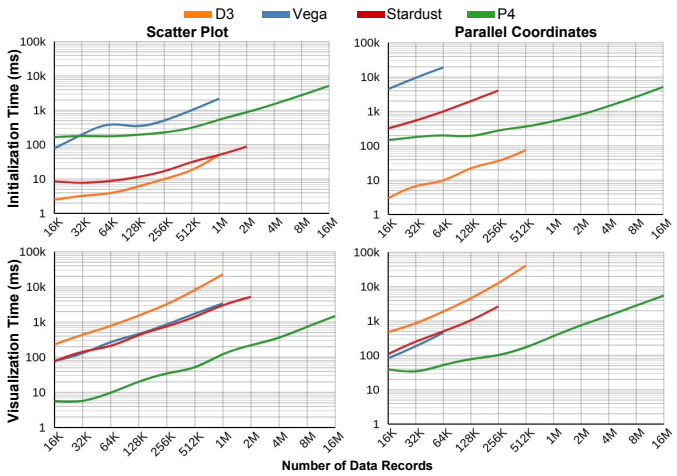


Fig. 15: Average initialization and frame time for P4 and other three stat-of-the-art visualization libraries to visualize scatter plots and parallel coordinates plots.

increase at particularly larger data sizes, depending on the computing power of the GPUs and the performance of the hosted system. With small data sizes, the overhead caused by P4 at runtime dominates the computation time. As data size increases, the computation time for data processing becomes higher and reduce the impact of the overhead on the overall performance. From *Derive*, we noticed that the performance numbers on different GPUs are similar, where more powerful GPUs (e.g., AMD HD7970 and Nvidia GTX Titan) did not have significant performance gain. This is because *Derive* needs to allocate extra GPU memory for storing the derived values, which results in a large overhead to the performance. For optimizing the performance of *Derive*, we can preallocate extra GPU memory to hide the latency. It is also possible to let users decide whether preallocation is used or not for trading memory space for better performance. We plan to further improve the performance of *Derive* along with other performance optimizations in our future work.

7.3 Visualization Performance

We compare the performance of P4, D3, Vega, and Stardust for visualizing scatter plots and parallel coordinates plots. While D3 is based on SVG, Stardust provides a similar API to D3 but use WebGL for rendering. Vega can use both SVG and Canvas for rendering, but it is measured to be more efficient when rendering to Canvas. Therefore, we configure Vega to use Canvas for rendering in this benchmark.

Figure 15 shows the benchmarking results of D3, Vega, Stardust, and P4 for visualizing scatter plots and parallel coordinates plots. For each visualization, we measure the initialization time and average frame time. Due to memory allocation or browser time limit, D3, Vega, and Stardust cannot visualize data with more than 2M records for scatter plot, and they cannot visualize data with more than 512K for parallel coordinates plot. Only P4 can visualize both parallel coordinates and scatter plot for data with 16M data records in this benchmark.

P4 has the same initialization for both parallel coordinates and scatter plots because it stores all the data dimensions in GPU memory, In addition, both P4 and Stardust

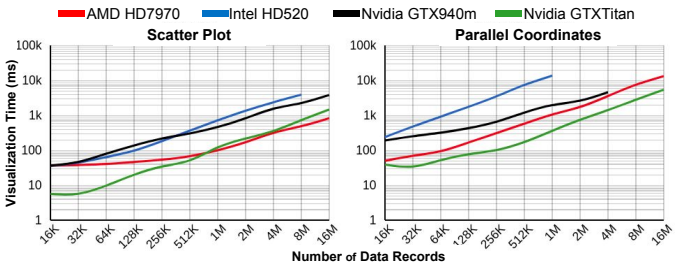


Fig. 16: Average initialization and frame time for P4 to visualize scatter plots and parallel coordinates plots.

need to generate and compile WebGL shaders. Therefore, D3 has the fastest initialization time since it has the least overhead for setting up visualizations, and Vega has the slowest initialization since it needs to parse the JSON specifications and arrange data for supporting reactive workflow. P4 is slower than Stardust for initializing scatter plots because P4 needs to transfer the data values for all the data attributes to GPU memory even if only two data attributes are used for visualization. For parallel coordinates plots, Stardust does not provide a way to interleave different data attributes in the shader programs like P4 does, so it requires some data rearrangement in order to visualize the plot. As a result, P4 is faster than Stardust for initializing parallel coordinates plots.

