Accelerating Web-Based Graph Drawing with Bottom-Up GPU Quadtree Construction

Landon Dyken[*]
University of Illinois Chicago

Will Usher
Luminary Cloud

Steve Petruzza
Utah State University

Stavros Sintos
University of Illinois Chicago

Sidharth Kumar
University of Illinois Chicago

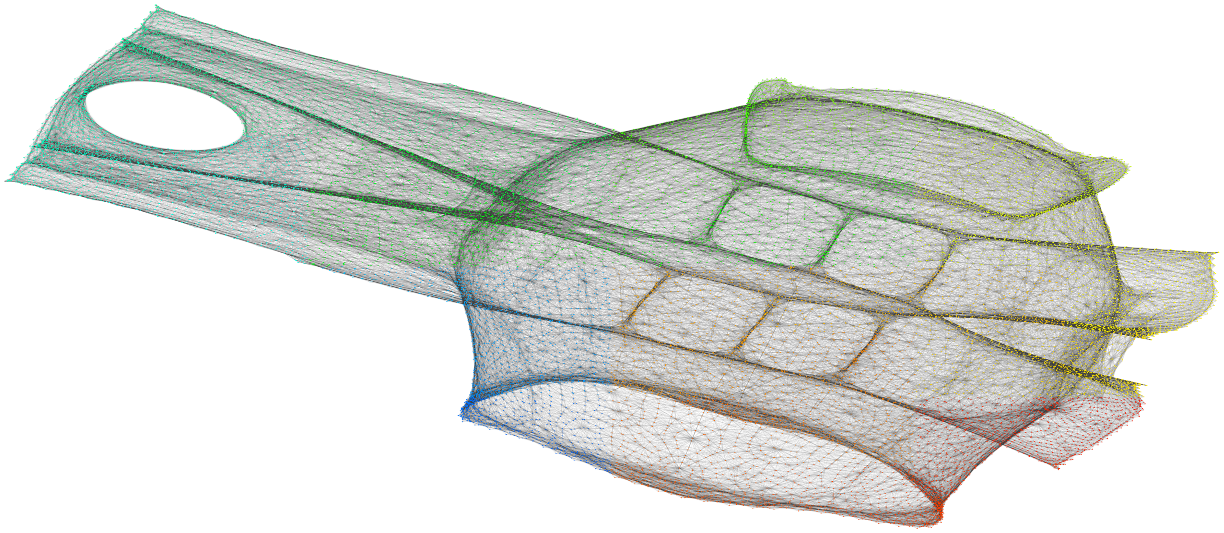


Figure 1: Example graph drawn for the Pkustk13 dataset from the SuiteSparse matrix collection [13], consisting of 94,893 vertices and 6,616,827 edges. This layout was computed in 1000 iterations of our force-directed algorithm, taking only 5.48 seconds on an Nvidia RTX 4070 Laptop GPU. Vertices are colored by their Hilbert code spatial ordering, which we use to efficiently build a quadtree on GPU for force approximation.

Abstract

Graph drawing, or graph layout creation, is a computationally difficult challenge in visualization that involves placing the vertices of a graph into a layout that provides insight into its structure. In order to visualize large-scale graphs, effective layouts are necessary for understanding. Previous work has shown the potential for graph drawing directly in the web browser by using WebGPU, a new API that brings the full capabilities of modern GPUs to the web. Compared to the existing state-of-the-art for web-based graph visualization, which rely on CPU-based graph drawing algorithms, WebGPU-accelerated work improves performance and scalability. However, we find that existing WebGPU solutions utilize sub-optimal quadtree data structures for graph drawing. In this work, we implement a modified quadtree data structure that uses a Hilbert spatial ordering for a fully parallelizable bottom-up construction algorithm in WebGPU. We utilize this data structure, along with optimizations to the quadtree traversal, to propose a massively more performant graph drawing algorithm. We evaluate the performance of our work against the existing state-of-the-art and demonstrate up to 69.5× speed-ups for layout creation of relevant graphs while enabling graph drawing for datasets of much larger size.

Keywords: Graph layout creation, web-based, quadtree, GPU.

[*]e-mail: ldyke@uic.edu

Index Terms: Human-centered computing [Visualization]: Visualization techniques—Graph drawings ;

1 Introduction

Efficient visualization of large-scale graphs is critical for numerous fields such as social networks, health science, web search, and road maps [2, 16, 15]. For this task, one of the key problems is *graph drawing*, which projects a graph, $G = (V, E)$, where V is the set of vertices and E is the set of edges, onto a $2D$ plane such that each vertex $v \in V$ is assigned a position P_v. The goal of graph drawing is to compute these positions in a way that visually captures the structure of the underlying graph. This is a computationally demanding challenge, especially as we see the scale of graph applications continuing to increase steadily [20].

