Eurographics Symposium on Parallel Graphics and Visualization (2022)
R. Bujack, J. Tierny, F. Sadlo (Editors)

GraphWaGu: GPU Powered Large Scale Graph Layout Computation and Rendering for the Web

Landon Dyken[1], Pravin Poudel[2], Will Usher[3], Steve Petruzza[2], Jake Y. Chen[1], Sidharth Kumar[1]

[1]University of Alabama at Birmingham
[2]Utah State University
[3]Intel

Abstract

Large scale graphs are used to encode data from a variety of application domains such as social networks, the web, biological networks, road maps, and finance. Computing enriching layouts and interactive rendering play an important role in many of these applications. However, producing an efficient and interactive visualization of large graphs remains a major challenge, particularly in the web-browser. Existing state of the art web-based visualization systems such as D3.js, Stardust, and NetV.js struggle to achieve interactive layout and visualization for large scale graphs. In this work, we leverage the latest WebGPU technology to develop GraphWaGu*, the first WebGPU-based graph visualization system. WebGPU is a new graphics API that brings the full capabilities of modern GPUs to the web browser. Leveraging the computational capabilities of the GPU using this technology enables* GraphWaGu *to scale to larger graphs than existing technologies.* GraphWaGu *embodies both fast parallel rendering and layout creation using modified Frutcherman-Reingold and Barnes-Hut algorithms implemented in WebGPU compute shaders. Experimental results demonstrate that our solution achieves the best performance, scalability, and layout quality when compared to current state of the art web-based graph visualization libraries. All of our source code for the project is available at https://github.com/harp-lab/GraphWaGu.*

1. Introduction

A graph is used to represent connected entities and the relationships between them. Graphs are ubiquitous, appearing in many application domains, such as social networks, the web, the semantic web, road maps, communication networks, biology, and finance. It is common to process and analyze graphs for the purpose of extracting features such as connected components, cliques/triangles, and shortest paths [JDG*20, HJC*19]. There are many production level graph mining [CDGP20, KG19, TFS*15, AN16] and analytic [DGH*19, NLP13] systems to perform these kinds of tasks. Visualizing graphs [HMM00] where nodes and edges are mapped and projected to the 2D plane is another crucial task. There are many applications of graph visualization, including community detection [CBP14], cluster analysis [AVHK06, EF96], and summarizing and understanding [KKVF15] the overall structure of connected data. There are two aspects associated with visualizing a graph in 2D, *layout creation* (also referred to as graph drawing), where Cartesian coordinates are computed for every node of the graph, and *rendering*, where nodes and edges are rendered to an image based on the computed layout. In this paper, we limit the discussion of layout creation and rendering to undirected graphs.

With the advent of big data and large scale applications leveraging HPC resources, we are observing graphs of increasing sizes [SMS*17], with millions of vertices and edges. Interactively visualizing such large scale graphs is challenging, as the massive number of nodes and edges poses a severe computational and rendering challenge. This problem is further exacerbated for web-based visualization systems. The web browser has become the preferred modality for data visualization, as it provides a standardized cross-platform environment through which applications can be deployed to users. Existing state of the art web based graph visualization libraries such as D3.js [BOH11], Cytoscape.js [FLH*16], and Stardust [RLH17] struggle to achieve interactive rendering and layout creation for such large scale graphs.

To deal with visualizing large graphs, existing state of the art graph mining and analytic systems perform concurrent parallel execution, leveraging either shared memory parallelism on GPUs or distributed parallelism through MPI. Existing web-based graph visualization libraries, however, have either been limited to serial execution on the CPU, as in D3.js, or have leveraged GPU capabilities for rendering only, as in Stardust.js. These limitations have caused existing web-based graph visualization systems to struggle with scaling to large graphs while maintaining interactivity. WebGPU [Web] is a new technology that enables addressing this limitation in the web browser. Currently in development for all major browsers, WebGPU is a low-level graphics API similar to DirectX 12, Vulkan, and Metal, that brings the advanced rendering and computational capabilities of modern GPUs to web browser

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

DOI: 10.2312/pgv.20221067

applications. Compared to WebGL [Gro22], the current standard for GPU rendering on the web which is used by Stardust [RLH17] and NetV.js [HPZC21] to accelerate rendering, WebGPU provides a significant increase in GPU capabilities. Specifically, WebGPU provides compute shaders and storage buffers, which are essential for implementing computational algorithms on the GPU.

We have developed a web-based graph visualization library, *GraphWaGu*, that uses WebGPU to leverage the GPU's computational power for layout creation and rendering for undirected graphs in 2 dimensions. By doing so, *GraphWaGu* is capable of visualizing large-scale networks at interactive frame rates. In particular, we make the following specific contributions:

- The first parallel and scalable WebGPU based graph rendering system, capable of rendering up to 100,000 nodes and 2,000,000 edges with interactive frame rates ($\geq$ 10 FPS)
- The first parallel WebGPU based implementation of force directed layout computation using the Frutcherman-Reingold algorithm.
- An optimized parallel implementation for force directed layout computation using quadtree generation and traversal for Barnes-Hut simulation in WebGPU compute shaders (without recursion or use of pointers).
- Experimental studies to compare the performance and scalability of our WebGPU solutions against state of the art web-based graph visualization libraries. For 100,000 nodes and 2,000,000 edges, we maintain rendering frame rates 4$\times$ higher than the next best library (NetV.js) and layout creation times up to half that of D3.js.

2. Background and Related Work

In this section we cover relevant background work for both phases of graph visualization: graph layout creation and rendering. We then briefly discuss the key features of WebGPU that we leverage in *GraphWaGu*.

2.1. Layout Creation

A graph G is comprised of a vertex (or node) set V and an edge (or link) set E, with cardinalities $|V|$ and $|E|$ respectively. When rendering a graph, every vertex is projected on a 2D plane, where the position of a vertex v is denoted by P_v. In this paper, we choose to restrict our discussion to only undirected graphs. We define two nodes u and v in G to be adjacent to each other if and only if $e_{u,v} \in E$.

Pioneering work by Eades [Ead84] proposed the spring embedder algorithm (SE) to generate aesthetically pleasing layouts by treating undirected graphs as mechanical systems of steel balls (nodes) and springs (edges). The spring force acting between nodes attracts or repels them from each other depending on the distance between them, bringing change to the energy of system. The nodes are allowed to move from their initial placement due to these spring forces until a global minimum energy state is attained. There have been multiple further works such as the KK method [KK*89] and Frutherman and Reingold (FR) force directed algorithm [FR91] to generate aesthetic graph layouts. The FR algorithm introduces a spring-electric model as a modification to the SE algorithm that imitates a physics simulation where vertices map to electrical particles and edges correspond to springs following Hook's law. In this method, attractive forces are computed between adjacent nodes that pull them together, and repulsive forces are computed between every pair of non-adjacent nodes that repel them from each other. The algorithm defines an ideal length l where these forces will cancel out, distance d_{uv} as euclidean distance between u and v, and computes attractive forces (f_a) and repulsive forces (f_r) as follows:

$$f_a(d) = \frac{{d_{uv}}^2}{l} \quad where\ u \neq v \quad and \quad e_{u,v} \notin E$$

$$f_r(d) = \frac{-l^2}{d_{uv}} \quad where\ e_{u,v} \in E$$

When the algorithm reaches a state of equilibrium or minimum energy, the layout computed tends to have edges of uniform length l and satisfying distance between separate connected components.