In P4, each pipeline only needs to be initialized once, and initialization is not required for a change of visualization types. Therefore, the visualization time is more important since it is crucial to interactive performance. It is also important to measure the actual visualization time rather than just the rendering time. The difference between visualization time and rendering time can be explained using the InfoVis Reference Model in Figure 3. While the rendering time only measures the performance of visual transformation, the visualization time measures performance of both visual mapping and visual transformation. P4 is about 10X faster than Vega and Stardust and about 50X faster than D3 (Figure 15). Both P4 and Stardust use WebGL for rendering, but their approaches are different. P4 performs both visual mapping and transformation in a single operation while Stardust uses multiple passes. In addition, Stardust leverages the vertex shader for constructing geometries (e.g., circles and squares), while P4 uses a pixel-based approach that leverages the fragment shader. As a result, P4 performs better than Stardust when visualizing scatter plots. For parallel coordinates, P4 can effectively interleave data dimensions for visualizing lines based on vertices mapped to different data attributes. This makes P4 outperform Stardust when visualizing parallel coordinates plots.

Figure 16 compares the visualization performance of P4 running on four different GPUs. Similar to what we saw for data transformation, P4 achieves better performance on more powerful GPUs. Due to browser time limit, Intel HD520 (integrated GPU) cannot visualize data with 16M records for scatter plot nor data with more than 2M records for parallel coordinates plots. On the other hand, Nvidia GTX 940m cannot visualize data with 4M records or more for parallel coordinates plots within the browser time limit.

For interactive visualization, user interactions can trigger different data transformations and updates to the visual-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

ization. Therefore, the time for both data transformation and visualization directly impacts the interactive performance. As P4 provides much better performance for both data transformation and visualization compared to other libraries, interactive performance can be greatly improved.

8 Applications

We have built several visualization applications using P4 over the course of its development. Here we present three of these applications to demonstrate P4's applicability for developing high-performance interactive systems.

8.1 Supercomputing Network Behavior

Supercomputing technology is essential to big data analytics and scientific discoveries. The overall efficiency of supercomputers largely relies on the performance of the interconnection networks. Based on P4, we have built an interactive web-based system to support explorations of the design space of such networks. To explore the behaviors of supercomputers for optimizing performance, supercomputing researchers typically collect data from either production supercomputers or accurate discrete-event simulations with sophisticated models. The data is often time-varying and multidimensional with millions of records. With P4, such large and complex data can be effectively visualized in multiple coordinated views for interactive exploration.

Using our system with supercomputing researchers, we have studied the behavior of Dragonfly [47] networks with various network scales, routing strategies, job placement policies, and parallel application workloads. Dragonfly network is a popular choice for building modern and next-generation supercomputers (e.g., exascale systems). While automated techniques cannot easily be applied to design space exploration, visual analysis techniques allows supercomputing researchers to apply their domain expertise and background knowledge for interactively exploring the network performance data and evaluating different design choices. With the insights gained from our exploration and analysis, we are able to develop optimization strategies to improve the performance and efficiency of Dragonfly networks. Our system design, visual analysis techniques, analysis results, and optimization strategies are published in Li et al. [48].

Figure 17 shows the user interface of the system and illustrates how P4 is used to process and visualize temporal statistics and individual network entities (e.g., network links and terminals), and at the same time aggregate the selected data and output the results to D3 for visualizing the network structures using concentric radial layouts (Figure 17b). Since P4 is efficient in processing and visualizing massive amount of data items while D3 is handy for creating advanced visualization layouts with a relatively small number of items, using both libraries together can effectively realize our system design. The implementation of this system demonstrates the interoperability between P4 and other visualization libraries. Statistical analysis results obtained by data processing and aggregation in P4 can be exported to another library to create customized visualizations for providing visual summaries and overviews. However, the

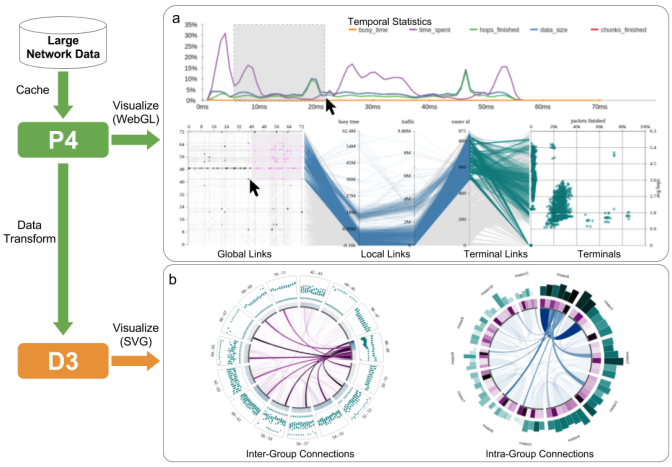


Fig. 17: P4 is used to process and visualize large network performance data. Aggregated results processed by P4 are passed to D3 for generating a more explanatory view for summarizing the network.

