TU WIEN Informatics

Improving the Performance of Edge-Path Bundling With WebGPU

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Yannic Ellhotka, MSc
Matrikelnummer 11776184

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Zweitbetreuung: Associate Prof. Dr.in techn. MSc Manuela Waldner

Wien, 7. August 2025

Yannic Ellhotka	Eduard Gröller

TU WIEN Informatics

Improving the Performance of Edge-Path Bundling With WebGPU

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Yannic Ellhotka, MSc
Registration Number 11776184

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Second advisor: Associate Prof. Dr.in techn. MSc Manuela Waldner

Vienna, August 7, 2025

Yannic Ellhotka | Eduard Gröller

Erklärung zur Verfassung der Arbeit

Yannic Ellhotka, MSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 7. August 2025

Yannic Ellhotka

Danksagung

Ich möchte mich herzlich bei meinem Betreuer Eduard Gröller und meiner Betreuerin Manuela Waldner für die wertvolle fachliche Unterstützung, die hilfreichen Anregungen sowie die konstruktive Kritik während des gesamten Arbeitsprozesses bedanken. Unsere regelmäßigen Meetings waren durchwegs unterhaltsam und haben mir geholfen, den Faden nicht zu verlieren.

Ein besonderer Dank gilt nochmals Manuela Waldner, da ihre Vorlesungen maßgeblich dazu beigetragen haben, dass ich nach längerer Pause endlich die Motivation gefunden habe, mein Studium abzuschließen.

Acknowledgements

I would like to sincerely thank my supervisor Eduard Gröller and my co-supervisor Manuela Waldner, for their valuable professional support, helpful suggestions, and constructive criticism throughout the entire working process. Our regular meetings were always enjoyable and helped me to stay on track.

A special thanks is extended to Manuela Waldner, as her lectures played a decisive role in motivating me, after a long break, to finally complete my studies.

Kurzfassung

Die Visualisierung großer Graphen ist aufgrund von visueller Unübersichtlichkeit, die wichtige Muster verdeckt, oft eine Herausforderung. Obwohl S-EPB eine effektive Technik zur Minderung dieses Problems ist, ist ihre CPU-basierte Implementierung für die interaktive Nutzung zu langsam, insbesondere in Web-Umgebungen. Diese Bachelorarbeit stellt einen P-EPB-Algorithmus vor, der diese Leistungseinschränkung behebt, indem er die massiv parallelen Verarbeitungsfähigkeiten moderner GPUs über die WebGPU-API nutzt.

Der vorgeschlagene Algorithmus überarbeitet die S-EPB-Technik für die parallele Ausführung, wobei der Fokus auf dem Finden und der anschließenden Optimierung der rechenintensivsten Aufgaben liegt: Spanner-Konstruktion und Kürzeste-Pfade-Berechnungen. Ein parallelisierter Floyd-Warshall-Algorithmus wird zusammen mit verschiedenen Spanner-Konstruktionsmethoden (Greedy und Theta-Graph) verwendet, um den P-EPB-Algorithmus als interaktive Webanwendung zu implementieren.

Eine detaillierte Auswertung vergleicht den WebGPU-basierten P-EPB-Algorithmus mit der CPU-basierten S-EPB-Implementierung. Die Ergebnisse zeigen, dass der P-EPB-Ansatz, insbesondere bei Verwendung eines Theta-Spanners, erhebliche Geschwindigkeitssteigerungen bei dichten Graphen liefert. Diese Arbeit unterstreicht das Potenzial von WebGPU, hochleistungsfähige, interaktive Graphvisualisierung und -analyse in webbasierten Umgebungen zu ermöglichen.

Abstract

Visualizing large graphs is often challenging due to visual clutter, which obscures important patterns. While Spanner-based Edge-Path Bundling (S-EPB) is an effective technique for mitigating this issue, its CPU-based implementation is too slow for interactive use, particularly in web environments. This thesis presents a Parallel Edge-Path Bundling (P-EPB) algorithm that addresses this performance limitation by utilizing the massively parallel processing capabilities of modern Graphics Processing Unit (GPU)s through the WebGPU API.

The proposed algorithm reengineers the S-EPB technique for parallel execution, with a focus on finding and then optimizing the most computationally demanding tasks: spanner construction and shortest-path calculations. A parallelized Floyd–Warshall algorithm, together with different spanner construction methods (greedy and theta-graph), is employed in implementing the P-EPB algorithm as an interactive web application.

A detailed evaluation compares the WebGPU-based P-EPB algorithm to the CPU-based S-EPB implementation. Results show that the P-EPB approach, particularly when using a theta-spanner, delivers substantial speedups on dense graphs. This work highlights the potential of WebGPU to enable high-performance, interactive graph visualization and analysis in web-based environments.

