Performance Optimization

General Performance Expectations

There are mainly two aspects that developers usually consider regarding the performance of any computer programs: the time and the memory consumption, both of which obviously depends on the specs of the hardware deck.gl is ultimately running on.

On 2015 MacBook Pros with dual graphics cards, most basic layers (like `ScatterplotLayer`) renders fluidly at 60 FPS during pan and zoom operations up to about 1M (one million) data items, with framerates dropping into low double digits (10-20FPS) when the data sets approach 10M items.

Even if interactivity is not an issue, browser limitations on how big chunks of contiguous memory can be allocated (e.g. Chrome caps individual allocations at 1GB) will cause most layers to crash during WebGL buffer generation somewhere between 10M and 100M items. You would need to break up your data into chunks and use multiple deck.gl layers to get past this limit.

Modern phones (recent iPhones and higher-end Android phones) are surprisingly capable in terms of rendering performance, but are considerably more sensitive to memory pressure than laptops, resulting in browser restarts or page reloads. They also tend to load data significantly slower than desktop computers, so some tuning is usually needed to ensure a good overall user experience on mobile.

Layer Update Performance

Layer update happens when the layer is first created, or when some layer props change. During an update, deck.gl may load necessary resources (e.g. image textures), generate WebGL buffers, and upload them to the GPU, all of which may take some time to complete, depending on the number of items in your `data` prop. Therefore, the key to performant deck.gl applications is to minimize layer updates wherever possible.

Minimize data changes

When the `data` prop changes, the layer will recalculate all of its WebGL buffers. The time required for this is proportional to the number of items in your `data` prop. This step is the most expensive operation that a layer does - also on CPU - potentially affecting the responsiveness of the application. It may take multiple seconds for multi-million item layers, and if your `data` prop is updated frequently (e.g. animations), "stutter" can be visible even for layers with just a few thousand items.

Some good places to check for performance improvements are:

Avoid unnecessary shallow change in data prop

The layer does a shallow comparison between renders to determine if it needs to regenerate buffers. If nothing has changed, make sure you supply the *same* data object every time you render. If the data object has to change shallowly for some reason, consider using the `dataComparator` prop to supply a custom comparison logic.

```
// Bad
const DATA = [...];
const filters = {minTime: -1, maxTime: Infinity};

function setFilters(minTime, maxTime) {
  filters.minTime = minTime;
  filters.maxTime = maxTime;
  render();
}

function render() {
  const layer = new ScatterplotLayer({
    // `filter` creates a new array every time `render` is called, even if the
filters have not changed
    data: DATA.filter(d => d.time >= filters.minTime && d.time <=
filters.maxTime),
    ...
  });

  deck.setProps({layers: [layer]});
}
```

```
// Good
const DATA = [...];
let filteredData = DATA;
const filters = {minTime: -1, maxTime: Infinity};
```

```
function setFilters(minTime, maxTime) {
  filters.minTime = minTime;
  filters.maxTime = maxTime;
  // filtering is performed only once when the filters change
  filteredData = DATA.filter(d => d.time >= minTime && d.time <= maxTime);
  render();
}

function render() {
  const layer = new ScatterplotLayer({
    data: filteredData,
    ...
  });

  deck.setProps({layers: [layer]});
}
```

Use updateTriggers

So `data` has indeed changed. Do we have an entirely new collection of objects? Or did just certain fields changed in each row? Remember that changing `data` will update *all* buffers, so if, for example, object positions have not changed, it will be a waste of time to recalculate them.

```
// Bad
const DATA = [...];
let currentYear = null;
let currentData = DATA;

function selectYear(year) {
  currentYear = year;
  currentData = DATA.map(d => ({
    position: d.position,
    population: d.populationsByYear[year]
  }));
  render();
}

function render() {
  const layer = new ScatterplotLayer({
    // `data` changes every time year changed, but positions don't need to
update
    data: currentData,
    getPosition: d => d.position,
    getRadius: d => Math.sqrt(d.population),
    ...
```

```
  });

  deck.setProps({layers: [layer]});
}
```