overhead of transferring a large amount of data from GPU to CPU can impact system performance. System designers and programmers need to be aware of this overhead and should only export data aggregation results from P4 to other libraries. Since the data size of aggregation results is usually small, the overhead due to data transferring is negligible.

8.2 Large-Scale Discrete-Event Simulation

Parallel discrete-event simulations (PDES) allows scientists to model and study complex physical phenomena. The scalability of PDES is important to analyzing large-scale phenomena. Collaborating with a team of PDES developers, we have created a visual analytics system for assessing and improving the efficiency and scalability of large-scale PDES. The system design and the associated visual analytics techniques are published in Ross et al. [49]. In the first version of this system, most data processing relied on CPU-based computations, which resulted in high system latency which adversely affects the exploration process. To improve interactive performance, P4 is used for data processing and management for supporting visualizations and interactions.

Improving the scalability of large-scale PDES requires analysis of large multivariate time-series data generated by the PDES engine about each program parameters used for controlling the simulation. The time-series data is collected from two temporal perspectives: the virtual time of the simulation and the real time of the simulation program running on a distributed computing system. As we can see from the user interface of the system shown in Figure 18, various user interactions are provided to analyze the PDES performance data, including connecting the temporal statistics between virtual and real time domains as well as selecting a subset of data for detailed investigation. These user interactions trigger a series of operations for transforming large amount of data to update the visualizations, which can be implemented using P4 in a straightforward way. If these data transformations were to rely on CPU-based computations, the system could only handle data with very low temporal resolution. The performance provided by P4 allows exploration of simulations with much higher temporal resolution at interactive speed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2018.2871139, IEEE Transactions on Visualization and Computer Graphics

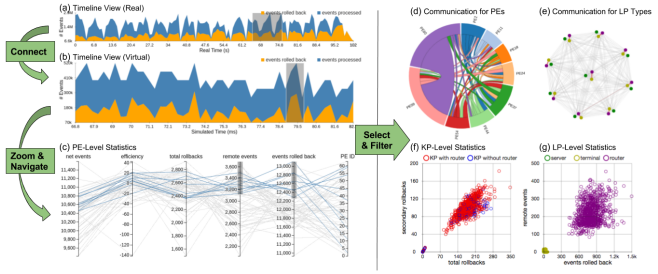


Fig. 18: User interactions, such as select, zoom, and connect, are provided in our PDES visualization system [49] for analyzing simulator performance data.

8.3 P4 Player

In parallel to the development of P4, we have created P4 Player, a web application for designing interactive visualizations using P4's JSON syntax to explore multidimensional datasets. The user interface of P4 Player is composed of an editor and a visualization dashboard. The editor is for inputting design specification via P4's declarative grammar, and the corresponding visualizations are displayed in the dashboard. This facilitates a process of edit, run, and repeat for iteratively exploring a dataset. Users can freely create and save design specifications for different datasets. In addition to load data from web servers, P4 Player leverages the HTML5 File API [50] for users to load data files from local hard disks, which does not require data uploads via the internet. This avoids the network bandwidth being a bottleneck for processing and visualizing large data. P4 Player also supports different file formats, including JSON, CSV, and binary files. The implementation of P4 Player demonstrates the interoperability between P4 and modern web technologies. Other web applications can use a similar approach to build high-performance systems using P4.

9 Discussion

Here we discuss the trade-off between expressiveness and performance, as well as the possible extensions and current limitations of P4.

9.1 Expressiveness and Performance

While low-level visualization grammars with fine-grained controls are needed for explanatory visualization, high-level and declarative grammars with concise expressions are more useful for exploratory visualization. By allowing GPU computing to be used with a declarative grammar with a concise syntax, P4 is particularly useful for exploratory visual analysis of big data. Comparing with P4, D3 provides better expressiveness for visualization representations, while Vega and Vega-Lite have more capabilities for both data transformation and visualization. These libraries inherit the expressiveness from the document object model (DOM) and JavaScript. However, the DOM is not designed for high performance, and thus cannot support large data visualization. To provide better scalability, P4 uses a specialized framework to offload both data processing and visualization to the GPU. In P4, we only use the DOM for configuring view positions and representing chart axes, legends, and other meta-visualizations. This design makes implementation simple. GPU computing is only used to mitigate performance bottlenecks, and the DOM is used for other components that do not require GPU computing.