The force directed FR algorithm is an iterative process; every iteration moves vertices by computing attractive and repulsive forces until a suitable layout is obtained. In each iteration, the cost to compute repulsive forces between every pair of nodes is $O(|V|^2)$ and the cost to compute the attractive forces of every edge is $O(|E|)$. Due to the $O(|V|^2)$ repulsive force calculation, this method is computationally expensive for graphs with large numbers of nodes and non-interactive at scale. A number of works have been proposed to address this issue, such as a grid-variant algorithm [FR91], Fast Multipole Method [HK00], Barnes-Hut (BH) approximation [BH86], Well-Separated Pair Decomposition [LWZ15], Random Vertex Sampling [Gov19], and combinations of these [Gov19]. These can reduce the repulsive force computation to $O(|V|log|V|)$, or even $O(|V|)$ for Random Vertex Sampling. BH is one of the most popular techniques among these techniques due to its simplicity. BH utilizes a quadtree data structure to approximate forces between nodes when they are distant from each other, allowing the computation of repulsive forces with $O(|V|log|V|)$ average cost.

Prior work has also explored parallelizing graph layout creation. Brinkmann et al. [BRT17] parallelized the ForceAtlas2 [JVHB14] algorithm, while Grama et al. [GKS94] and Burtscher and Pingali [BP11] dealt with parallelizing the quadtree construction and traversal steps necessary for BH. There have also been efforts to improve performance by CPU parallelization, with Tikhonova and Ma [TM08] using a pre-processing step and grouping of vertices. These works successfully improved the speed and scalability of force directed layout creation, but are limited to GPU programming frameworks like CUDA and OpenCL. We propose a unique parallel implementation of layout creation in the web written completely in the WebGPU shading language (WGSL) [wgs].

2.2. Graph Rendering

Several visualization frameworks provide APIs to render graphs in the web, with some of the most popular being D3.js [BOH11], Cytoscape.js [FLH*16], and Stardust [RLH17]. Most of these visualization frameworks use Canvas API [con22a] or SVG [con22b] to render, which are poor in terms of scalability and performance.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

The Canvas API performs better than SVG, but the ability of both to handle large-scale data with real time interaction is limited compared to GPU-accelerated technologies. This can be observed in Figure 3. The WebGL API is a more scalable and powerful alternative to Canvas API and SVG but requires users wanting visualizations to create their own rendering pipeline with shader code. In recent years though, new WebGL-based tools provide the necessary abstractions to perform easy rendering of large-scale graphs. Examples of such libraries are Stardust, Sigma.js [Coe18], and NetV.js [HPZC21]. Among these, Stardust presents itself as a complement to D3.js whereas Sigma.js uses a separate library, graphology [Pli21], to manage its graph model. NetV.js [HPZC21] also uses WebGL and promises higher scalability thanks to its concise programming interfaces which allow an efficient way of storing and manipulating graph data. Although WebGL based libraries like NetV.js promise good rendering ability, their overall performance is still limited by the limitations of WebGL, and they must rely on serial graph layout libraries like D3.js for graph generation.

2.3. WebGPU

WebGPU is a new graphics API that brings the full capabilities of modern GPUs to the web browser. WebGPU is built from the ground up, complete with its own shading language WGSL. In contrast to WebGL, WebGPU is not a port of an existing native API, though it is designed to easily map to modern APIs like DirectX12, Vulkan, and Metal for performance, and draws inspiration from them in its design.

WebGPU provides a number of advantages over WebGL for developing complex compute and rendering applications in the browser. WebGPU enables rendering applications to construct a description of the rendering or compute pipeline state ahead of time, specifying the shaders, input and output data locations, and data layouts, to build a fixed description of the pipeline. Data are fed to the pipeline through bind groups, whose layouts similarly encode ahead of time the structure of the data to be provided to the pipeline, while allowing the buffers being read or written to be changed efficiently at execution time. The pre-configured state stored in the pipeline significantly reduces state configuration overhead costs incurred during execution, while still allowing flexibility as to what data buffers are read or written when the pipeline is executed. Furthermore, WebGPU supports compute shaders and storage buffers, providing unique support for developing GPU algorithms capable of processing large data sets in the browser. Prior work on deep neural networks [HKUH17] and scientific visualization [UP20] has leveraged the capabilities of WebGPU to deploy large-scale GPU parallel compute applications in the browser.

3. Implementation

We implement rendering and force-directed layout techniques in our tool, *GraphWaGu*, a web-based GPU-accelerated library for interacting with large-scale graphs. *GraphWaGu* utilizes WebGPU to create visualizations from input graphs onto mouse-interactive HTML Canvas5 elements. *GraphWaGu* also presents GPU implementations of Fruchterman-Reingold and Barnes-Hut approximate force-directed layout algorithms using WebGPU compute shaders. Together, these features establish a user-friendly prototype for computing, evaluating, and recomputing graph layouts on the web. We structure this section by first discussing the graph rendering in order to introduce how *GraphWaGu* handles redrawing the changing of node positions that comes with interactivity and force directed layout iteration, then propose our algorithms for computing layouts with and without BH approximation.

3.1. Graph Rendering

The *GraphWaGu* approach to graph rendering is built to take advantage of the unique features of WebGPU for web-based graphics using bind groups and large storage buffers. At system initialization, we create rendering pipelines for nodes and edges, with bind groups containing edge and node storage buffers, and then start an animation frame to continuously run these pipelines' render passes. The edge buffer consists of $2*|E|$ uint32s corresponding to the indices of each edge's source and target nodes, and the node buffer is filled with $2*|V|$ float32 positions. For view changes such as panning and zooming, events are captured by an HTML canvas controller and new view parameters are written to a uniform buffer, while changes to the node and edge data buffers can be written by WebGPU API calls in the CPU or by shaders running on the GPU.

For both pipelines, vertex buffers of one element are used to describe the type of primitive to be drawn; for edges a line and for nodes two triangles in the form of a square. A draw call is made to instance $|V|$ nodes and $|E|$ edges, and in each vertex shader, the instances access their corresponding element in the storage buffers and return its position to the rest of the pipeline. For edges, a color is simply returned in the fragment shader. For nodes, a check around the radius is made and the fragment alpha is computed as 1 minus the sigmoid of the distance of the fragment to the node center to simulate anti-aliasing.

There are two main advantages to this approach. First, the position of edges does not have to be pre-computed in order to create and fill a vertex buffer. When node positions change, the edge vertex shader will freely output the correct position by checking against the node storage buffer. Second, the storage buffers can be used in other WebGPU pipelines, e.g. to write new node positions each iteration of *GraphWaGu* force directed compute shaders. This saves the time of copying data from the source of the graph layout computation to the rendering tool.

3.2. Graph Layout Computation

We present algorithms *GraphWaGu* FR and *GraphWaGu* BH for Frutcherman-Reingold and Barnes-Hut approximate force directed layouts respectively. Iterations of the algorithms are computed and applied to nodes using WebGPU compute passes. These compute passes are run parallel to the render passes of the *GraphWaGu* animation frame, so that as new node positions are computed and applied, a new graph is rendered.