In this case, it is more efficient to use `updateTriggers` to invalidate only the selected attributes:

```
// Good
const DATA = [...];
let currentYear = null;

function selectYear(year) {
  currentYear = year;
  render();
}

function render() {
  const layer = new ScatterplotLayer({
    // `data` never changes
    data: DATA,
    getPosition: d => d.position,
    // radius depends on `currentYear`
    getRadius: d => Math.sqrt(d.populationsByYear[currentYear]),
    updateTriggers: {
      // This tells deck.gl to recalculate radius when `currentYear` changes
      getRadius: currentYear
    },
    ...
  });

  deck.setProps({layers: [layer]});
}
```

Handle incremental data loading

A common technique for handling big datasets on the client side is to load data in chunks. We want to update the visualization whenever a new chunk comes in. If we append the new chunk to an existing data array, deck.gl will recalculate the whole buffers, even for the previously loaded chunks where nothing have changed:

```
// Bad
let loadedData = [];
```

```
function onNewDataArrive(chunk) {
  loadedData = loadedData.concat(chunk);
  render();
}

function render() {
  const layer = new ScatterplotLayer({
    // If we have 1 million rows loaded and 100,000 new rows arrive,
    // we end up recalculating the buffers for all 1,100,000 rows
    data: loadedData,
    ...
  });

  deck.setProps({layers: [layer]});
}
```

To avoid doing this, we instead generate one layer for each chunk:

```
// Good
const dataChunks = [];

function onNewDataArrive(chunk) {
  dataChunks.push(chunk);
  render();
}

function render() {
  const layers = dataChunks.map((chunk, chunkIndex) => new ScatterplotLayer({
    // Important: each layer must have a consistent & unique id
    id: `chunk-${chunkIndex}`,
    // If we have 10 100,000-row chunks already loaded and a new one arrive,
    // the first 10 layers will see no prop change
    // only the 11th layer's buffers need to be generated
    data: chunk,
    ...
  }));

  deck.setProps({layers});
}
```

Starting v7.2.0, support for async iterables is added to efficiently update layers with incrementally loaded data:

```
// Create an async iterable
async function* getData() {
```

```
  for (let i = 0; i < 10; i++) {
    await const chunk = fetchChunk(...);
    yield chunk;
  }
}

function render() {
  const layer = new ScatterplotLayer({
    // When a new chunk arrives, deck.gl only updates the sub buffers for the
new rows
    data: getData(),
    ...
  });

  deck.setProps({layers: [layer]});
}
```

See Layer properties for details.

Favor layer visibility over addition and removal

Removing a layer will lose all of its internal states, including generated buffers. If the layer is added back later, all the WebGL resources need to be regenerated again. In the use cases where layers need to be toggled frequently (e.g. via a control panel), there might be a significant perf penalty:

```
// Bad
const DATA = [...];
const layerVisibility = {circles: true, labels: true}

function toggleLayer(key) {
  layerVisibility[key] = !layerVisibility[key];
  render();
}

function render() {
  const layers = [
    // when visibility goes from on to off to on, this layer will be completely
removed and then regenerated
    layerVisibility.circles && new ScatterplotLayer({
      data: DATA,
      ...
    }),
    layerVisibility.labels && new TextLayer({
      data: DATA,
      ...
```

```
    })
  ];

  deck.setProps({layers});
}
```

The `visible` prop is a cheap way to temporarily disable a layer:

```
// Good
const DATA = [...];
const layerVisibility = {circles: true, labels: true}

function toggleLayer(key) {
  layerVisibility[key] = !layerVisibility[key];
  render();
}

function render() {
  const layers = [
    // when visibility is off, this layer's internal states will be retained in
memory, making turning it back on instant
    new ScatterplotLayer({
      data: DATA,
      visible: layerVisibility.circles,
      ...
    }),
    new TextLayer({
      data: DATA,
      visible: layerVisibility.labels,
      ...
    })
  ];

  deck.setProps({layers});
}
```

Optimize Accessors

99% of the CPU time that deck.gl spends in updating buffers is calling the accessors you supply to the layer. Since they are called on every data object, any performance issue in the accessors is amplified by the size of your data.