As P4 cannot inherit the expressiveness from the DOM and JavaScript, we provide a declarative grammar for concise specification of interactive visualization. P4 aims to provide the same level of expressiveness and conciseness in Vega-Lite, but our declarative grammar requires the dataset schema (data types and attribute names) to be first specified for ensuring efficient data management in GPU memory. In addition, P4 strictly separates the specifications of data transformation, visual mapping, and interaction, so each operation can be parallelized to run on the GPU. Unlike the syntax in D3 or Vega-Lite where the specifications of data transformation and visual mapping are often coupled together, P4 provides a cleaner syntax for designing information visualization as well as tracking provenance.

The development of many visualization libraries tend to focus ensuring expressiveness and attempt to seek performance improvement later. However, the improvement of performance is often limited in such an approach. In the development of P4, we first built a high-performance framework and then extend it to add more expressiveness.

9.2 Optimization and Extension

We have implemented P4 using WebGL 1.0 since it is currently the only standardized interface to access GPU computing in modern web browsers. As WebGL 2.0 is soon to be available in web browsers which enables access to more features in modern GPUs, we can relax or remove some of the current limitations in P4 while optimizing the performance. However, many advanced features are still not available, such as shared GPU resources between contexts and Compute Shader. In particular, we use the blending operations provided by WebGL to implement the *Reduce* primitive, which can only support a few metrics, such as min, max, count, sum, and average. Although it is sufficient to support a large set of the common operations for information visualization, aggregation is limited to these metrics. If Compute Shader or programmable blending is supported in future WebGL versions, we can change the implementation of the *Reduce* primitive to provide more options and flexibility for aggregation. Future parallel computing interfaces in web browsers will also permit alternative implementations. Beyond web-based systems, our approach of using data-parallel primitives and runtime code generation can be applied to standard desktop contexts. The features for the high-level operations of data transformation and visualization remain the same with different implementations and underlying architectures.

In addition to optimizing performance, P4 can be easily extended. New data-parallel primitives can be added to P4's parallel processing framework. New high-level operations can also be added by leveraging existing and new data-parallel primitives. People with knowledge in GPU programming and parallel computing can add new data parallel primitives, and people in the field of information visualization can add new high-level operations. This allows researchers and developers from the fields of information visualization and high-performance computing to work to-

gether for advancing data analytics and visualization technologies.

P4 should be easily adopted by application developers and data analysts given its simplicity. Public deployment and collecting user feedback can help us improve and extend our programming interface and declarative grammar. We also plan to create standardization for others to write extensions and plug-ins for P4, so the users of P4 can write and share custom methods based on our parallel processing framework.

9.3 Limitations and Future Work

A drawback in our approach is that the performance gain achieved by P4 depends on the computing power of the GPU on the underlying device. GPUs in today's mainstream desktop and laptop computers can achieve good speedup, but devices with less powerful GPUs, such as smart phones and tablet computers, might not have sufficient computing power to handle datasets with millions of records. In addition, because P4 manages all the data in GPU memory, the GPU memory capacity limits the data size can be handled. When the data size cannot be handled by the GPU's computing power or memory capacity, data preprocessing and progressive visual analytics can be used to support interactive exploration. To allow data preprocessing in P4, we can extend our syntax for specifying data preprocessing when initializing a P4 *pipeline*. Nevertheless, similar to the precomputed data tile method used in Immense [35], data preprocessing limits the flexibility of data exploration and requires users to have background knowledge about the dataset. To support progressive processing, we can partition the data in chunks that can fit into GPU memory, and then we can incrementally process each chunk and update the result and visualizations. Interactions to the visualizations can also be supported for users to explore the intermediate results. In addition to the GPU memory limit, the execution time limit in web browsers should also be considered when determining the time interval for progressively updating visualizations. As we learned from our performance benchmark, very large datasets may cause the computation time to exceed what web browsers typically allow when using GPUs with less computing power.