Contents

CHAPTER 1

Introduction

Graphs are a fundamental structure for representing and analyzing complex systems. Visualizing these graphs is crucial for data analysis, as it allows researchers and analysts to uncover patterns, identify outliers, and comprehend the underlying structure of the data. However, as datasets grow in size and complexity, their corresponding graph visualizations often become overwhelmingly dense [VLKS+11]. This visual clutter renders the visualization illegible, obscuring the very insights it is meant to reveal. The screen becomes a tangled mass of edges, making it impossible to trace connections or identify important pathways. An example of what is sometimes called a *Hairball* can be seen in Figure 1.1. To address this critical issue of visual clutter, various automated graph layout and simplification techniques have been developed, with edge bundling being one of the most prominent and effective.

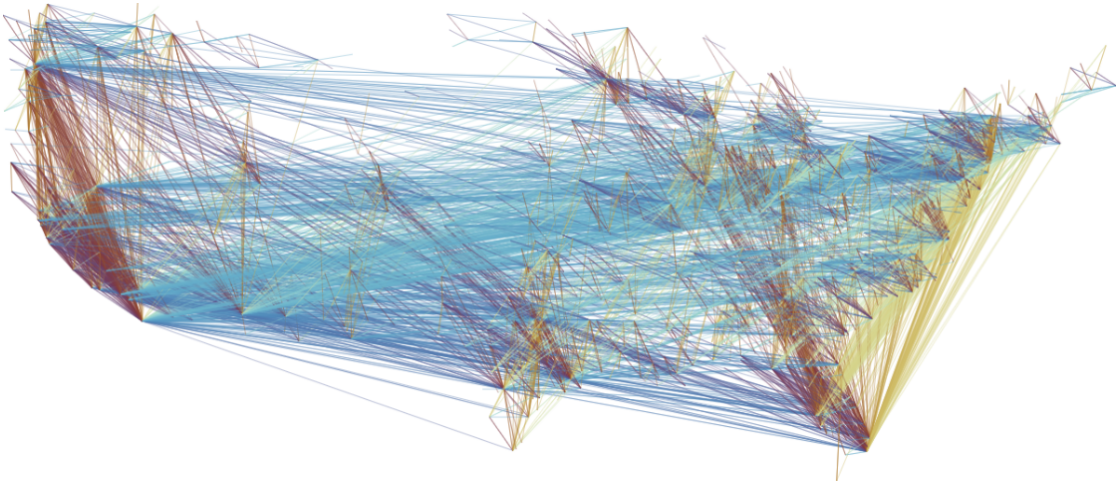


Figure 1.1: Visualization of a *Hairball*.

Edge bundling techniques aim to reduce visual clutter by grouping geometrically similar edges into bundles, creating a cleaner, more abstract representation of the graph's connectivity [WAA+21, WAA+23, HVW09]. An example of such a bundled graph can be seen in Figure 1.2. However, many traditional methods, such as force-directed bundling, can introduce significant independent edge ambiguities by merging edges that are not structurally related in the underlying graph [BRH+17]. This can mislead the viewer and compromise the faithfulness of the visualization to the original data.

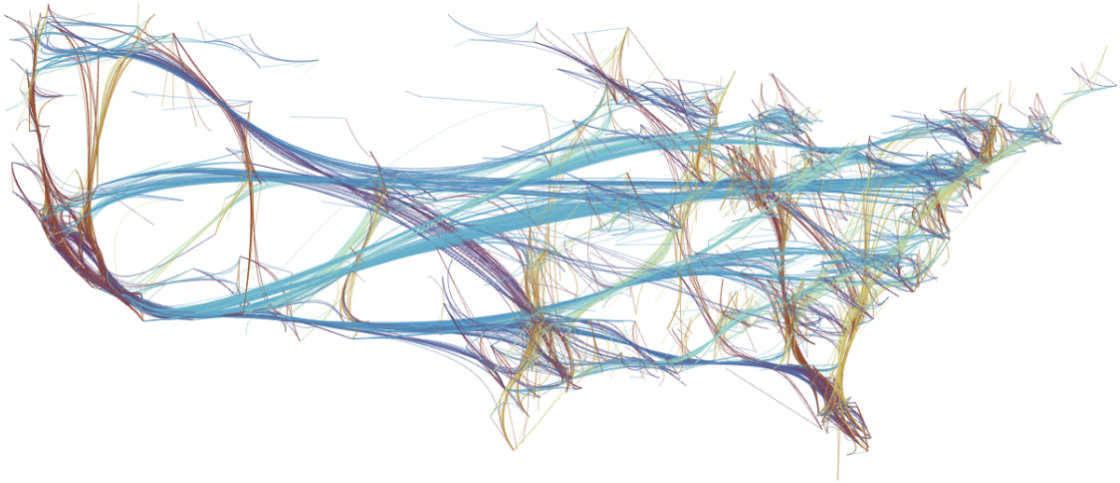


Figure 1.2: Visualization of a bundled graph.

The Edge-Path Bundling (EPB) algorithm introduced by Wallinger et al. [WAA+21] was developed to address this specific issue. It ensures that all bundled edges follow the shortest paths within the graph's network structure, resulting in a less ambiguous and more truthful representation. Despite its advantages in quality, the original EPB algorithm is computationally intensive, making it unsuitable for interactive applications. Wallinger at al. [WAA+23] introduced a significant performance improvement by using graph spanners. This S-EPB algorithm dramatically reduces computation time, claiming a 5-256 times speed up depending on the dataset.