At the same time, as software is increasingly accessed through the web rather than through desktop applications, the browser has emerged as the go-to for deploying visualization tools. This further complicates the challenge of graph drawing, as existing solutions for the web offer poor performance for large-scale graphs due to their reliance on CPU-based algorithms. To address this, GraphWaGu [5] utilized WebGPU [22] to create a framework for GPU-accelerated graph layout creation and rendering in the web that outperforms existing web-based graph libraries.

However, while the rest of their graph drawing algorithm is parallelized fully on GPU, GraphWaGu relies on a single-threaded algorithm for quadtree [7] construction, where all vertices of the graph are inserted one after another to build the quadtree in a top-down manner. This greatly constrains the scalability and performance of their system. In our work, we take inspiration from recent work in unstructured volume rendering [18] to propose a modified quadtree data structure that utilizes a Hilbert spatial ordering for a fully parallelizable bottom-up construction algorithm in WebGPU. We uniquely apply this data structure for Barnes-Hut approximation [1] in our force-directed graph drawing algorithm. Additionally, we improve upon GraphWaGu's algorithm by optimizing quadtree traversal for repulsive force computation. Our evaluation shows that our web-based system outperforms GraphWaGu by massive margins on relevant graph sizes while enabling graphs of even larger scale. In summary, our paper makes the following contributions:

- Developed a parallel WebGPU-based bottom-up quadtree construction algorithm using Hilbert spatial ordering for use in Barnes-Hut force-directed graph drawing, resulting in huge performance improvement over the state-of-the-art in web-based graph visualization
- Utilized this quadtree with an optimized traversal approach for graph drawing, resulting in interactive computation for much larger graphs than previously supported (almost 100,000 nodes and 6 million edges)

2 Background and Related Work

The most widely used methods for creating insightful graph layouts are *force-directed* algorithms [6, 12, 8], where forces are modeled between vertices in the input graph, pushing them to positions that fit the graph's structure. These algorithms compute repulsive forces between all pairs of vertices that repel them from each other, along with attractive forces between vertices connected by an edge, pulling them together. These forces are modeled iteratively, and the vertices of the graph are moved every iteration until a suitable layout is attained.

In each iteration, while the cost to compute attractive forces is $O(|E|)$, the cost for directly computing repulsive forces is $O(|V|^2)$, making it infeasible for large graphs. There have been many works that address this challenge [8, 11, 1, 14, 9] which reduce the repulsive force computation to $O(|V|\log|V|)$ or $O(|V|)$ through methods such as approximation or random sampling. Among these, the Barnes-Hut (BH) approximation is one of the most widespread techniques. This method constructs a quadtree over the input graph's vertices, augmented with the center of mass and total mass contained in every quadtree node. When computing repulsive forces for a vertex, the quadtree is traversed recursively from the root. If the distance between the vertex and the node's center of mass is above a certain threshold, the node is used to approximate forces for all vertices contained within it; otherwise, all of the node's children are added to the stack for further traversal. This leads to an expected $O(|V|\log|V|)$ running time, as each vertex greatly reduces the number of vertices it needs to compute repulsive forces against.

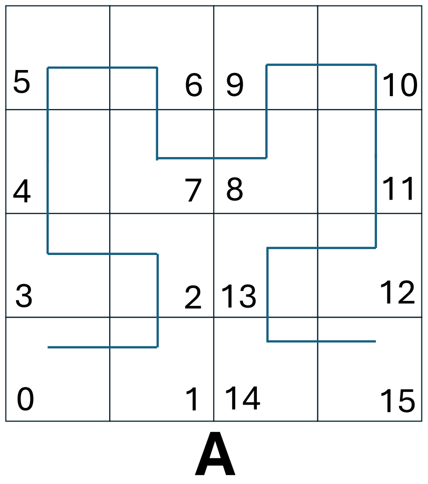
In this work, we build a system for web-based graph visualization that extends GraphWaGu [5], which uses a graph drawing algorithm with BH approximation where attractive and repulsive forces are computed on GPU. While GraphWaGu [5] presents the first force-directed graph algorithm in WebGPU, parallel implementations of both force-directed algorithms and quadtree construction have been explored in earlier research. One of the most important works is Warren and Salman [21], who propose a method for distributed quadtree construction, which was also adapted for force-directed graph drawing by Rahman et al. [19] in a native OpenMP setting. This method begins by assigning a code to each item being inserted into the quadtree by using its spatial Morton order (also known as Z-order). By sorting based on these codes, the items are put in order of spatial proximity, where spatially close items are close in the sorted list. After the sort, the items are divided into sets, and each processor creates its own quadtree over its set of items, giving a distributed quadtree. This work was extended by Grama et al. [10] for better load balancing through improved domain decomposition and assignment of subdomains. Grama et al. also replaced the use of Morton codes with Hilbert codes. While these works construct a quadtree in parallel, they are built for using multiple distributed CPU cores, not for GPU acceleration.