Favor constants over callback functions

Most accessors accept constant values as well as functions. Constant props are extremely cheap to update in comparison. Use `ScatterplotLayer` as an example, the following two prop settings yield exactly the same visual outcome:

- `getFillColor: [255, 0, 0, 128]` - deck.gl uploads 4 numbers to the GPU.
- `getFillColor: d => [255, 0, 0, 128]` - deck.gl first builds a typed array of `4 * data.length` elements, call the accessor `data.length` times to fill it, then upload it to the GPU.

  Aside from accessors, most layers also offer one or more `*Scale` props that are uniform multipliers on top of the per-object value. Always consider using them before invoking the accessors:

```
// Bad
const DATA = [...];

function animate() {
  render();
  requestAnimationFrame(animate);
}

function render() {
  const scale = Date.now() % 2000;

  const layer = new ScatterplotLayer({
    data: DATA,
    getRadius: object => object.size * scale,
    // deck.gl will call `getRadius` for ALL data objects every animation
frame, which will likely choke the app
    updateTriggers: {
      getRadius: scale
    },
    ...
  });

  deck.setProps({layers: [layer]});
}
```

```
// Good
const DATA = [...];

function animate() {
  render();
```

```
  requestAnimationFrame(animate);
}

function render() {
  const scale = Date.now() % 2000;

  const layer = new ScatterplotLayer({
    data: DATA,
    getRadius: object => object.size,
    // This has virtually no cost to update, easily getting 60fps animation
    radiusScale: scale,
    ...
  });

  deck.setProps({layers: [layer]});
}
```

Use trivial functions as accessors

Whenever possible, make the accessors trivial functions and utilize pre-defined and/or pre-computed data.

```
// Bad
const DATA = [...];

function render() {
  const layer = new ScatterplotLayer({
    data: DATA,
    getFillColor: object => {
      // This line creates a new values array from each object
      // which can incur significant cost in garbage collection
      const maxPopulation = Math.max.apply(null,
Object.values(object.populationsByYear));
      // This switch case creates a new color array for each object
      // which can also incur significant cost in garbage collection
      if (maxPopulation > 1000000) {
        return [255, 0, 0];
      } else if (maxPopulation > 100000) {
        return [0, 255, 0];
      } else {
        return [0, 0, 255];
      }
    },
    getRadius: object => {
      // This line duplicates what's done in `getFillColor` and doubles the cost
      const maxPopulation = Math.max.apply(null,
Object.values(object.populationsByYear));
```

```
      return Math.sqrt(maxPopulation);
    }
    ...
  });

  deck.setProps({layers: [layer]});
}
```

```
// Good
const DATA = [...];

// Use a for loop to avoid creating new objects
function getMaxPopulation(populationsByYear) {
  let maxPopulation = 0;
  for (const year in populationsByYear) {
    const population = populationsByYear[year];
    if (population > maxPopulation) {
      maxPopulation = population;
    }
  }
  return maxPopulation;
}

// Calculate max population once and store it in the data
DATA.forEach(d => {
  d.maxPopulation = getMaxPopulation(d.populationsByYear);
});

// Use constant color values to avoid generating new arrays
const COLORS = {
  ONE_MILLION: [255, 0, 0],
  HUNDRED_THOUSAND: [0, 255, 0],
  OTHER: [0, 0, 255]
};

function render() {
  const layer = new ScatterplotLayer({
    data: DATA,
    getFillColor: object => {
      if (object.maxPopulation > 1000000) {
        return COLORS.ONE_MILLION;
      } else if (maxPopulation > 100000) {
        return COLORS.HUNDRED_THOUSAND;
      } else {
        return COLORS.OTHER;
      }
    },
```

```
    getRadius: object => Math.sqrt(object.maxPopulation),
    ...
  });

  deck.setProps({layers: [layer]});
}
```

Use Binary Data

When creating data-intensive applications, it is often desirable to offload client-side data processing to the server or web workers.

The server can send data to the client more efficiently using binary formats, e.g. protobuf, Arrow or simply a custom binary blob.

Some deck.gl applications use web workers to load data and generate attributes to get the processing off the main thread. Modern worker implementations allow ownership of typed arrays to be transferred directly between threads at virtually no cost, bypassing serialization and deserialization of JSON objects.