Although this advancement makes S-EPB viable for static visualizations, it still fails to deliver the performance required for truly interactive applications, especially in a universally accessible environment like the web browser. True interactivity, where users adjust parameters and observe updates at real-time frame rates (e.g., >30 fps), demands per-frame computations under 33ms. The current CPU-based S-EPB implementation cannot reliably meet this target for large graphs.

To bridge this performance gap, this thesis proposes the design and implementation of a P-EPB algorithm based on the S-EPB algorithm, specifically tailored for execution on a GPU. By leveraging the massively parallel architecture of modern GPUs, the computationally expensive components of the algorithm can be executed simultaneously, offering the potential for a substantial performance increase.

This work utilizes WebGPU, the modern web standard [WEB25b] for GPU computing. WebGPU provides a lower-level, more efficient API that unlocks advanced capabilities, most notably compute shaders. This allows for general-purpose GPU computing directly within the browser, eliminating the need for complex workarounds and enabling the high-performance parallel processing required to accelerate the S-EPB algorithm [Sca24, SP24]. The primary goal is to achieve performance gains significant enough to enable interactive frame rates for large graph visualizations directly in a standard web browser.

This thesis seeks to answer the following research questions:

1. To what extent can the computationally expensive components of the S-EPB algorithm, such as spanner construction and shortest path searches, be effectively parallelized for a GPU architecture?
2. How does the performance of a WebGPU based implementation compare to the Central Processing Unit (CPU) based S-EPB implementation across graph datasets of varying size and density?
3. What are the primary challenges and architectural trade-offs involved in adapting and implementing graph algorithms for in browser GPU execution?

The main contributions of this thesis are:

1. The design of a parallel algorithm for S-EPB Bundling, specifically adapted for the massively parallel architecture of modern GPUs.
2. An open-source implementation of this algorithm using modern web technologies, namely JavaScript and the WebGPU API [WEB25a], made available as an interactive web application.
3. A comprehensive analysis and discussion of the performance, quality, strengths, weaknesses, and limitations of the proposed P-EPB parallel algorithm when compared to the baseline CPU-based S-EPB algorithm, supported by evaluations on multiple real-world datasets.

CHAPTER 2

Background and Related Work

2.1 Edge-Path Bundling

EPB is a technique introduced by Wallinger at al. [WAA+21] that reduces visual clutter while avoiding the independent edge ambiguities common in other bundling methods like Force-directed Edge Bundling (FDEB) or CUDA-based Universal Bundling (CUBu), which are described in section 2.3. These approaches often cluster edges based on geometric proximity and orientation, which can result in visually implying connections between structurally unrelated edges. EPB fundamentally avoids this by clustering independent edges. The algorithm bundles each edge along the existing weighted shortest path between its endpoints. Given a graph with a fixed layout, the algorithm iterates through edges, typically from longest to shortest. For each edge (u, v), it computes a shortest path in the graph between u and v that does not include the edge (u, v) itself. The pathfinding is weighted to penalize long detours from the straight-line drawing. If a path is found whose length is within a user-defined distortion threshold, the original edge is redrawn as a curve following the vertices of that path. By definition, this ensures that all visual bundles correspond to actual pathways in the graph, creating a more faithful representation and eliminating a key source of visual ambiguity.

2.2 Faster EPB with Graph Spanners (S-EPB)

To address the high computational cost of the original EPB algorithm, Wallinger et al. [WAA+23] later proposed an improved method, S-EPB. The primary bottleneck in the original algorithm is the repeated execution of Dijkstra's shortest path algorithm on the entire graph, leading to a practical runtime that scales poorly. S-EPB accelerates this process by fundamentally changing the graph on which pathfinding is performed.

The fundamental change of the S-EPB algorithm is the use of graph spanners. The algorithm first constructs a sparse t-spanner H of the input graph G. A t-spanner is a

subgraph that preserves the approximate shortest path distances of the original graph, guaranteeing that the distance between any two vertices in H is at most t times their distance in G, where t is the stretch factor [PS89]. The edges from the original graph G that are not included in the spanner H become the candidates for bundling. The algorithm then searches for bundling paths for these candidate edges, but crucially, it confines the search to the much sparser spanner H. This significantly reduces the number of edges that need to be explored during each shortest path computation, resulting in a substantial speed-up of 5 to 256 times in practice. While the resulting bundling is not identical to the original, it maintains a comparable visual quality and, most importantly, upholds the same guarantee of avoiding independent edge ambiguities.

2.3 Alternative edge bundling techniques

Besides the path-based approach of EPB and S-EPB, other techniques exist that bundle edges based either on geometric properties or use an image based approach. rather than the underlying graph topology. A common drawback to these approaches is that they group edges based on visual proximity alone, which can create misleading bundles that do not represent actual structural paths in the data.