3.2.1. WebGPU Frutcherman-Reingold

Before running the *GraphWaGu* FR algorithm, we initialize the compute pipelines needed: adjacency matrix creation, force calculation, and force application with their respective compute shaders

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

Algorithm 1 *GraphWaGu* FR

1: **Input:** $G(V,E), coolingFactor$
2: **for** e in E **do**
3: $adjacencyMatrix[e.source + e.target * |V|] = 1$
4: $adjacencyMatrix[e.target + e.source * |V|] = 1$
5: **end for**
6: **while** $coolingFactor \geq \varepsilon$ **do**
7: **for** $i \leftarrow 0$ to $|V|$ **do in parallel**
8: $f = 0$
9: $v = V[invocationId]$
10: **for** $i \leftarrow 0$ to $|V|$ **do**
11: **if** $adjacencyMatrix[invocationId + i * |V|]$ **then**
12: $f \leftarrow f + f_a(v, V[i])$
13: **else**
14: $f \leftarrow f + f_r(v, V[i])$
15: **end if**
16: **end for**
17: $F[invocationID] = \frac{f}{|f|} * min(coolingFactor, |f|)$
18: **end for**
19: **for** $i \leftarrow 0$ to $|V|$ **do in parallel**
20: $V[i].position \leftarrow V[i].position + F[i]$
21: **end for**
22: $coolingFactor \leftarrow coolingFactor * initialCoolingFactor$
23: **end while**

and necessary bind group layouts. An overview of the general algorithm used in WebGPU Frutcherman-Reingold is detailed in Algorithm 1. It follows closely to the original force directed algorithm proposed by Frutcherman and Reingold, apart from the use of an adjacency matrix and parallel computation of forces.

An adjacency matrix A for a graph G is defined as a $|V|$ by $|V|$ matrix where $A(i, j)$ is 0 when (i, j) is not in E and 1 when (i, j) is. This structure allows us to check for edges in the force calculation shader in $O(1)$ time to decide whether to compute attractive or repulsive forces for that node. If an adjacency matrix or similar data structure is not used, attractive forces must be computed separately from repulsive forces in $O(|E|)$ time. A concern when using an adjacency matrix is that it can require a large amount of memory, as its size grows exponentially with $|V|$.

To address this, we note that each element of the adjacency matrix requires only one bit to track whether two nodes are connected or not. We create the adjacency matrix with $|V| \times |V|$ bits and access individual entries through bitwise operations and shifts in the compute shader. A simplified depiction of this is in lines 2-5 of Algorithm 1

Force computation is shown in lines 7-18 of Algorithm 1, where either repulsive or attractive forces are calculated for each vertex to each other vertex using the adjacency matrix, and application is shown in lines 19-21. We parallelize the FR force directed algorithm by dispatching one GPU thread per node, effectively running a parallel for loop over the node buffer and computing or applying forces for each. This method runs into issues when $|V|$ is above 65,535, as that is the maximum supported dispatch size in WebGPU. To address this we create batches of nodes of 50,000, and run a separate compute pass for each batch. However, even with parallelizing the repulsive force computation this method struggles to scale to very high node counts as the exact force computation incurs $O(|V|^2)$ cost. This requires us to implement some method of approximating these forces, for which we use BH approximation to achieve $O(|V|log|V|)$.

Algorithm 2 *GraphWaGu* BH Algorithm

1: **Input:** G(V, E), coolingFactor
2: createSortedEdgeLists();
3: **while** $coolingFactor \geq \varepsilon$ **do**
4: createQuadTree();
5: **for** $v \leftarrow 0$ to $|V|$ **do in parallel**
6: **for** $e_{i,j}$ in E with i or $j = v$ **do**
7: $F[v] \leftarrow F[v] + f_a(V[i], V[j])$
8: **end for**
9: **end for**
10: computeRepulsiveForces();
11: **for** $i \leftarrow 0$ to $|V|$ **do in parallel**
12: $V[i].position \leftarrow V[i].position + F[i]$
13: $F[i] \leftarrow 0$
14: **end for**
15: $coolingFactor \leftarrow coolingFactor * initialCoolingFactor$
16: **end while**

3.2.2. WebGPU Barnes-Hut

The Barnes-Hut approximation algorithm [BH86] has multiple challenges to be effectively implemented in parallel systems. This task has been addressed in previous work by [BP11], where they showed an efficient method of optimizing this process in CUDA utilizing barriers and thread-voting functions. The Barnes-Hut algorithm involves building and traversing a quadtree data structure, two tasks typically implemented through recursion, with nodes having pointers to their children, and dynamic allocation storage in a heap to hold the tree. Recursion, pointers, and dynamic allocation at runtime are not supported in programming compute shaders in WebGPU. Methods for building quadtrees in parallel do exist such as through the use of linear quadtrees [Gar82], and are depicted in work such as [Kar12], but these are not adapted specifically for the task of layout creation and often rely on structures and GPU programming functions not currently available in the WebGPU API. We present our own methods of building and traversing a quadtree in compute passes on the GPU compatible with the WebGPU API. Our quadtree implementation utilizes one of the two approaches described by [Sam90] for pointerless representations, treating the quadtree as a collection of its leaf nodes, with directional codes for indexes of its northwest (NW), northeast (NE), southwest (SW), and southeast (SE) children. From the same source, we can also obtain the weak upper bound of $6 * |V|$ as an appropriate limit to the total number of items in our quadtree, including null pointers.

We safely initialize a WebGPU buffer of length $6 * |V|$ to contain all items within the quadtree, where each item is a struct with empty attributes for its boundary rectangle, center of mass, mass, and indices of NW, NE, SW, and SE children in the same buffer. This buffer is bound to a compute pipeline which runs at the beginning of each *GraphWaGu* BH layout iteration. When creating a quadtree, a bounding box must first be computed containing all of the nodes of

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

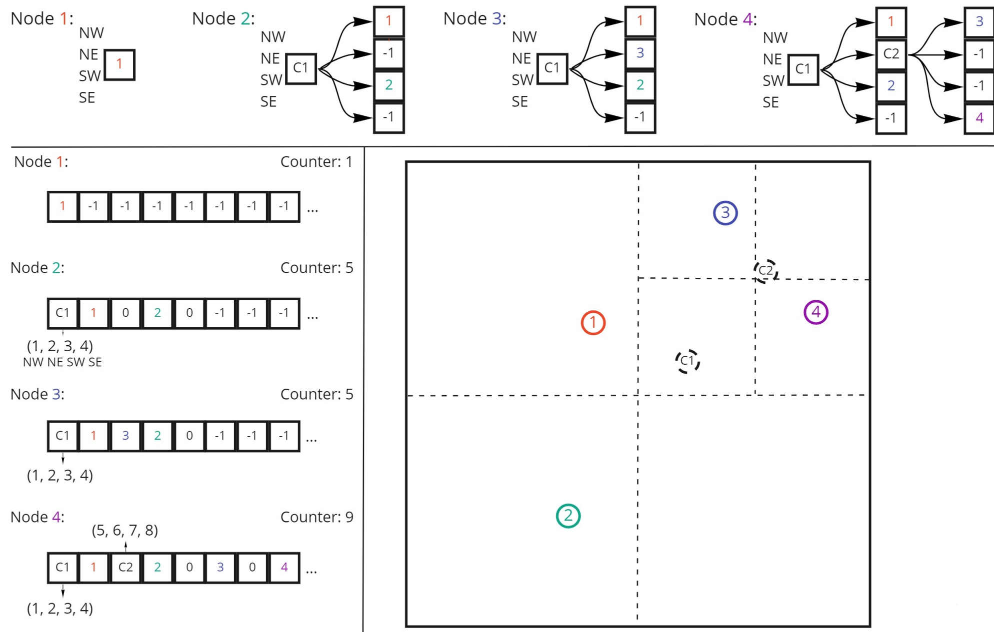

Figure 1: *Inserting a 4 node graph into an empty quadtree. At the top, a visualization of the classic quadtree creation using pointers and dynamic allocation is shown, from left to right. To the left of the example graph, the same sequence of insertions is shown for the* GraphWaGu *BH algorithm into our quadtree buffer, from top to bottom. In the first insertion, the quadtree struct at index 0 is set to the first node, and counter is made 1. At insertion 2, a partition is made, so the 0 spot in our buffer becomes cell C1 with center of mass between 1 and 2, its NW, NE, SW, and SE attributes are set, and counter increments by 4. Insertion 3 changes the center of mass of C1, and insertion 4 does the same and requires another partition to make C2. Ellipses at the end of each buffer indicate the array should be size* $6*|V| = 24$*, -1 refers to null items, and 0s are empty quadtree structs.*