Although the available operations in our current implementation can support a large set of common visualization and exploration tasks, the expressiveness for specifying highly customized data transformations and visualizations is limited. For data transformation, embarrassingly parallel operations can be easily added to P4, but operations that are not embarrassingly parallel are more challenging to implement. This requires further improvement on P4's framework as well as adding more advanced data-parallel primitives. For example, a scan primitive can be added to realize effective parallel sort. For visualization, our current implementation only support parallel projection based on Cartesian coordinates and a few types of visual mark. Future extension can add support for Polar coordinates and more types of visual mark. More complicated visualization layouts that are not based on parallel projection (e.g., treemap) can also be supported by adding new visualization operations.

10 Conclusion

We have introduced P4, a high-performance and flexible toolkit for declarative design of interactive information visualization. While most existing toolkits mainly focus on expressiveness but pay little attention to the importance of interactive performance, we combine GPU computing and declarative grammar in P4 to facilitate the development of information visualizations and visual analytics systems. We have also contributed an extensible parallel processing framework that combines data-parallel primitives and runtime code generation to make GPUs as flexible and efficient accelerators for data transformations and making visualizations. Our benchmark results show that P4 is at least an order of magnitude faster than the current state of the art tools in both data transformation and visualization. In addition, we have shown three visualization applications built with P4 for analyzing and exploring large data. P4 is an open source software available at https://jpkli.github.io/p4. As we and others further extend P4's functionality and capability, data visualization in P4 with GPU acceleration will become an even more attractive tool for making sense of complex information and discovering new knowledge.

Acknowledgments

The authors thank Chris Ye and Min Shih for their insight and support. This research is sponsored in part by the U.S. National Science Foundation through grants IIS-1741536 and IIS-1528203, and the U.S. Department of Energy through grant DESC0014917.

References

[1] S. K. Card, J. D. Mackinlay, and B. Shneiderman, *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.
[2] S. K. Card, A. Newell, and T. P. Moran, "The psychology of human-computer interaction," 1983.
[3] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2122–2131, 2014.
[4] M. Bostock, V. Ogievetsky, and J. Heer, "D^3: Data-Driven Documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, 2011.
[5] H. Wickham, *ggplot2: elegant graphics for data analysis*. Springer, 2016.
[6] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer, "Reactive vega: A streaming dataflow architecture for declarative interactive visualization," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 659–668, 2016.
[7] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 341–350, 2017.
[8] J. Heer and M. Bostock, "Declarative language design for interactive visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1149–1156, 2010.
[9] L. Wilkinson, *The Grammar of Graphics*. Springer-Verlag, Inc., 1999.
[10] M. Bostock and J. Heer, "Protovis: A graphical toolkit for visualization," *IEEE transactions on visualization and computer graphics*, vol. 15, no. 6, 2009.
[11] M. Harris, "Gpgpu: General-purpose computation on gpus," *SIGGRAPH 2005 GPGPU COURSE*, pp. 1–51, 2005.
[12] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
[13] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] Khronos. WebGL Specification. https://www.khronos.org/webgl/. Accessed:2017-12-15.

[15] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane, *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[16] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[17] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," in *ACM SIGGRAPH 2008 classes*. ACM, 2008, p. 16.

[18] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU computing gems Jade edition*, vol. 2, pp. 359–371, 2011.

[19] J. Szuppe, "Boost.compute: A parallel computing library for c++ based on opencl," in *Proceedings of the 4th International Workshop on OpenCL*. ACM, 2016, p. 15.

[20] B. McDonnel and N. Elmqvist, "Towards utilizing gpus in information visualization: A model and implementation of image-space operations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1105–1112, 2009.

[21] D. Ren, B. Lee, and T. Höllerer, "Stardust: Accessible and transparent gpu support for information visualization rendering," in *Computer Graphics Forum*, vol. 36, no. 3. Wiley Online Library, 2017, pp. 179–188.

[22] C. Ahlberg and B. Shneiderman, "Visual information seeking: Tight coupling of dynamic query filters with starfield displays," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1994, pp. 313–317.

[23] D. Fisher, I. Popov, and S. Drucker, "Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster," *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208294

[24] C. Stolte, D. Tang, and P. Hanrahan, "Multiscale visualization using data cubes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 2, pp. 176–187, 2003.

[25] L. Lins, J. T. Klosowski, and C. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 19, no. 12, pp. 2456–2465, 2013.

[26] TIBCO Spotfire. Data Visualization and Analytics Software. http://spotfire.tibco.com/. Accessed:2017-12-15.

[27] Tableau Software. Business Intelligence and Analytics. http://www.tableau.com/. Accessed:2017-12-15.