Supply binary blobs to the data prop

Assume we have the data source encoded in the following format:

```
// lon1, lat1, radius1, red1, green1, blue1, lon2, lat2, ...
const binaryData = new Float32Array([-122.4, 37.78, 1000, 255, 200, 0, -122.41,
37.775, 500, 200, 0, 0, -122.39, 37.8, 500, 0, 40, 200]);
```

Upon receiving the typed arrays, the application can of course re-construct a classic JavaScript array:

```
// Bad
const data = [];
for (let i = 0; i < binaryData.length; i += 6) {
  data.push({
    position: binaryData.subarray(i, i + 2),
    radius: binaryData[i + 2],
    color: binaryData.subarray(i + 3, i + 6)
  });
}

new ScatterplotLayer({
```

```
  data,
  getPosition: d => d.position,
  getRadius: d => d.radius,
  getFillColor: d => d.color
});
```

However, in addition to requiring custom repacking code, this array will take valuable CPU time to create, and significantly more memory to store than its binary form. In performance-sensitive applications that constantly push a large volume of data (e.g. animations), this method will not be efficient enough.

Alternatively, one may supply a non-iterable object (not Array or TypedArray) to the `data` object. In this case, it must contain a `length` field that specifies the total number of objects. Since `data` is not iterable, each accessor will not receive a valid `object` argument, and therefore responsible of interpreting the input data's buffer layout:

```
// Good
// Note: binaryData.length does not equal the number of items,
// which is why we need to wrap it in an object that contains a custom `length`
field
const DATA = {src: binaryData, length: binaryData.length / 6}

new ScatterplotLayer({
  data: DATA,
  getPosition: (object, {index, data}) => {
    return data.src.subarray(index * 6, index * 6 + 2);
  },
  getRadius: (object, {index, data}) => {
    return data.src[index * 6 + 2];
  },
  getFillColor: (object, {index, data, target}) => {
    return data.src.subarray(index * 6 + 3, index * 6 + 6);
  }
})
```

Optionally, the accessors can utilize the pre-allocated `target` array in the second argument to further avoid creating new objects:

```
// Good
const DATA = {src: binaryData, length: binaryData.length / 6}

new ScatterplotLayer({
  data: DATA,
```

```
  getPosition: (object, {index, data, target}) => {
    target[0] = data.src[index * 6];
    target[1] = data.src[index * 6 + 1];
    target[2] = 0;
    return target;
  },
  getRadius: (object, {index, data}) => {
    return data.src[index * 6 + 2];
  },
  getFillColor: (object, {index, data, target}) => {
    target[0] = data.src[index * 6 + 3];
    target[1] = data.src[index * 6 + 4];
    target[2] = data.src[index * 6 + 5];
    target[3] = 255;
    return target;
  }
})
```

Supply attributes directly

While the built-in attribute generation functionality is a major part of a `Layer`s functionality, it can become a major bottleneck in performance since it is done on CPU in the main thread. If the application needs to push many data changes frequently, for example to render animations, data updates can block rendering and user interaction. In this case, the application should consider precalculated attributes on the back end or in web workers.

Deck.gl layers accepts external attributes as either a typed array or a WebGL buffer. Such attributes, if prepared carefully, can be directly utilized by the GPU, thus bypassing the CPU-bound attribute generation completely.

This technique offers the maximum performance possible in terms of data throughput, and is commonly used in heavy-duty, performance-sensitive applications.

To generate an attribute buffer for a layer, take the results returned from each object by the `get*` accessors and flatten them into a typed array. For example, consider the following layers:

Should we move the attribute generation to a web worker:

```
// Worker
// positions can be sent as either float32 or float64, depending on precision
requirements
// point[0].x, point[0].y, point[0].z, point[1].x, point[1].y, point[1].z, ...
const positions = new Float64Array(POINT_CLOUD_DATA.flatMap(d => d.position));
```

```
// point[0].r, point[0].g, point[0].b, point[1].r, point[1].g, point[1].b, ...
const colors = new Uint8Array(POINT_CLOUD_DATA.flatMap(d => d.color));

// send back to main thread
postMessage({pointCount: POINT_CLOUD_DATA.length, positions, colors},
[positions.buffer, colors.buffer]);
```

```
  getNormal: [0, 0, 1]
```

```
// Main thread
// `data` is received from the worker
new PointCloudLayer({
  data: {
    // this is required so that the layer knows how many points to draw
    length: data.pointCount,
    attributes: {
      getPosition: {value: data.positions, size: 3},
      getColor: {value: data.colors, size: 3},
    }
  },
  // constant accessor works without raw data
  getNormal: [0, 0, 1]
});
```

Note that instead of `getPosition`, we supply a `data.attributes.getPosition` object. This object defines the buffer from which `PointCloudLayer` should access its positions data. See the base `Layer` class' data prop for details.