the input graph. In order to simplify this step and maintain robustness for our float32 node positions, *GraphWaGu* BH forces all node positions within the range $[0,1]$ by clamping values when forces are applied to change node positions. Because duplicate node positions can lead to loops of partitioning when creating a quadtree, this clamp also applies a small position randomization on the boundaries. The first step of the creation shader then is to declare a root node with boundary rectangle $[0,0,1,1]$ and a counter variable to keep track of the latest insertion into the quadtree buffer. A for loop is then run to insert all nodes from the node data buffer. Each node insertion is accomplished iteratively, incrementing counter and setting the NW, NE, SW, and SE attributes to the indices of children when partitions are made. Center of mass and total mass are computed in the usual way for cells and leaf nodes and stored in the struct attributes of the correct index in the quadtree buffer. An example quadtree creation is given by Figure 1 showing this method.

Once the quadtree is created, attractive and repulsive forces are computed in their own compute pipelines before being applied in the same way as *GraphWaGu* FR. An overview of the full *GraphWaGu* BH algorithm is given in Algorithm 2. Attractive forces in this algorithm now have to be computed in $O(|E|)$ time, as an adjacency matrix is not built and repulsive and attractive forces must be computed in separate passes. To speed this up, the algorithm begins by creating sorted lists (by source index and target index) of the input graph's edges so that we can parallelize the attractive force computation. This allows us to compute attractive forces for each node in parallel, dispatching a thread for each node which then iterates through the list of edges it is a part of and accumulates attractive forces. This is seen in lines 5-9 in Algorithm 2. The same force buffer is then used for the repulsive force pass, taking in the result of the attractive forces and returning the total force results. In order to traverse the quadtree to compute Barnes-Hut approximate repulsive forces for each node, a large buffer is sent to be used as a pseudo-stack by the repulsive force compute shader described in line 10 of Algorithm 2. Because the average height of a quadtree is $O(log_4|V|)$, each traversal will add around this many items to the stack, so the size of the stack buffer is set to just above $|V|log_4|V|$. Two variables are used to keep track of the shader's current place in the stack and the last item it needs to process. An example of this method is shown in Figure 2. Introducing parallelism to the repulsive force pass in this algorithm is done in the same way as force computation in the *GraphWaGu* FR algorithm, by dispatching a thread per node, effectively running a parallel for loop over the nodes.

4. Evaluation

To evaluate the performance of *GraphWaGu*, we performed three different sets of experiments comparing our solution with state-of-the-art techniques. In particular, we report: (i) rendering benchmark using synthetic graphs at different scales; (ii) layout generation benchmark using synthetic graphs at different scales; (iii) layout creation and rendering of a variety of real graphs.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

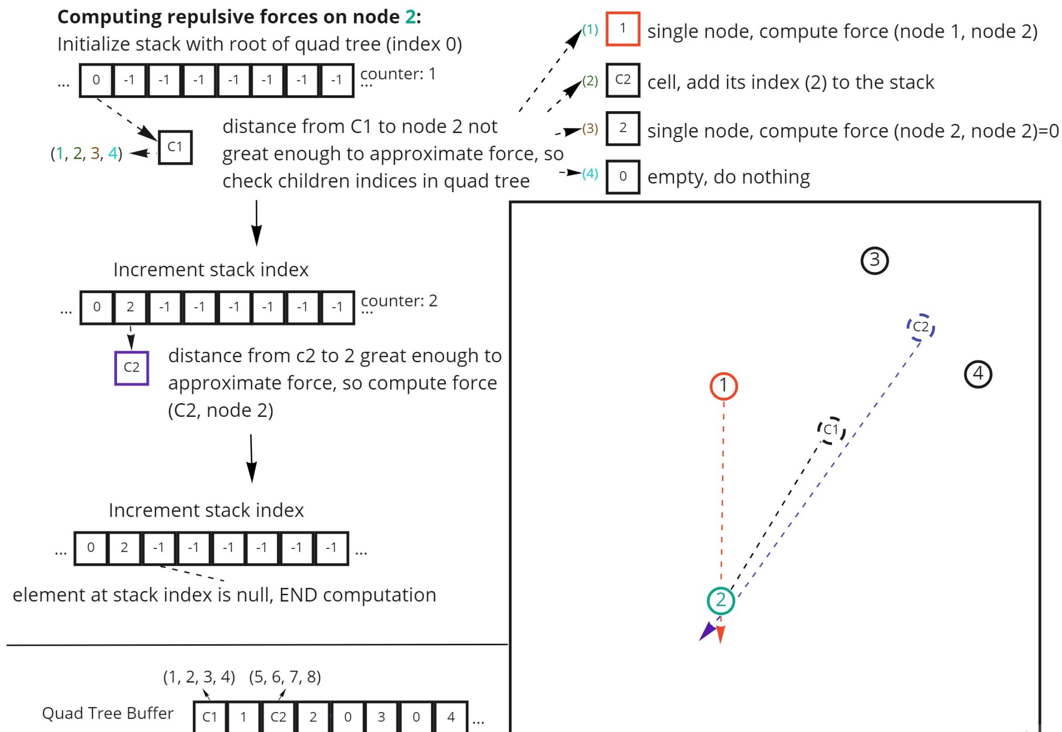

Figure 2: *Computing repulsive forces for node 2 in* GraphWaGu *BH algorithm. This uses the same graph and quadtree buffer as 1. First, the root item of the quadtree C1 is assigned to the start of the stack and is checked. This is a cell item containing 4 children, so each child is evaluated, adding the cell C2 to the stack. This cell is then checked, and the distance from node 2 is far enough away to approximate the repulsive force, so we do not check C2's children. We then reach the end of the stack buffer, so the computation is finished.*

4.1. Experimental setup

We ran all experiments on two different system setups with browser Google Chrome Canary: (i) equipped with AMD Ryzen 4600 CPU (3 GHZ, 6 cores), 8GB RAM memory and integrated AMD Radeon C7 GPU (512MB Video RAM); (ii) equipped with AMD Ryzen 5600 CPU (3.7 GHz, 6 cores), 16GB RAM memory and a dedicated NVIDIA GeForce RTX 2060 GPU. All the code for the application running the node rendering and layout creation pipelines is written in *React*, compiled with *babel* transcompiler, and bundled with webpack module bundler.

For both the rendering and layout computation benchmarks, we generated 11 graphs with varying vertex counts, all with completely random edge connectivity and initial node positions using the javascript *Math.random* function. For these artificial graphs, we maintained an edge to vertex count ratio ($|V|:|E|$) of 20 to model real-world data (similar to the experimental setup of [HPZC21]). The total vertex count for each graph varies from 10^2 to 10^6. To demonstrate that our performance results are not biased by the nature of the synthetic (randomly generated) graphs, we also performed an additional evaluation using datasets generated by a wide range of applications and hosted by the SuiteSparse Matrix Collection [KAB*19]. Formerly known as the University of Florida Sparse Matrix Collection, these diverse datasets are large and actively maintained resources that arise in real applications. From this collection, we use five graphs and report layout and performance results in Table 1.