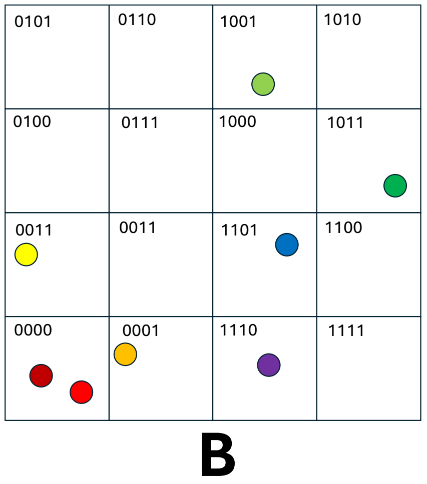
Algorithm 1 Complete pseudocode for our algorithm described in Section 3.

```
1:  Input: $G(V,E), coolingFactor, b, \theta$
2:  while $coolingFactor \geq \varepsilon$ do
3:      for $v \leftarrow 0$ to $|V|$ do in parallel
4:          $H[v] \leftarrow HilbertCode(V[v])$
5:      end for
6:      $sortByHilbertCodes(V)$
7:      for $v \leftarrow 0$ to $|V|$ do in parallel
8:          $T[v] \leftarrow TreeNode(v)$
9:      end for
10:     $s \leftarrow 0$
11:     for $i \leftarrow 0$ to $\log_b |V|$ do
12:         $e \leftarrow s + |V|/b^i$
13:         for $j \leftarrow 0$ to $\frac{|V|}{b^{i+1}}$ do in parallel
14:             $T[e+j] \leftarrow Merge(T[s+j*b], ..., T[s+j*b+b-1])$
15:         end for
16:         $s \leftarrow e$
17:     end for
18:     $computeAttractiveForces()$
19:     for $v \leftarrow 0$ to $|V|$ do in parallel
20:         $node \leftarrow root, counter \leftarrow 0, stack[64]$
21:         while $node \neq null$ do
22:             if $\theta > \frac{2.0*node.size}{distance(V[v],node.CoM)}$ then
23:                 $F[v] \leftarrow F[v] + node.mass * f_r(V[v], node.CoM)$
24:             else
25:                 for $child$ of $node$ do
26:                     $stack[counter] = child$
27:                     $counter \leftarrow counter + 1$
28:                 end for
29:             end if
30:             $counter \leftarrow counter - 1$
31:             $node = stack[counter]$
32:         end while
33:     end for
34:     for $i \leftarrow 0$ to $|V|$ do in parallel
35:         $V[i].position \leftarrow V[i].position + F[i] * coolingFactor$
36:         $F[i] \leftarrow 0$
37:     end for
38:     $coolingFactor \leftarrow coolingFactor * initialCoolingFactor$
39: end while
```

A popular GPU-accelerated graph drawing algorithm is the ForceAtlas2 implementation by Brinkmann et al. [3], who utilize a CUDA implementation of BH approximation [4] with an irregular tree-based structure and complex traversal. Later work by Zhang and Gruenwald [23], which does not touch on BH approximation or graph drawing, shows how to improve the performance of quadtree construction on GPU by using a bottom-up approach. Here, they sort all items by their Morton code again (using a GPU radix sort), then create the full quadtree directly in one GPU buffer. They do this by successively applying a CUDA *reduce_by_key* primitive to build up the tree from the leaves to the root and show that their performance is faster than other parallel implementations.