It is also possible to use interleaved or custom layout external buffers:

```
// Worker
// point[0].x, point[0].y, point[0].z, point[0].r, point[0].g, point[0].b,
point[1].x, point[1].y, point[1].z, point[1].r, point[1].g, point[1].b, ...
const positionsAndColors = new Float32Array(POINT_CLOUD_DATA.flatMap(d => [
  d.position[0],
  d.position[1],
  d.position[2],
  // colors must be normalized if sent as floats
  d.color[0] / 255,
  d.color[1] / 255,
  d.color[2] / 255
]));

// send back to main thread
postMessage({pointCount: POINT_CLOUD_DATA.length, positionsAndColors},
[positionsAndColors.buffer]);
```

```
import {Buffer} from '@luma.gl/core';
const buffer = new Buffer(gl, {data: data.positionsAndColors});

new PointCloudLayer({
  data: {
    length : data.pointCount,
    attributes: {
      getPosition: {buffer, size: 3, offset: 0, stride: 24},
      getColor: {buffer, size: 3, offset: 12, stride: 24},
    }
  },
  // constant accessor works without raw data
  getNormal: [0, 0, 1]
});
```

See full example in examples/experimental/interleaved-buffer.

Note that external attributes only work with primitive layers, not composite layers, because composite layers often need to preprocess the data before passing it to the sub layers. Some layers that deal with variable-width data, such as `PathLayer`, `SolidPolygonLayer`, require additional information passed along with `data.attributes`. Consult each layer's documentation before use.

Layer Rendering Performance

Layer rendering time (for large data sets) is essentially proportional to:

1. The number of vertex shader invocations, which corresponds to the number of items in the layer's `data` prop
2. The number of fragment shader invocations, which corresponds to the total number of pixels drawn.

Thus it is possible to render a scatterplot layer with 10M items with reasonable frame rates on recent GPUs, provided that the radius (number of pixels) of each point is small.

It is good to be aware that excessive overdraw (drawing many objects/pixels on top of each other) can generate very high fragment counts and thus hurt performance. As an example, a `Scatterplot` radius of 5 pixels generates ~ 100 pixels per point. If you have a `Scatterplot` layer with 10 million points, this can result in up to 1 billion fragment shader invocations per frame. While dependent on zoom levels (clipping will improve performance to some extent)

this many fragments will certainly strain even a recent MacBook Pro GPU.

Layer Picking Performance

deck.gl performs picking by drawing the layer into an off screen picking buffer. This essentially means that every layer that supports picking will be drawn off screen when panning and hovering. The picking is performed using the same GPU code that does the visual rendering, so the performance should be easy to predict.

Picking limitations:

- The picking system can only distinguish between 16M items per layer.
- The picking system can only handle 256 layers with the pickable flag set to true.

Number of Layers

The layer count of an advanced deck.gl application tends to gradually increase, especially when using composite layers. We have built and optimized a highly complex application using close to 100 deck.gl layers (this includes hierarchies of sublayers rendered by custom composite layers rendering other composite layers) without seeing any performance issues related to the number of layers. If you really need to, it is probably possible to go a little higher (a few hundred layers). Just keep in mind that deck.gl was not designed to be used with thousands of layers.

Common Issues

A couple of particular things to watch out for that tend to have a big impact on performance:

- If not needed disable Retina/High DPI rendering. It generates 4x the number of pixels (fragments) and can have a big performance impact that depends on which computer or monitor is being used. This feature can be controlled using `useDevicePixels` prop of `DeckGL` component and it is on by default.
- Avoid using luma.gl debug mode in production. It queries the GPU error status after each operation which has a big impact on performance.

Smaller considerations:

- Enabling picking can have a small performance penalty so make sure the `pickable` property is `false` in layers that do not need picking (this is the default value).

Edit this page