4.2. Graph Rendering Benchmark

We compare the rendering performance of *GraphWaGu* with other popular web-based graph rendering frameworks: NetV.js, D3.js (Canvas and SVG), G6.js, Cytoscape, Sigma.js, and Stardust. Among these frameworks, D3.js is the most popular due to its robust catalogue of visualization options and built-in libraries for algorithms such as computing force directed graph layouts. However, due to its dependency on the DOM tree and SVG, D3.js cannot effectively handle a large number of graphical marks, especially when running animation. The frameworks NetV.js, Sigma.js, and Stardust are built on top of WebGL, and can therefore utilize the GPU for high performance rendering; however, they do not accelerate graph layout computation. G6.js does not leverage the GPU for rendering nor computation, and has limited scalability for large scale graph rendering as a result.

In order to properly stress test the abilities of the rendering libraries chosen, we one-by-one call each to render a completely different graph of the size being tested each frame. We randomly generate the graphs for each frame as detailed in the experimental setup, and do not include graph generation time in the time it takes to complete a frame. Frames per second (FPS) is calculated for each

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

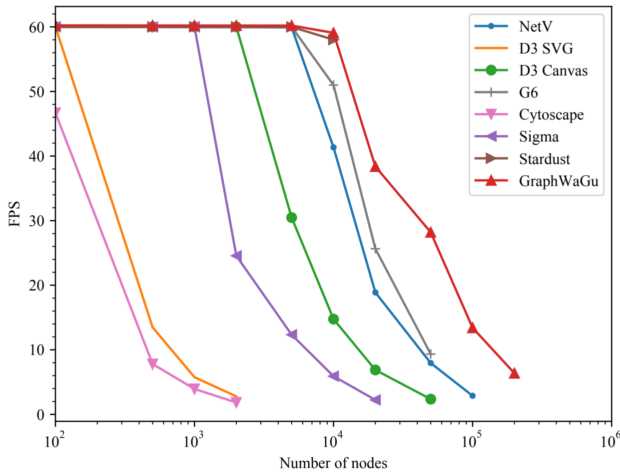

Figure 3: *Rendering performance on computer with high-end dedicated GPU (NVIDIA GeForce RTX 2060). Performance for all rendering libraries on dedicated GPU. Rendering performance of* GraphWaGu *becomes best among competitors after* 10,000 *nodes and is able to render the graph with* 50,000 *nodes and* 1,000,000 *edges in around 10fps. With ample GPU power, WebGL and WebGPU systems outcompete serial based libraries.*

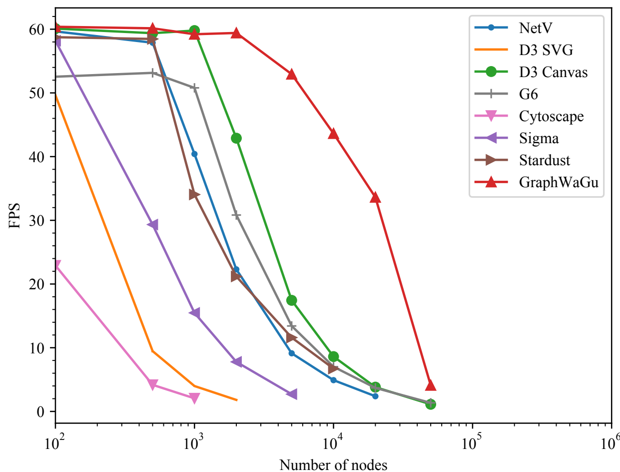

Figure 4: *Rendering performance on low-end computer with integrated GPU (AMD Radeon C7). The rendering performance of* GraphWaGu *becomes significantly better than other visualization frameworks after* 1,000 *nodes and* 20,000 *edges. Because of the lack of GPU power, D3 canvas is able to be competitive with GPU-accelerated libraries like NetV and Stardust.*

library after repeating this process for a total duration of five seconds for each graph size. The FPS is then averaged over this five seconds, and this is what we record, capped at 60 FPS since it is the maximum display frequency. This process of FPS calculation was done using the *stats* library [Cab]. We report these FPS results for all libraries on both the dedicated and integrated GPUs in Figures 3 and 4 respectively.

On both systems, *GraphWaGu* outperforms other frameworks and is the most scalable solution. The rendering performance of *GraphWaGu* is significantly better for the largest scale graphs even when using an integrated GPU (see Figure 4). On the integrated GPU, WebGL based libraries (NetV and Stardust) performed similarly to optimized serial libraries (D3 Canvas and G6) up to the highest number of nodes, as there was not much GPU power for them to take advantage of. On the dedicated GPU, we can see that WebGL based libraries perform much better, with Stardust maintaining about equal FPS to *GraphWaGu* until passing 10,000 nodes and 200,000 edges, which is the limit of Stardust graph rendering. For small sized graphs, all frameworks perform well, hence the FPS performance reaches the nominal limit of 60 FPS. For larger graphs, *GraphWaGu* shows better performance compared to other frameworks. The reason behind this are differences between the structure of the *GraphWaGu* system and the WebGL based libraries. These require a higher number of draw calls (which are costlier) than WebGPU to reflect the change of state in between the passing of frames. WebGPU allows resources and command bounding together in groups for dispatching them in chunk on GPU, which reduces the CPU overhead and improves performance.

We report that we are able to maintain interactive rendering ($\geq$ 10 FPS) until 100,000 nodes and 2,000,000 edges on the dedicated GPU system, while NetV.js is the only other library that maintains rendering, at 3 FPS. The rendering of *GraphWaGu* continues to be possible until 200,000 nodes, and 4,000,000 edges, a benchmark that is not possible on any other web-based graph visualization library. (see Figure 3).

4.3. Layout Computation Benchmarks

In this section, we evaluate the efficacy of *GraphWaGu* FR and *GraphWaGu* BH layout computation functionality. Because technologies like Stardust and NetV.js support only rendering while relying on other libraries to create graph layouts, we evaluate our system only against D3.js's D3-force layout computation ability. This library utilizes a Barnes-Hut approximation of repulsive forces, so that each iteration of its algorithm is $O(|V|log|V|)$, but it is only a serial implementation created in javascript. As with rendering, we use the synthetic graphs described in experimental setup to run benchmarks for our layout computations. We create one graph for each size chosen, and compute the layout determined by its random structure with *GraphWaGu* FR, *GraphWaGu* BH, and D3-force. For each generated graph, we measured the average iteration time of the force-directed layout algorithm after computing 87 iterations (i.e., enough iterations to reach equilibrium for the graphs being used). Because graph rendering and layout computation are coupled in the *GraphWaGu* system through the setup of WebGPU command encoding, each iteration time includes both layout computation and rendering of the current graph layout on that frame.