In order to take advantage of GPU acceleration for quadtree construction, we take inspiration from Zhang and Gruenwald [23] and create a bottom-up algorithm. We differ from their implementation





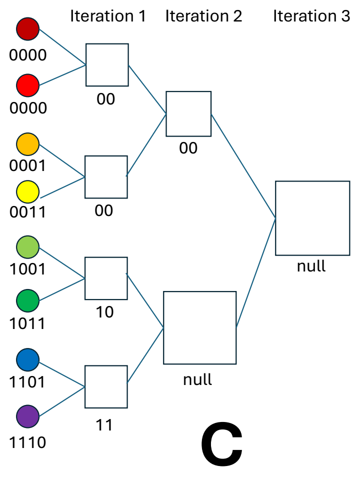


Figure 2: Illustration of our quadtree construction algorithm with a branching factor of 2 on a small example where the minimum spatial subdivision is $1/2^2$. A) shows the Hilbert spatial curve on this space, ordering the cells by spatial proximity beginning with the bottom left and ending with the bottom right. B) shows an input graph defined by 8 vertices in this space. Hilbert codes are computed for each vertex using the ordering from earlier, converting the decimal numbers into binary codes, and assigning them to the vertices contained in each cell. Vertices are colored by their Hilbert codes. C) shows the process of bottom-up construction of the tree. The vertices are first sorted based on their Hilbert codes, leading to the ordering shown. Then, the next levels of the tree are iteratively computed by finding the shared prefix in the Hilbert codes of neighboring nodes in the previous level. These shared prefixes correspond to spatial partitions, for example with 00 and 10 representing the bottom left and top right quadrants of the space.

in several key aspects: WebGPU's lack of a *reduce_by_key* primitive necessitates us to implement our own method for merging nodes to create the levels of the tree; our quadtree nodes store additional data (center of mass and total mass) for BH approximation; we optimize memory by eliminating empty child nodes; and we parameterize node branching factors rather than fixing them at four. Additionally, insights from unstructured volume rendering [18] guide our choice of Hilbert over Morton ordering to improve spatial locality during tree construction.

3 Implementation

We propose a web-based algorithm for force-directed graph drawing that can fully utilize the capabilities of modern GPUs. To do this, we begin from the starting point of GraphWaGu, creating an iterative algorithm that models attractive and repulsive forces between vertices to gradually move them towards the lowest energy arrangement. For convergence, a cooling factor is used that slowly reduces the forces being applied every iteration. Because we found their method for attractive force computation to be effective, our algorithm focuses on improving the computation of repulsive forces by overhauling their quadtree construction and traversal methods.

The complete pseudocode for our algorithm is given in Algorithm 1. The input to our algorithm consists of the graph $G(V,E)$, the cooling factor (usually set between 0.95 and 0.99), the branching factor b for our modified quadtree, and the approximation factor θ. Each iteration of our algorithm begins by constructing our modified quadtree in parallel, which occurs in Lines 3-17 in the pseudocode and is described in Section 3.1. We then compute attractive forces by using the method of GraphWaGu, shown in Line 18. After this, we use our quadtree to compute repulsive forces in parallel, which occurs in Lines 19-33 in the pseudocode and is described in Section 3.2. Finally, we apply the forces to the vertices of the input graph in parallel and update the cooling factor in Lines 34-38 of the pseudocode.

3.1 Quadtree Construction

The first step of constructing our modified quadtree is to compute Hilbert codes for each vertex of the input graph in parallel on GPU (Lines 3-5 in Algorithm 1). This is done for each vertex by converting its x and y coordinates to 16 bit unsigned integers, then using bit shift and rotation operations to encode both into one 32 bit unsigned integer code according to the Hilbert spatial ordering. After this, we use a GPU radix sort implemented in WebGPU to sort the vertices of the graph according to these Hilbert codes (Line 6 in Algorithm 1). This step is crucial, as it places vertices together in the buffer based on their spatial proximity. Next, we create the leaf nodes of the tree in parallel (Lines 7-9 in Algorithm 1), writing out each vertex's minimum containing node with a mass of 1, center of mass (CoM) at the vertex's position, and size of $1/2^{16}$. This size comes from the fact that we are using 16 bits to encode the x and y coordinates of each vertex in a Hilbert code, meaning our minimum spatial subdivision is of size $1/2^{16}$. Because these nodes were built in order from the sorted vertices, they will also be written in the tree with preserved spatial proximity.

Once the leaf nodes of the tree are created, we can apply a bottom-up approach to build the higher levels in parallel through successively merging adjacent nodes. While a quadtree typically subdivides nodes into 4 at every level, we allow this branching factor b to be given as a parameter to our algorithm, although experimentally we found best performance using the typical value of 4. This bottom-up merging process can be seen in Lines 10-17 in Algorithm 1. The merging process iterates $\log_b|V|$ times, each iteration i successively creating one of the $\log_b|V|$ levels of the tree until the root is reached. Each iteration dispatches $|V|/b^{i+1}$ GPU threads, where each thread merges b adjacent nodes at the current level to create a new node at the level one higher. s and e are used as the starting and ending indices to find the nodes that are being merged at the current iteration.