2.3.1 Force-directed Edge Bundling

A prominent geometric bundling techniques is FDEB, proposed by Holten and van Wijk [HVW09]. This method treats edges as flexible springs in a self-organizing system. It works by subdividing each edge into a series of points and then applying attractive forces between corresponding points on nearby edges, pulling them together into bundles. To prevent the bundling of unrelated edges, the attraction strength is weighted by compatibility metrics that consider factors like angular similarity, length, and proximity, allowing for more nuanced control over the bundling process.

The primary difference from EPB is that FDEB is purely geometry-based, bundling edges that are close to each other in the visualization regardless of their structural relationship in the graph. This can introduce independent edge ambiguities where visually bundled edges do not represent actual paths, a problem EPB is specifically designed to avoid. While FDEB excels at simplifying the visual representation based on layout geometry, EPB prioritizes topological faithfulness by ensuring all bundles correspond to valid paths. Furthermore, FDEB's iterative, force-based simulation can be computationally intensive. However GPU-based implementations like the one proposed by Delu et al. [ZWGC12] exist, speeding up the algorithm depending on the edge count up to 11 times.

2.3.2 Winding Roads

The Winding Roads edge bundling technique proposed by Lambert et al. [LBA10], is a geometry-based edge bundling technique that reduces visual clutter by routing edges along a grid structure. The method discretizes the drawing space into a grid, often using

a hybrid of quad-trees and Voronoï diagrams to adapt to varying node densities. Edges are then rerouted along the shortest paths on this grid. To encourage bundling, grid segments that are part of many shortest paths have their weights reduced, attracting more edges to follow these common routes.

This approach is effective at reducing edge crossings and can be configured to avoid node-edge overlaps, resulting in a cleaner and more readable layout. However, a significant drawback of the Winding Roads technique lies in its bundling primitive. Because edges are bundled based on geometric proximity on an auxiliary grid rather than the underlying graph structure, it can group edges that are not structurally related. This can create the same independent edge ambiguities found in FDEB. Furthermore, according to [WAA+23], this approach is also slower compared to S-EPB.

2.3.3 CUBu

CUBu is a high-performance, alternative approach that operates in image-space rather than on the graph's structure. Introduced by van der Zwan et al. [VDZCT16], the technique calculates a density field from the 2D edge layout and iteratively pulls edges toward denser areas, causing them to form bundles. As a fully GPU-based algorithm, its primary advantage is exceptional speed, enabling the real-time bundling of graphs with millions of edges. However, similar to FDEB and Winding Roads, this approach can introduce independent edge ambiguities.

2.3.4 Pixel-Based Edge Bundling (PBEB)

PBEB is a modern framework proposed by Wu et al. [WSX+23] designed to visualize large graphs effectively on web-based platforms. It utilizes an image-based approach, employing Kernel Density Estimation (KDE) to compute edge similarities on a pixel grid. By leveraging the parallel processing capabilities of GPUs through WebGL, PBEB's performance is tied to image resolution rather than the number of edges, making it highly scalable. This method excels at reducing visual clutter to reveal high-level patterns in complex data, and its web-native design ensures portability across devices. While powerful, the technique can produce visual artefacts that necessitate additional smoothing and resampling steps to refine the final output.

In comparison to geometry-based methods like FDEB, PBEB demonstrates a significant performance advantage. While FDEB operates by manipulating the control points of edge paths, PBEB's image-space calculations are inherently more suited for GPU parallelization. However, similar to the other discussed techniques, PBEB can introduce independent edge ambiguities.

CHAPTER 3

S-EPB Parallelization

In this chapter, both the EPB and the S-EPB algorithms are analyzed to identify potential parts that can be parallelized. After potential optimizations have been identified, different approaches for optimizing the algorithms are discussed, leading into the next chapter where these approaches are then implemented.

3.1 Code Analysis

As a starting point, both the EPB and the S-EPB algorithms were analyzed to find parts that could be adapted for parallel execution. According to Gebali [Geb11], for-loops and while-loops often signal potential parallelism. In both algorithms, the most expensive computations occur within such loops.

The original EPB algorithm iterates through all edges in descending order of weight. In each iteration, the shortest path between the start and end points of the current edge is computed using Dijkstra's algorithm. Initially, Dijkstra's algorithm is applied to the full graph, but with each iteration, bundled edges are removed. Since each iteration updates which edges are excluded, the order of execution is important, requiring the loop to be executed sequentially.

Algorithm 3.1 describes the S-EPB algorithm. To summarize, it takes a graph and two parameters: the maximum distortion threshold t and an edge weight parameter k as input. First, a t-spanner is constructed from the original graph. Next, each spanner edge e has its weight updated to $\|e\|^k$. Finally, every edge in the original graph, that is not in the spanner, is bundled if the shortest-path distance between its endpoints is at most $t \cdot \|e\|$.

The algorithm can be divided into two main steps: spanner construction and edge bundling. The algorithm uses the greedy algorithm 3.2, which, similar to the EPB algorithm, iterates over sorted edges and performs shortest-path calculations on a dynamic graph. This

means the loop must be executed sequentially. However, the bundling step, shown in Algorithm 3.1, does not have this constraint. In each iteration, the shortest path between two points is computed. The key difference, however, is that these shortest-path calculations are carried out on the precomputed spanner, which remains fixed.