Figure 5 shows the results for the integrated GPU. We observe that *GraphWaGu* FR yields the best performance for graphs with relatively fewer number of nodes, however, D3-force outperforms *GraphWaGu* algorithms for graphs with larger number of nodes. This is because the low computational benefit offered by the low-end dedicated GPU does not offset the additional cost of creating and running the compute pipelines for *GraphWaGu* BH here; the

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

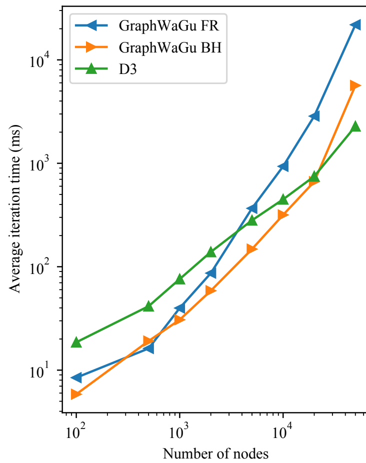

Figure 5: *Layout computation performance on integrated GPU. Layout computation is fastest in* GraphWaGu *algorithms until* 20,000 *nodes, when the integrated graphics card becomes unable to properly scale parallel computation of forces.*

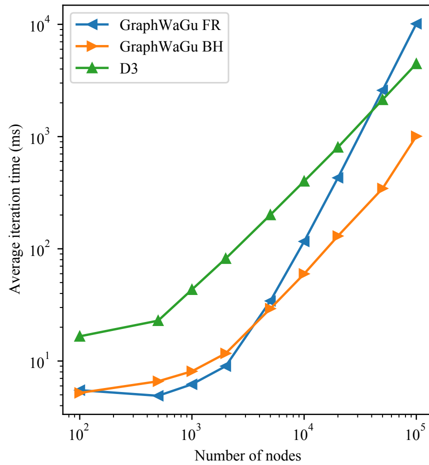

Figure 6: *Layout computation performance on dedicated GPU. Average iteration time is best in* GraphWaGu *FR until* 5,000 *nodes, when* GraphWaGu *BH becomes the best performing. Both* GraphWaGu *algorithms outperform D3.js except for huge number of nodes where* GraphWaGu *FR's* $O(|V|^2)$ *repulsive force calculation causes it to be the worst.*

GPU is likely unable to properly scale the parallel computation of forces for such large storage buffers being used.

Figure 6 shows the results for the dedicated GPU. For the largest graphs, *GraphWaGu* BH has the best iteration time, although it grows at the same rate as D3-force iteration time as both have $O(|V|log|V|)$ repulsive force calculation and $O(|E|)$ attractive force calculation cost. Iteration times are better for both *GraphWaGu* algorithms compared to D3-force when number of nodes is below 20,000, which is the point when the parallel $O(|V|^2)$ repulsive force calculation in *GraphWaGu* FR becomes worse than the serial $O(|V|log|V|)$ calculation done by D3-force. When the number of nodes is below 5,000 though, *GraphWaGu* FR is the best performing algorithm. This trend can be attributed to the overhead of constructing the quadtree, computation of the center of mass and centroid, and tree-traversal for force calculation associated with Barnes-Hut approximation used by both *GraphWaGu* BH and D3-force. The overhead becomes negligible for graphs with larger number of nodes, as the performance benefits of the quadtree greatly outweigh the overhead of its creation and traversal. These results follow closely to the trends for Fruchterman-Reingold and Barnes-Hut algorithms in parallel seen by [RSA20] for their BatchLayout and BatchLayout BH algorithms, although their performance is better with more optimized algorithms in native code, and they do not include rendering in their system. Overall, *GraphWaGu* BH is able to maintain interactive frame rate (around 100 millisecond iteration time) while computing layout and rendering until 20,000 nodes and 400,000 edges, while D3-force lasts until 2,000 nodes and 40,000 edges, and *GraphWaGu* FR lasts until 10,000 nodes and 200,000 edges. *GraphWaGu* allows users to pan and zoom with the graph rendering while the layout is being computed, and this shows that we can permit user interaction with graphs while rendering and computing layout for around 400,000 graphical marks. Also, with the use of *GraphWaGu* BH, we can scale our layout computation to a graph of 100,000 nodes and 2,000,000 edges while computing its convergence in less than three minutes.

4.4. Layout Quality Evaluation

In order to accurately measure the performance of the different layout creation algorithms, a comparison must be made on the quality of outputted graphs. There are many aesthetic metrics available [Pur02] which seek to quantitatively evaluate different ideal features in a 2-dimensional graph. For the sake of this paper, we have chosen the following three metrics to verify the quality of graphs outputted by D3, *GraphWaGu* FR, and *GraphWaGu* BH.

Edge Uniformity: Edge uniformity (EU) is a measure of the variance of a layout's edge lengths. We calculate this measure in the same manner as defined by [RSA20]: the normalized standard deviation of edge length is computed as $\sqrt{\frac{\sum_{e \in E}(l_e - l_\mu)^2}{|E| * l_\mu^2}}$ where l_μ is the average length of all edges. Because a higher value corresponds to greater standard deviation among edge lengths, a lower value of EU is a better aesthetic measure for a layout.

Stress: Stress (S) measures the difference between the input graph theoretical distances and output realized 2-dimensional distances for each pair of vertices in a layout. The graph theoretical distance between two vertices is determined with regard to the shortest path of edges between them, with closer to adjacent vertices having lesser theoretical distance. The realized layout distance is the actual Euclidean distance between the two vertices in an output layout. Stress is computed for a layout as the sum of the differences of these two measures for each pair of vertices in the graph. A lower value of stress corresponds to a better layout as it means depicting the input graph more accurately.

Neighborhood Preservation: Neighborhood Preservation (NP) evaluates the precision of a layout in maintaining the proximity of nodes when translating from graph space to 2-dimensional space. To do this, for each vertex v we first extract the neighborhood