For our function to merge the b nodes in each thread (defined as "Merge" in the pseudocode), we utilize the fact that each node has a corresponding Hilbert code. We compute the containing node for the b nodes being merged by comparing their Hilbert codes and finding the shared code prefix between all nodes. Because any prefix of a Hilbert code corresponds to a higher level subdivision that contains the original Hilbert code, this prefix effectively gives us the minimum-size containing node for the nodes being merged. The

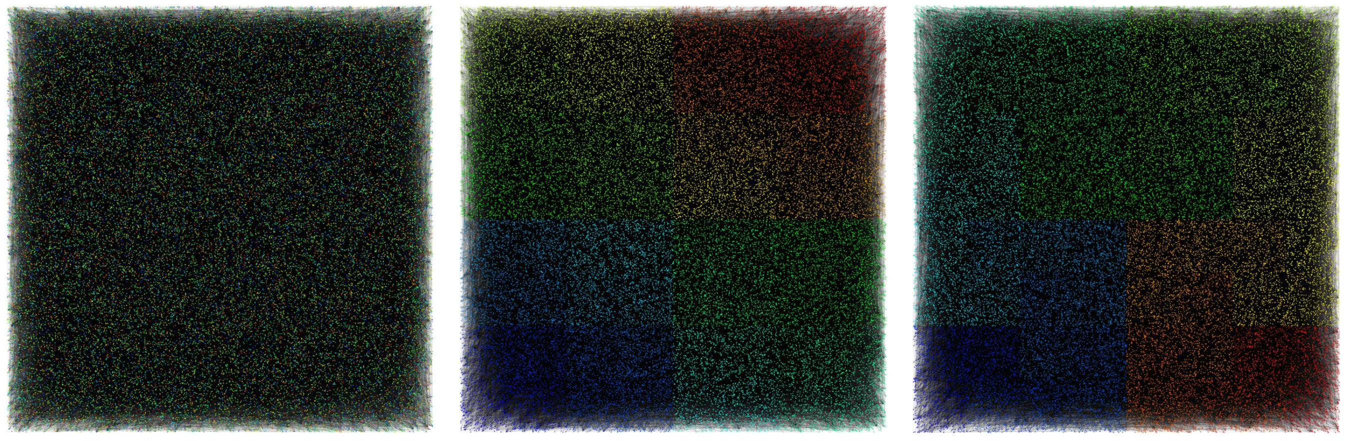


Figure 3: Illustration of quadtrees created through bottom-up construction using either unsorted vertices, vertices sorted by their Morton codes, or vertices sorted by their Hilbert codes. Vertices are colored by their containing node in the quadtree using a rainbow colormap, giving a linear relationship between color similarity and distance between nodes in the quadtree. For quadtree traversal and approximation, it is important to have nodes contain only vertices that are spatially close to each other. The unsorted quadtree performs terribly, since vertices with the same color are distant from each other, resulting in inefficient spatial partitioning as the containing nodes become huge. The Morton quadtree is much better, but the z-order jumps also result in sub-optimal quadtree creation, as can be seen with the green vertices that are spatially distant from each other. Only the Hilbert quadtree enforces spatial proximity for quadtree nodes, due to the optimal clustering properties of Hilbert space-filling curves [17].

size of this new node is computed with $1/2^{p/2}$, where p is the number of bits in the nodes' shared prefix. The mass of the new node is computed as the sum of the masses of the merged nodes, and the CoM is computed as the average of the merged nodes' CoMs, weighting each by the nodes' mass. We illustrate a simple example of our quadtree construction algorithm in Figure 2.

This algorithm allows us to build our quadtree from the bottom up, recursively merging groups of neighboring vertices to create the spatial partitions of our graph for BH approximation. Because the vertices were initially sorted via a spatial ordering, the partitions created do not become too large and have minimal overlap. This is important for computation of repulsive forces, as inefficient partitioning will slow quadtree traversal and reduce the likelihood of the BH approximation condition being met, resulting in worse performance. We present a visual comparison between spatial partitions produced by unsorted vertices, Morton codes, and Hilbert codes in Figure 3.