Algorithm 3.1: Spanner-Edge-Path bundling algorithm [WAA+23]

Input: $Graph = (V, E)$, maximum distorition $t > 1$, edge weight parameter k
Output: Control points for an Edge-Path bundled drawing controlPoints

```
1  // Greedy t Spanner construction step 3.2
2  Spanner = (V, E') ← constructGreedyTSpanner(G, ||·||, t)
3  // Bundling step
4  for edge ∈ E' do
5  |  edge.weight ← ||e||^k
6  end
7  for edge = (start, end) ∈ E \ E' do
8  |  path ← shortestPath(Spanner, start, end)
9  |  if path.length ≤ t * ||e|| then
10 |  |  controlPoints(edge) ← path.getVertexCoordinates()
11 |  end
12 end
13 return controlPoints
```

To summarize these findings, both the EPB and S-EPB contain inherently sequential components. For the EPB algorithm, the entire process consists of a single sequential loop. The S-EPB algorithm is similar, but it has one key difference: if there is an efficient way to calculate a t-spanner, the bundling step can be parallelized. Therefore, the S-EPB is chosen as the basis for further optimizations.

3.2 Parallel Algorithms

Several approaches could speed up the runtime, such as replacing Dijkstra's algorithm with a faster shortest-path method. Because this thesis focuses on taking advantage of GPU parallelism, it instead aims to make each loop iteration independent for parallel execution or to move expensive computations like the spanner construction or the edge bundling, out of loops so they, too, can run in parallel.

3.2.1 GPU Computing for Shortest Path Algorithms

Initially, Dijkstra's algorithm was considered for the shortest path calculations, as it is a standard and efficient method often used CPU-based bundling implementations. However, an early GPU-based prototype of Dijkstra's algorithm proved to be a significant performance bottleneck. The algorithm's greedy and inherently sequential nature poses a challenge for parallelization. At each step, it must identify the unvisited node with the

minimum current distance, which would require a synchronization across all processing threads. This dependency prevents the kind of massive, independent computation at which GPUs excel, leading to suboptimal performance as many cores remain idle while waiting for the next node to be selected.

Given the limitations of a parallel Dijkstra implementation, the Floyd-Warshall [Tor23, DTC+14] algorithm was chosen as a more suitable alternative for the GPU architecture. Although its theoretical time complexity of $O(|V|^3)$ is higher than the repeated Dijkstra runs on sparse graphs, its matrix-based structure is exceptionally well-suited for parallel execution. The algorithm's iterative process can be mapped directly to compute shaders, allowing the workload to be distributed across thousands of GPU cores to be processed simultaneously. Consequently, the practical runtime is significantly reduced, and more importantly, it scales with the number of vertices ($|V|$) rather than the number of edges ($|E|$). This makes the implementation's performance predictable and particularly effective for dense graphs, which can be processed in nearly the same amount of time as sparse graphs with an equivalent number of nodes.

3.3 Adapting S-EPB for Parallel Execution

3.3.1 Parallel t-spanner construction

The S-EPB algorithm uses the greedy spanner algorithm of Althöfer et al.[ADD+93] depicted in Algorithm 3.2. However, this algorithm is inherently sequential and thus difficult to parallelize without drawbacks or resorting to approximations. Furthermore, the S-EPB algorithm depends on the particular t-spanner edge set that the greedy algorithm produces. This means that different spanner constructions can create different bundling results. Therefore, the following two options seem reasonable:

1. Optimize the greedy spanner construction.
2. Find an alternative t-spanner algorithm that produces comparable bundling quality.

Some time was invested in trying to improve the greedy spanner construction, but this did not yield significant results. Therefore, option 2 was chosen for this thesis.

Naive Greedy Spanner

The algorithm 3.2 constructs a t-spanner from a weighted graph. It begins by initializing the spanner with all vertices of G but without edges. The edges of G are then processed in non-decreasing order of weight.

For each (u, v) with weight w, the algorithm checks if the shortest path distance between u and v in the current spanner is greater than $t \cdot w$. If this (condition, shortestPath$(S, u, v) > t \cdot w$), holds, the edge is added to the spanner. This iterative process guarantees that the final subgraph not only satisfies the t-spanner property but also only contains edges that were in the original graph.

Algorithm 3.2: Greedy t-spanner algorithm [ADD+93]

Input: Graph $G = (V, E)$, stretch factor $t > 1$
Output: t-spanner $S = (V, E' \subseteq E)$

```
1 S = (V, ∅)
2 sortedEdges ← sortAscendingByWeight(E)
3 for edge = (u, v) ∈ sortedEdges do
4 |   path ← shortestPath(S, u, v)
5 |   if path.length > t * edge.weight then
6 |   |   S.addEdge(edge)
7 |   end
8 end
9 return Spanner
```

Improved Greedy Spanner

Although the greedy algorithm in Algorithm 3.2 is inherently sequential, certain components can be accelerated. For instance, the shortest-path computations can be performed in parallel. As mentioned by Liang and Brent [LB96], one way to speed up the greedy spanner construction is to employ an all-pairs shortest-path algorithm instead of repeatedly invoking a single-source shortest-path algorithm. A prerequisite for this approach to be faster than its sequential counterpart is that at least $n^2/\log n$ processors are available, where n denotes the number of vertices in the graph.