[28] H. Piringer, C. Tominski, P. Muigg, and W. Berger, "A multi-threading architecture to support interactive visual exploration," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1113–1120, 2009.

[29] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer, "Profiler: Integrated statistical analysis and visualization for data quality assessment," in *Proceedings of the International Working Conference on Advanced Visual Interfaces*. ACM, 2012, pp. 547–554.

[30] H.-J. Schulz, M. Angelini, G. Santucci, and H. Schumann, "An enhanced visualization process model for incremental visualization," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 7, pp. 1830–1842, 2016.

[31] C. Hurter, "Image-based visualization: interactive multidimensional data exploration," *Synthesis Lectures on Visualization*, vol. 3, no. 2, pp. 1–127, 2015.

[32] J.-D. Fekete and C. Plaisant, "Interactive information visualization of a million items," in *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*. IEEE, 2002, pp. 117–124.

[33] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 215–226.

[34] C. Hurter, B. Tissoires, and S. Conversy, "Fromdady: Spreading aircraft trajectories across views to support iterative queries," *IEEE transactions on visualization and computer graphics*, vol. 15, no. 6, pp. 1017–1024, 2009.

[35] Z. Liu, B. Jiang, and J. Heer, "immens: Real-time visual querying of big data," in *Computer Graphics Forum*, vol. 32, no. 3pt4. Wiley Online Library, 2013, pp. 421–430.

[36] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[37] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM, 2011, pp. 3–14.

[38] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Acm Sigplan Notices*, vol. 46, no. 2. ACM, 2010, pp. 127–136.

[39] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "Sejits: Getting productivity and performance with selective embedded jit specialization," *Programming Models for Emerging Architectures*, vol. 1, no. 1, pp. 1–9, 2009.

[40] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "CUDPP: CUDA data parallel primitives library," 2007.

[41] E. H.-H. Chi, "A taxonomy of visualization techniques using the data state reference model," in *Information Visualization, 2000. InfoVis 2000. IEEE Symposium on*. IEEE, 2000, pp. 69–75.

[42] D. Florescu and G. Fourny, "Jsoniq: The history of a query language," *IEEE internet computing*, vol. 17, no. 5, pp. 86–90, 2013.

[43] K. Kaur and R. Rani, "Modeling and querying data in nosql databases," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–7.

[44] M. Stone, *A field guide to digital color*. CRC Press, 2016.

[45] M. Stone and L. Bartram, "Alpha, contrast and the perception of visual metadata," in *Color and Imaging Conference*, vol. 2008, no. 1. Society for Imaging Science and Technology, 2008, pp. 355–359.

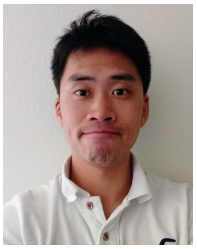
[46] J. Heer and B. Shneiderman, "Interactive dynamics for visual analysis," *Queue*, vol. 10, no. 2, p. 30, 2012.

[47] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 77–88.

[48] J. K. Li, M. Mubarak, R. B. Ross, C. D. Carothers, and K.-L. Ma, "Visual analytics techniques for exploring the design space of large-scale high-radix networks," in *IEEE International Conference on Cluster Computing*, 2017, pp. 193–203.

[49] C. Ross, C. D. Carothers, M. Mubarak, P. Carns, R. Ross, J. K. Li, and K.-L. Ma, "Visual data-analytics of large-scale parallel discrete-event simulations," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on*. IEEE, 2016, pp. 87–97.

[50] W3C. File API. https://www.w3.org/TR/file-upload/. Accessed:2017-12-15.



Jianping Kelvin Li is a graduate student in the VIDI lab at the University of California, Davis, studying computer science. His research interests include data management, visual analytics, and high-performance computing. The mission of his research is to help more people understand complex information and discover new knowledge by using data analytics and visualization technologies. Jianping received a BS degree in electrical and computer engineering from the University of California, Davis.

Kwan-Liu Ma Kwan-Liu Ma is a professor of computer science, University of California, Davis, where he leads the VIDI Lab and the UC Davis Center for Visualization. He received his PhD degree in computer science from the University of Utah in 1993. His research interests include visualization, computer graphics, high-performance computing, and user interface design. Professor Ma received the 2013 IEEE VGTC Visualization Technical Achievement Award for his significant research contributions to the field of visualization. He is presently the AEIC of IEEE CG&A. Professor Ma is a fellow of the IEEE.