3.2 Repulsive Force Computation

In GraphWaGu, repulsive force computation utilized a large pre-allocated storage buffer as a pseudo-stack to complete a breadth-first quadtree traversal for each vertex in parallel. While the rest of our algorithm for computing repulsive forces is similar, we remove the need to use a storage buffer for the stack by simply replacing the breadth-first quadtree traversal with a depth-first quadtree traversal. Removing the large stack buffer improves overall performance and enables our algorithm to compute layouts for larger graphs than GraphWaGu, as this buffer was the memory bottleneck for many input graphs.

Our repulsive force computation is shown in Lines 19-33 of Algorithm 1. This process dispatches a GPU thread for each vertex to compute its own repulsive forces in parallel. Beginning at the root, the thread traverses our quadtree data structure using a stack array of length 64. At each node, there is a check against the BH approximation condition (Line 22 in Algorithm 1), where forces are approximated for all vertices contained in that node if the condition is met. Otherwise, all of the node's children are added to the stack for traversal. Importantly, the traversal pops the last added element off the stack each iteration, rather than the first, resulting in a depth-first traversal and allowing for a much smaller array to maintain the current traversal's stack.

Dataset	Nodes	Edges
sf_ba6000	6,000	5,999
fe_4elt2	11,143	65,636
pkustk02	10,800	399,600
pkustk01	22,044	979,380
finance256	37,376	298,496
finance512	74,752	261,120
pkustk13	94,893	6,616,827
comYoutube	1,134,890	5,975,248

Table 1: Graphs used to evaluate graph drawing. The datasets vary in size and density to illustrate the scalability of our implementation. All of these graphs are undirected and two-dimensional.

4 Evaluation

In this section, we evaluate the performance improvement achieved by our force-directed layout algorithm compared to GraphWaGu. We outline the datasets used for this evaluation in Section 4.1, describe the experimental setup in Section 4.2, and present results in Section 4.3.

4.1 Datasets

To evaluate the performance of our algorithm, we use the same five real undirected graphs from the SuiteSparse Matrix Collection as in GraphWaGu [13, 5]. These datasets, shown in Table 1, were previously used to demonstrate that GraphWaGu's graph drawing outperforms the state-of-the-art CPU-based JavaScript visualization library D3.js. We also include three additional larger datasets from the same collection to show that our algorithm can continue to scale for even larger graphs. Notably, we include the comYoutube dataset, which consists of over 1.1 million nodes, dwarfing the largest dataset used by GraphWaGu by over 30x the number of nodes.

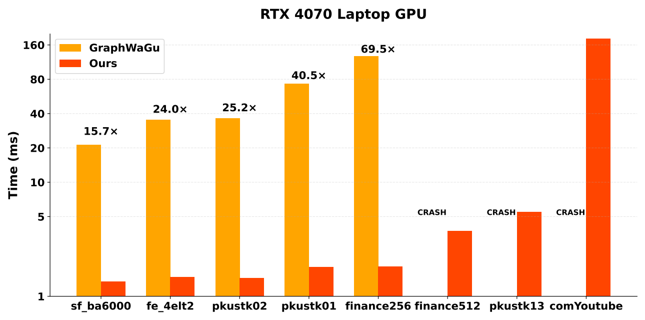


Figure 4: Average iteration times in milliseconds (logarithmic scale) for GraphWaGu and our algorithm on the RTX 4070 Laptop GPU, demonstrating speedups ranging from 15.7× to 69.5×. GraphWaGu was unable to compute layouts for the larger finance512, pkustk13, and comYoutube datasets.

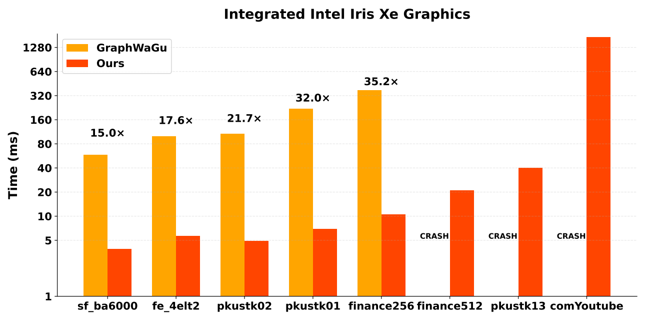


Figure 5: Average iteration times in milliseconds (logarithmic scale) for GraphWaGu and our algorithm on the integrated Intel Iris Xe Graphics, demonstrating speedups ranging from 15.0× to 35.2×. GraphWaGu was unable to compute layouts for the larger finance512, pkustk13, and comYoutube datasets.