At first glance, this strategy may seem counterintuitive, since computing all-pairs shortest paths after each edge insertion involves significantly more work than computing a single-source shortest path. However, in environments with massive parallelism, such as when running on a GPU, the additional work can be executed concurrently, potentially yielding overall time savings.

Theta Graph Spanner

Another approach to computing a t-spanner is to use a geometric spanner algorithm, such as the Theta-graph. Although a geometric spanner is fundamentally different from a greedy spanner, both methods can guarantee a spanner with stretch factor t.

A geometric spanner is constructed over a set of points by adding a minimal set of edges such that, for any two points, the graph distance is at most a constant factor times their Euclidean distance. In the Theta-graph construction, the plane around each point is partitioned into k equal-angle cones, and for each cone, an edge is added to the closest point within that cone [GNS08, VR14]. An example of such a cone is shown in Figure 3.2. The red point in the center is connected to the closest point in each of its cones, and this process is repeated for all vertices in the graph.

One drawback of using a geometric spanner in the S-EPB algorithm is that the Theta-

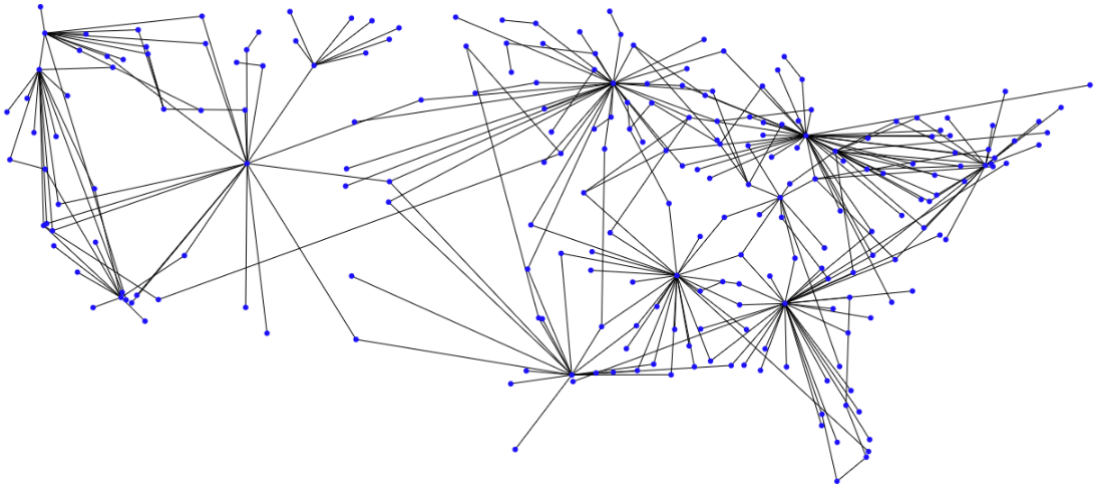


Figure 3.1: Visualization of a greedy graph spanner for the Airliens dataset 5.1.2.

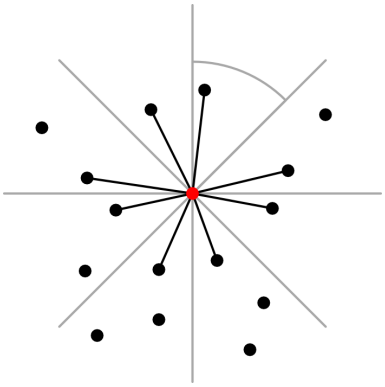


Figure 3.2: Visualization of a theta cone.

graph may introduce edges that are not present in the original graph, leading to ambiguity. Moreover, the overall structure of the resulting spanner differs between the greedy and geometric approaches, as seen in Figures 3.1 and 3.3.

3.3.2 Parallel edge bundling

The edge-bundling phase of the algorithm, illustrated in Algorithm 3.1, relies on single-source shortest-path calculations. It is called single-source because such methods calculate the shortest distance to every node in the graph, starting from a source node. One approach is to execute each iteration of the main loop in a separate thread, which makes the algorithm well-suited for parallelization and can deliver substantial performance improvements with relatively little additional effort. Another approach, similar to the method used in section 9, replaces single-source shortest-path computations with all-pairs shortest-path calculations. The difference from single-source shortest paths is that, in