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

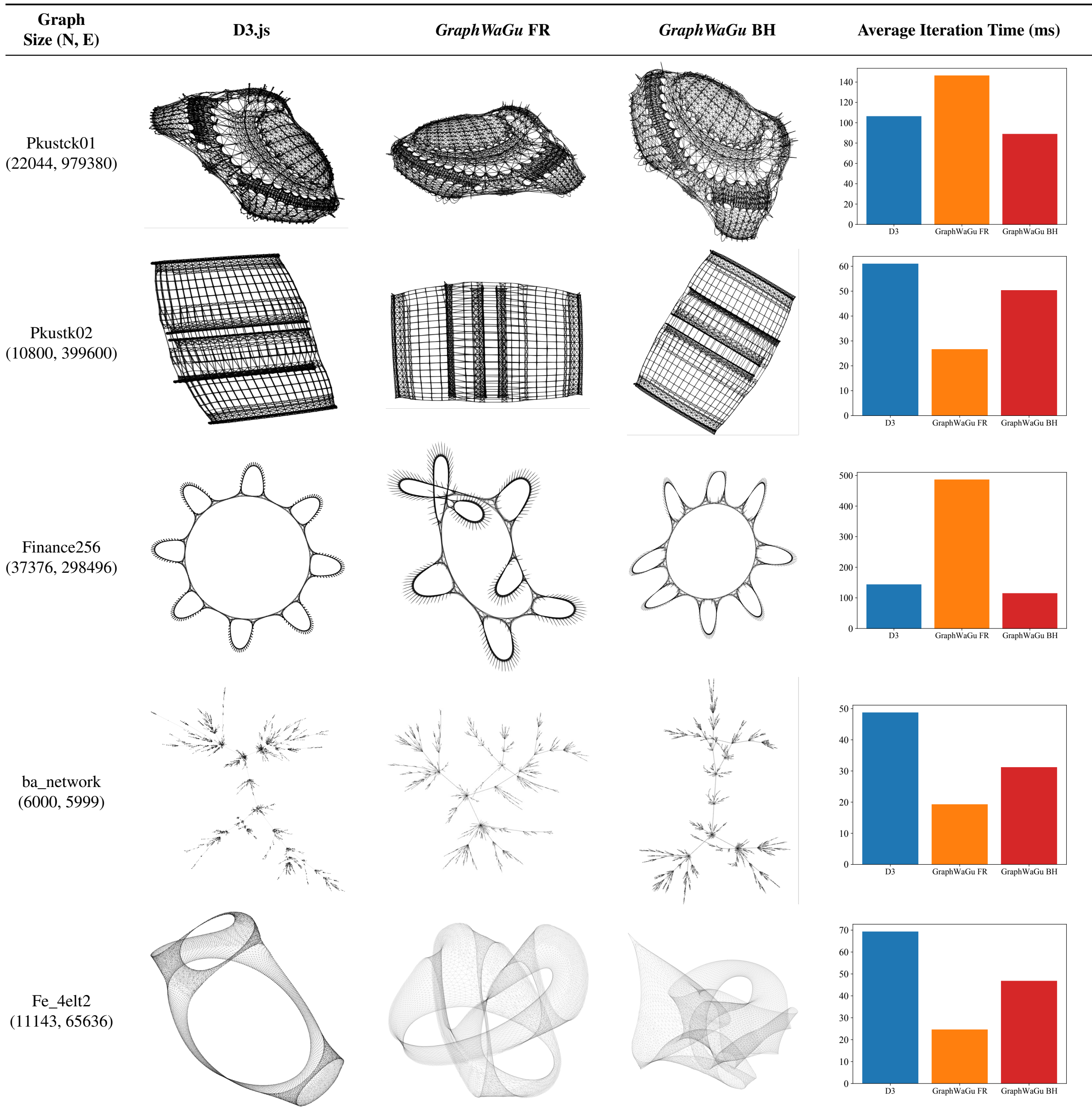

Table 1: *Layout and performance comparisons of our approach (*GraphWaGu *FR and* GraphWaGu *BH) with D3.*

Graph	Quality metric								
	D3-EU	**FR-EU**	**BH-EU**	**D3-S**	**FR-S**	**BH-S**	**D3-NP**	**FR-NP**	**BH-NP**
Pkustk01	8.3E-01	8.3E-01	**7.7E-01**	3.0E+07	3.6E+07	**2.5E+07**	4.4E-01	4.3E-01	**4.4E-01**
Pkustk02	8.1E-01	7.3E-01	**6.9E-01**	1.3E+08	9.8E+07	**1.1E+08**	3.3E-01	**3.4E-01**	3.3E-01
Finance_256	1.3E+00	9.2E-01	**8.8E-01**	3.9E+08	**3.2E+08**	3.7E+08	3.4E-02	4.8E-02	**1.8E-02**
ba_network	1.1E+00	1.6E+00	**1.1E+00**	9.1E+06	**7.3E+06**	7.9E+06	2.9E-02	**7.4E-02**	3.6E-02
Fe_4elt2	4.8E-01	6.3E-01	**4.5E-01**	4.2E+07	2.7E+07	**2.5E+07**	1.3E-01	**1.8E-01**	1.0E-01

Table 2: *Quality metrics for the layouts depicted in Table 1. In bold is the best result for each metric: edge uniformity (EU), stress (S), neighborhood preservation (NP).*

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

$N_G(v_i, r_G)$ of vertices connected to v_i via a path of at most r_G edges. We then create the neighborhoods $N_L(v_i)$ from a given layout L as the closest $|N_G(v_i, r_G)|$ vertices to each v_i using Euclidean distance. The NP for this layout and graph is then the Jaccard similarity between these two neighborhoods, averaged over the vertices of the graph. This gives a measure which shows the precision of the layout neighborhoods in representing the input graph neighborhoods. This method is shown to be useful in [KRM*17] with r_G of 2, and $r_G \in \{1, 3\}$ are indicated to yield similar results. For our NP calculations we choose $r_G = 1$. A higher value of NP means a better layout, with a value of 1 meaning that all vertices preserve all their neighbors.

As shown in Table 1, we compute the layout of five real graphs from the SuiteSparse Matrix Collection [KAB*19] in D3.js, *GraphWaGu* FR, and *GraphWaGu* BH on the dedicated GPU. To fairly compare the quality of the layouts being output, and to ensure that they would reach convergence of energy minimization, each algorithm ran for 2000 iterations for each dataset. Each layout was then rendered using *GraphWaGu* for consistency, and this output is shown. In terms of performance, we observe the same behaviour experienced in our benchmarks (see Figure 6), where the *GraphWaGu* outperforms D3.js for small graphs (with both approaches) and large graphs (with the BH approach).

In layout creation, better performance must also be accompanied by similar or better quality layouts or a comparison between the algorithms being used is not possible. Qualitatively, one can judge that the layouts shown in Table 1 seem similar, although they are not identical due to differences in algorithmic design. Because of this, we compute the quality metrics described above to judge quantitatively the aesthetic standard of all the layouts, and depict the results in Table 2. For edge uniformity (EU), *GraphWaGu* BH is the winner for all 5 graphs. For both stress (S) and neighborhood preservation, either *GraphWaGu* FR or *GraphWaGu* BH has the best score for all datasets, although D3 is comparable for most. Overall, these results point that *GraphWaGu* creates layouts of equal or better quality then D3 when using both the FR and BH algorithms. These results validate our layout generation works for a diverse set of data and that *GraphWaGu* has the potential to generate and render aesthetically pleasing layouts with superior performance to existing web-based graph visualization libraries.

5. Conclusions

We have presented *GraphWaGu*, the first WebGPU based graph visualization tool that enables parallel layout computation and rendering for large scale graphs in the browser. We have implemented a WebGPU based layout computation for both Frutcherman-Reingold and Barnes-Hut algorithms. Performance scaling studies have demonstrated that *GraphWaGu* can achieve the best performance for both small (thousands of edges) and large graphs (millions of edges) when compared with existing state-of-the-art web based visualization libraries. The layout computation and rendering performance of *GraphWaGu* pave the way for a new generation of web-based graph visualization tools that leverage the full power of the GPU.

6. Acknowledgement

This work was funded in part by NSF RII Track-4 award 2132013.

References

[AN16] ARIDHI S., NGUIFO E. M.: Big graph mining: Frameworks and techniques. *Big Data Research 6* (2016), 1–10. 1

[AVHK06] ABELLO J., VAN HAM F., KRISHNAN N.: Ask-graphview: A large scale graph visualization system. *IEEE transactions on visualization and computer graphics 12*, 5 (2006), 669–676. 1

[BH86] BARNES J., HUT P.: A hierarchical o (n log n) force-calculation algorithm. *nature 324*, 6096 (1986), 446–449. 2, 4

[BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D³ data-driven documents. *IEEE Transactions on Visualization and Computer Graphics* (2011). doi:10.1109/tvcg.2011.185. 1, 2

[BP11] BURTSCHER M., PINGALI K.: An efficient cuda implementation of the tree-based barnes hut n-body algorithm. In *GPU computing Gems Emerald edition*. Elsevier, 2011, pp. 75–92. 2, 4

[BRT17] BRINKMANN G. G., RIETVELD K. F., TAKES F. W.: Exploiting gpus for fast force-directed visualization of large-scale networks. In *2017 46th International Conference on Parallel Processing (ICPP)* (2017), IEEE, pp. 382–391. 2

[Cab] CABELLO R.:. https://github.com/mrdoob/stats.js/. 7

[CBP14] CRUZ J. D., BOTHOREL C., POULET F.: Community detection and visualization in social networks: Integrating structural and semantic information. *ACM Transactions on Intelligent Systems and Technology (TIST) 5*, 1 (2014), 1–26. 1