4.2 Experimental Setup

The experimental environment for this project was a laptop system featuring an Intel core i9-13900H processor with 20 cores and a base clock speed of 2.60 GHz, paired with 32 GB of RAM and integrated Intel Iris Xe Graphics. The system also includes a dedicated NVIDIA GeForce RTX 4070 laptop GPU with 8 GB of dedicated VRAM. DirectX 12 was used as the graphics backend with the WebGPU framework. The web application was developed using React JS, and the benchmarks were executed using Node.js v22.12.

For our experiment, we measured the time taken to compute a graph layout for our chosen datasets with both GraphWaGu and our algorithm. In order to showcase the performance of our method on both dedicated and integrated GPUs, we computed our benchmarks on both the integrated Intel Iris Xe Graphics and NVIDIA GeForce RTX 4070 Laptop GPU of the experimental system. To ensure consistency and reliability, we ran both algorithms for 1000 iterations for each dataset and report the average time for one iteration. Because both algorithms have the same BH approximation condition, we used an approximation factor of 2 for all benchmarks to ensure the comparison is fair. All times are reported in milliseconds (ms).

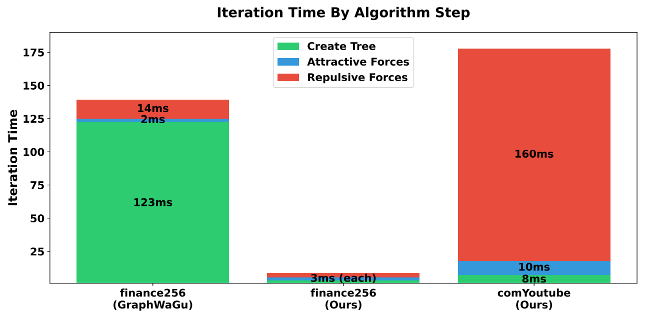


Figure 6: Average time taken by the different steps of GraphWaGu and our algorithm for selected datasets on the RTX 4070 Laptop GPU. Device synchronization to time each algorithm step adds some constant time (around 2-3ms). Results on the finance256 dataset show that our method greatly reduces the time for both tree creation and repulsive force computation compared to GraphWaGu. In addition, we show that on the much larger comYoutube graph, our tree creation step is still incredibly efficient, and the majority of the time comes from computing repulsive forces.

4.3 Results

Figure 4 and Figure 5 show the average iteration time comparison between GraphWaGu and our algorithm on the dedicated and integrated GPUs respectively. Across all data sizes and both systems, our algorithm consistently outperforms GraphWaGu by a significant margin. The results show that our algorithm achieves speedups ranging from 15.7× (*sf_ba6000*) to 69.5× (*finance256*) on the dedicated GPU, and from 15.0× (*sf_ba6000*) to 35.2× (*finance256*) on the integrated GPU, with the most significant improvements observed for larger datasets. Speedups for the integrated GPU are slightly less than for the dedicated GPU because the performance gain from parallelizing quadtree construction will necessarily be larger with a more powerful GPU.

In addition, the lower memory footprint achieved with our depth-first quadtree traversal for repulsive force computation enables layout creation for larger graphs while also improving total iteration performance. This can be seen in our results for the three larger example graphs that lead to memory errors in GraphWaGu (*finance512*, *pkustk13*, and *comYoutube*). We find on both dedicated and integrated GPUs that the iteration times for all graphs except comYoutube remain below the iteration time for even the smallest graph with GraphWaGu. Impressively, we show our method is able to compute one iteration of forces for pkustk13 in only 5.48 ms (182fps) on the dedicated GPU, despite it having almost 100,000 nodes and 6.6 million edges. In addition, although performance starts to suffer, we show that it is possible to run graph drawing using our algorithm even for a massive graph such as comYoutube (1.1 million nodes) directly in the browser. We note that, although we've restricted these benchmarks to using an approximation factor of 2, a higher approximation factor could be used to greatly improve the iteration time for this massive graph, enabling interactivity at the cost of some accuracy in the resulting layout. We leave analysis of this tradeoff to future work. Overall, these results confirm that our algorithm significantly reduces computation time for layout creation and enables better scalability, demonstrating the effectiveness of leveraging parallelism in WebGPU for scalable and efficient graph visualization.