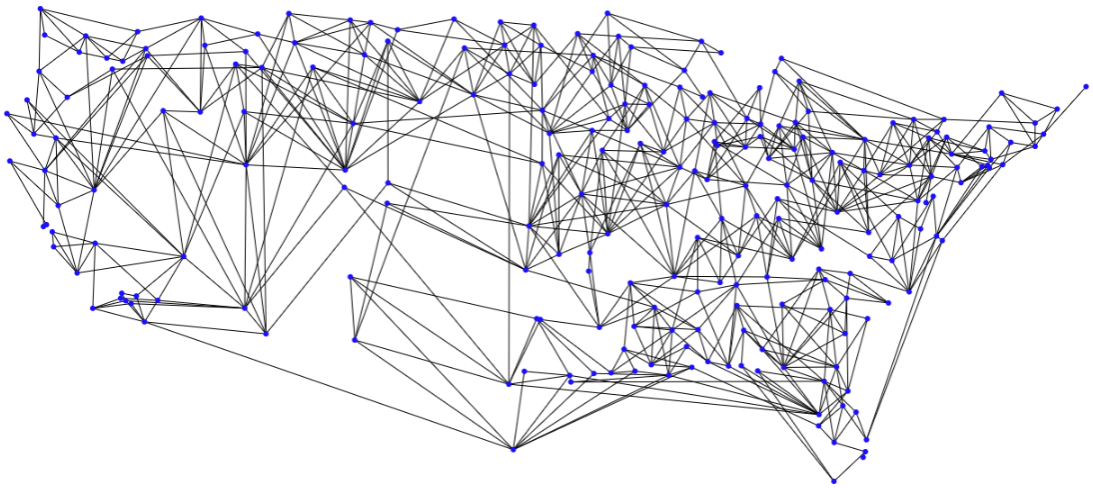


Figure 3.3: Visualization of a theta graph spanner for the Airlines dataset 5.1.2.

all-pairs shortest paths, the shortest distances between every pair of nodes in a graph are calculated. Each pair consists of a source node and a target node. In this case, all shortest paths are precomputed prior to the loop and subsequently retrieved during each iteration. In processing environments with sufficient computational resources, this precomputation strategy can lead to additional performance gains [LB96].

As discussed in Section 3.2.1, computing single-source shortest paths on the GPU proved to be inefficient. Therefore, since the Floyd–Warshall algorithm is well-suited for GPU computation, the all-pairs shortest path method was chosen for this thesis.

CHAPTER 4

Implementation

This chapter details the technical implementation of the P-EPB algorithm as a web application. The core of this implementation involves parallelizing the Floyd-Warshall algorithm on the GPU to compute all-pairs shortest paths efficiently. The chapter also describes the implementation of two spanner construction methods: an adapted greedy algorithm and a geometric Theta-graph algorithm. The P-EPB implementation and the online demo can be found in this public GitHub repository: `https://github.com/YannicEl/edge_bundling_webgpu`

4.1 Technology Stack

The technical implementation of this thesis is a web application built with the SvelteKit framework [SVE25] and written in TypeScript to ensure type safety and maintainable code. The development and build processes are managed by Vite [VIT25b], chosen for its fast performance and efficient module handling. At the heart of the visualization is the native WebGPU API [WEB25b], which provides direct, low-level control over graphics hardware for high performance. All compute and rendering shaders are written in WebGPU Shading Language (WGSL). The edge-bundling algorithms and the data parser were implemented from scratch. The user interface is styled using the utility-first css framework UnoCSS [UNO25].

4.2 System Architecture

4.2.1 Data Ingestion

To begin, the user chooses which graph to visualize. The web application presents several prepared datasets for easy access. The predefined datasets include those from the original paper, namely the airlines, airtraffic, and migration graphs. On the CPU side, the graph

is represented as an adjacency list. This representation is more memory efficient than an adjacency matrix because it only stores the actual edges rather than all possible edges. Furthermore, this enables fast, index-based node and edge lookups. A visual representation of such an adjacency can be seen in Figure 4.1.

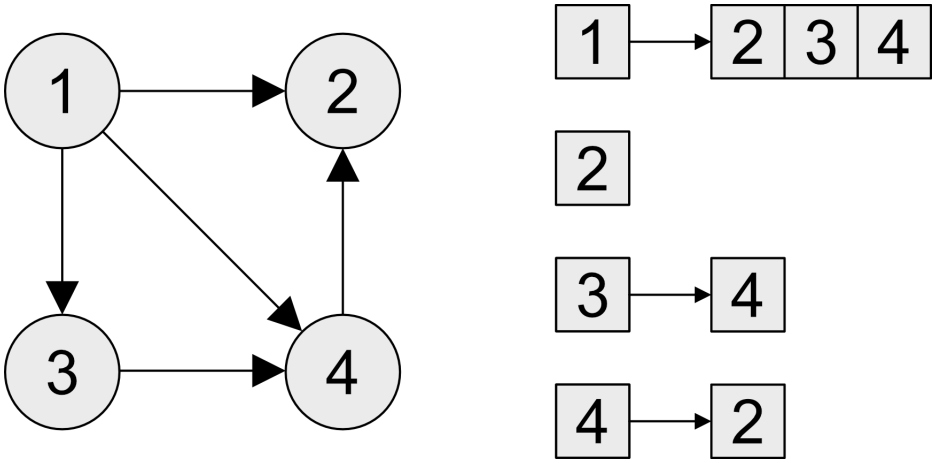


Figure 4.1: Visualization of a small graph (left) and its adjacency list (right).