[CDGP20] CHEN X., DATHATHRI R., GILL G., PINGALI K.: Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proceedings of the VLDB Endowment 13*, 8 (2020), 1190–1205. 1

[Coe18] COENE J.-P.: sigmajs: An r htmlwidget interface to the sigma. js visualization library. *Journal of Open Source Software 3*, 28 (2018), 814. 3

[con22a] CONTRIBUTORS M.: Canvas API. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API, 2022. [Online; accessed 11-March-2022]. 2

[con22b] CONTRIBUTORS M.: SVG. https://developer.mozilla.org/en-US/docs/Web/SVG, 2022. [Online; accessed 11-March-2022]. 2

[DGH*19] DATHATHRI R., GILL G., HOANG L., JATALA V., PINGALI K., NANDIVADA V. K., DANG H.-V., SNIR M.: Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2019), IEEE, pp. 15–28. 1

[Ead84] EADES P.: A heuristic for graph drawing. *Congressus numerantium 42* (1984), 149–160. 2

[EF96] EADES P., FENG Q.-W.: Multilevel visualization of clustered graphs. In *International symposium on graph drawing* (1996), Springer, pp. 101–112. 1

[FLH*16] FRANZ M., LOPES C. T., HUCK G., DONG Y., SUMER O., BADER G. D.: Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics 32*, 2 (2016), 309–311. 1, 2

[FR91] FRUCHTERMAN T. M., REINGOLD E. M.: Graph drawing by force-directed placement. *Software: Practice and experience 21*, 11 (1991), 1129–1164. 2

[Gar82] GARGANTINI I.: An effective way to represent quadtrees. *Commun. ACM 25*, 12 (dec 1982), 905–910. URL: https://doi.org/10.1145/358728.358741, doi:10.1145/358728.358741. 4

[GKS94] GRAMA A. Y., KUMAR V., SAMEH A.: Scalable parallel formulations of the barnes-hut method for n-body simulations. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing* (1994), IEEE, pp. 439–448. 2

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

[Gov19] GOVE R.: A random sampling o (n) force-calculation algorithm for graph layouts. In *Computer Graphics Forum* (2019), vol. 38, Wiley Online Library, pp. 739–751. 2

[Gro22] GROUP K.: WebGL: LOW-LEVEL 3D GRAPHICS API BASED ON OPENGL ES. https://www.khronos.org/api/webgl, 2022. [Online; accessed 11-March-2022]. 2

[HJC*19] HOANG L., JATALA V., CHEN X., AGARWAL U., DATHATHRI R., GILL G., PINGALI K.: Disttc: High performance distributed triangle counting. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)* (2019), IEEE, pp. 1–7. 1

[HK00] HAREL D., KOREN Y.: A fast multi-scale method for drawing large graphs. In *International symposium on graph drawing* (2000), Springer, pp. 183–196. 2

[HKUH17] HIDAKA M., KIKURA Y., USHIKU Y., HARADA T.: Webdnn: Fastest dnn execution framework on web browser. In *Proceedings of the 25th ACM International Conference on Multimedia* (New York, NY, USA, 2017), MM '17, Association for Computing Machinery, p. 1213–1216. URL: https://doi.org/10.1145/3123266.3129394, doi:10.1145/3123266.3129394. 3

[HMM00] HERMAN I., MELANÇON G., MARSHALL M. S.: Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on visualization and computer graphics 6*, 1 (2000), 24–43. 1

[HPZC21] HAN D., PAN J., ZHAO X., CHEN W.: Netv. js: A web-based library for high-efficiency visualization of large-scale graphs and networks. *Visual Informatics 5*, 1 (2021), 61–66. 2, 3, 6

[JDG*20] JATALA V., DATHATHRI R., GILL G., HOANG L., NANDIVADA V. K., PINGALI K.: A study of graph analytics for massive datasets on distributed multi-gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2020), IEEE, pp. 84–94. 1

[JVHB14] JACOMY M., VENTURINI T., HEYMANN S., BASTIAN M.: Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one 9*, 6 (2014), e98679. 2

[KAB*19] KOLODZIEJ S. P., AZNAVEH M., BULLOCK M., DAVID J., DAVIS T. A., HENDERSON M., HU Y., SANDSTROM R.: The suitesparse matrix collection website interface. *Journal of Open Source Software 4*, 35 (2019), 1244. 6, 10

[Kar12] KARRAS T.: Maximizing parallelism in the construction of bvhs, octrees, and <i>k</i>-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Goslar, DEU, 2012), EGGH-HPG'12, Eurographics Association, p. 33–37. 4

[KG19] KUMAR S., GILRAY T.: Distributed relational algebra at scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE* (2019). 1

[KK*89] KAMADA T., KAWAI S., ET AL.: An algorithm for drawing general undirected graphs. *Information processing letters 31*, 1 (1989), 7–15. 2

[KKVF15] KOUTRA D., KANG U., VREEKEN J., FALOUTSOS C.: Summarizing and understanding large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal 8*, 3 (2015), 183–202. 1

[KRM*17] KRUIGER J., RAUBER P., MARTINS R., KERREN A., KOBOUROV S., TELEA A.: Graph layouts by t-sne. *Computer Graphics Forum 36* (06 2017), 283–294. doi:10.1111/cgf.13187. 10

[LWZ15] LIPP F., WOLFF A., ZINK J.: Faster force-directed graph drawing with the well-separated pair decomposition. In *International Symposium on Graph Drawing* (2015), Springer, pp. 52–59. 2

[NLP13] NGUYEN D., LENHARTH A., PINGALI K.: A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles* (2013), pp. 456–471. 1

[Pli21] PLIQUE G.: Graphology, a robust and multipurpose Graph object for JavaScript. Zenodo. https://www.khronos.org/api/webgl, 2021. [Online; accessed 11-March-2022]. 3

[Pur02] PURCHASE H. C.: Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing 13*, 5 (2002), 501–516. URL: https://www.sciencedirect.com/science/article/pii/S1045926X02902326, doi:https://doi.org/10.1006/jvlc.2002.0232. 8

[RLH17] REN D., LEE B., HÖLLERER T.: Stardust: Accessible and transparent gpu support for information visualization rendering. In *Computer Graphics Forum* (2017), vol. 36, Wiley Online Library, pp. 179–188. 1, 2

[RSA20] RAHMAN M. K., SUJON M. H., AZAD A.: Batchlayout: A batch-parallel force-directed graph layout algorithm in shared memory. *CoRR abs/2002.08233* (2020). URL: https://arxiv.org/abs/2002.08233, arXiv:2002.08233. 8

[Sam90] SAMET H.: *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, Inc, 1990. 4

[SMS*17] SAHU S., MHEDHBI A., SALIHOGLU S., LIN J., ÖZSU M. T.: The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment 11*, 4 (2017), 420–431. 1

[TFS*15] TEIXEIRA C. H., FONSECA A. J., SERAFINI M., SIGANOS G., ZAKI M. J., ABOULNAGA A.: Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 425–440. 1

[TM08] TIKHONOVA A., MA K.-L.: A scalable parallel force-directed graph layout algorithm. In *Proceedings of the 8th Eurographics conference on Parallel Graphics and Visualization* (2008), pp. 25–32. 2

[UP20] USHER W., PASCUCCI V.: Interactive visualization of terascale data in the browser: Fact or fiction? In *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)* (2020), pp. 27–36. doi:10.1109/LDAV51489.2020.00010. 3

[Web] WebGPU. https://gpuweb.github.io/gpuweb/. 1

[wgs] WebGPU Shading Language. https://www.w3.org/TR/WGSL/. 2

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.