To analyze the specific performance of the steps of our algorithm compared to GraphWaGu, we also include a breakdown for some of the iteration times using the RTX 4070 Laptop GPU in Figure 6. Iteration times are longer than presented in Figure 4 due to the added time from device synchronization to time each algorithm step (around 2-3ms). We compare the iteration times on finance256 for GraphWaGu and our method, along with comYoutube for our method. Results show that our method drastically improves both tree creation and repulsive force computation, although the tree creation improvement is much more significant. When scaling up to larger datasets, while the iteration times for GraphWaGu are dominated by the quadtree construction step, the majority of iteration time in our method comes from the computation of repulsive forces. From this, we see that the performance improvements of our method can be attributed to the parallelization of our tree construction algorithm, which effectively divides the workload among GPU threads, compared to the single-threaded construction algorithm used by GraphWaGu.

5 Conclusion

We have presented an approach for accelerating web-based graph drawing through a parallel bottom-up quadtree construction algorithm implemented in WebGPU. Our method significantly improves upon the state-of-the-art by utilizing Hilbert spatial ordering for efficient GPU quadtree construction and optimizing quadtree traversal for force computation. The experimental results demonstrate massive performance gains, with speedups ranging from 15.7× to 69.5× compared to GraphWaGu on a dedicated GPU, and 15.0× to 35.2× on integrated graphics. Our approach also enables layout creation of much larger graphs than previously possible in web-based environments, successfully handling graphs with almost 100,000 nodes and 6.6 million edges while maintaining interactive performance. Our approach not only improves computational efficiency, but also reduces memory requirements, making our solution particularly well-suited for web-based applications. By leveraging the full capabilities of modern GPUs through WebGPU, we demonstrate that graph visualization can be effectively implemented in web browsers without compromising on performance or scalability.

Acknowledgments

This work was partly funded by NSF Collaborative Research Awards 2401274 and 2221812, and NSF PPoSS Planning and Large awards 2217036 and 2316157.

References

[1] J. Barnes and P. Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986. 2

[2] O. Batarfi, R. E. Shawi, A. G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, and S. Sakr. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18:1189–1213, 2015. 1

[3] G. G. Brinkmann, K. F. Rietveld, and F. W. Takes. Exploiting gpus for fast force-directed visualization of large-scale networks. In *2017 46th International Conference on Parallel Processing (ICPP)*, pp. 382–391. IEEE, 2017. 2

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

[5] L. Dyken and P. Poudel. Graphwagu: Gpu powered large scale graph layout computation and rendering for the web. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2022. 1, 2, 4

[6] P. Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984. 2

[7] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4:1–9, 1974. 1

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

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

[10] A. Grama, V. Kumar, and A. Sameh. Scalable parallel formulations of the barnes-hut method for n-body simulations. In *Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pp. 439–448, 1994. doi: 10.1109/SUPERC.1994.344307 2

[11] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. In *International symposium on graph drawing*, pp. 183–196. Springer, 2000. 2

[12] T. Kamada, S. Kawai, et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989. 2

[13] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019. 1, 4

[14] F. Lipp, A. Wolff, and J. Zink. Faster force-directed graph drawing with the well-separated pair decomposition. In *International Symposium on Graph Drawing*, pp. 52–59. Springer, 2015. 2

[15] N. Liu, D.-s. Li, Y.-m. Zhang, and X.-l. Li. Large-scale graph processing systems: a survey. *Frontiers of Information Technology & Electronic Engineering*, 21(3):384–404, 2020. 1

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, 2010. 1

[17] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001. doi: 10.1109/69.908985 4

[18] N. Morrical, A. Sahistan, U. Gudukbay, I. Wald, and V. Pascucci. Quick clusters: A gpu-parallel partitioning for efficient path tracing of unstructured volumetric grids, 08 2022. doi: 10.13140/RG.2.2.34351.20648 2, 3

[19] M. K. Rahman, M. Haque Sujon, and A. Azad. BatchLayout: A Batch-Parallel Force-Directed Graph Layout Algorithm in Shared Memory . In *2020 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 16–25. IEEE Computer Society, Los Alamitos, CA, USA, June 2020. doi: 10.1109/PacificVis48177.2020.3756 2

[20] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment*, 11(4):420–431, 2017. 1

[21] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, p. 12–21. Association for Computing Machinery, New York, NY, USA, 1993. doi: 10.1145/169627.169640 2

[22] WebGPU. `https://gpuweb.github.io/gpuweb/`. 1

[23] J. Zhang and L. Gruenwald. Efficient quadtree construction for indexing large-scale point data on gpus: Bottom-up vs. top-down. In *ADMS@VLDB*, 2019. 2