4.2.2 WebGPU

WebGPU is a modern web API designed to be the successor to WebGL, offering high-performance access to a computer's GPU for both graphics rendering and general-purpose computation. Most relevant for this thesis is that WebGPU has support for compute shaders, which were not directly available with WebGL and enable general-purpose computations on the GPU directly in the browser.

4.3 Implementing the P-EPB algorithm

As mentioned earlier, the EPB algorithm is structured in a way that the method for calculating a t-spanner can be swapped out. In the following sections, the technical implementation of both the greedy and the spanner algorithms, as well as the edge bundling algorithm, are explained.

Starting from the JavaScript S-EPB algorithm, performance monitoring in the form of flame graphs was used to identify and optimize bottlenecks. These flame graphs can be generated within the development tools of most browsers and are a handy way to determine, down to the exact line of code, how much time is spent on computation. Figure 4.2 shows such a flame graph. Each block represents a function that is executed. Clicking on a block takes you to the source code, where each line is annotated with the number of milliseconds it took to run.

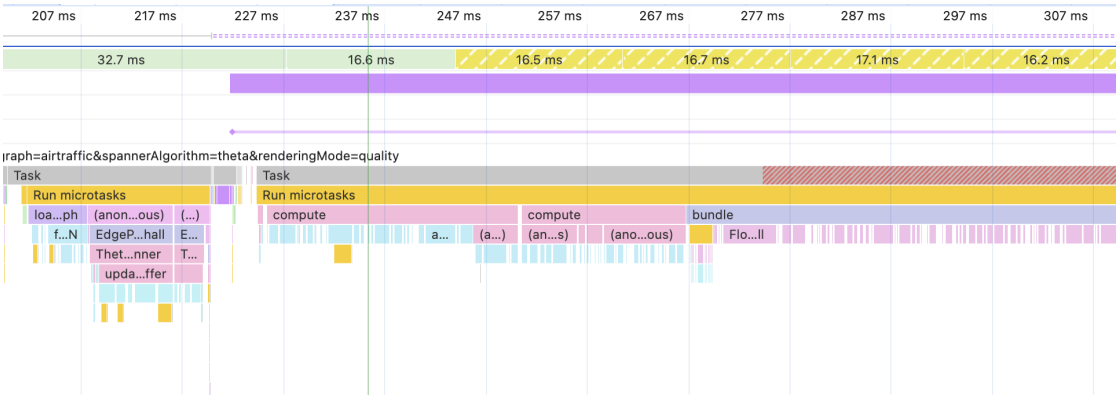


Figure 4.2: Visualization of a flame graph.

4.3.1 Floyd-Warshall

The Floyd–Warshall algorithm is used in both the greedy t-spanner algorithm and the edge-bundling step to compute shortest paths between all pairs of vertices in a graph. The graph is represented by an $n \times n$ adjacency matrix, where n is the number of nodes. The algorithm iterates over three nested loops, over indices k, x, and y, resulting in a time complexity of $O(n^3)$.

The outer loop runs on the CPU. For each k, n^2 compute shader threads are launched on the GPU, each responsible for a single matrix cell (x, y). Each thread compares the current distance d_{xy} with the path length $d_{xk} + d_{ky}$ via vertex k, updating d_{xy} if the latter is smaller. To enable path reconstruction, we maintain a second $n \times n$ matrix next$[i][j]$ that records, for every pair (i, j), the index of the next vertex on the shortest path. This matrix is initialized prior to the main loops and updated in tandem with the distance matrix within the compute shader.

4.3.2 Spanner construction

Both of the following spanner algorithms take the original graph as well as a stretch factor t as input parameters. The output is a t-spanner, which is then used in the edge bundling step.

Greedy Spanner

The greedy t-spanner algorithm employs an adapted version of the Floyd–Warshall algorithm. First, a $n \times n$ distance matrix is initialized and the graph's edges are sorted in descending order of weight. Then, for each edge $e = (u, v)$, n^2 compute shader threads are launched similarly to the outer loop in Algorithm 3.2. The distance of a cell is only updated if the spanner condition $d_{xy} > w_e \cdot t$ is true. Furthermore, whenever the condition is true, the edge e is added to the spanner.

Theta Spanner

The θ-graph algorithm builds a geometric spanner by partitioning the plane around each vertex into b equal cones with an angle of $\theta = 2\pi/b$. For each vertex u and each cone i, the distance ℓ of all vertices that lie within cone i is calculated and the vertex with the smallest ℓ is kept. If such a vertex v exists, the edge (u, v) gets added to the spanner. In practice, $n \times b$ compute shader threads, one for each pair (u, i), are launched. Each thread atomically tracks the minimum distances and the corresponding neighbor. The resulting graph is a t-spanner with:

$$t \;=\; \frac{1}{1 - 2\sin(\theta/2)}\,.$$

Because this approach can result in a spanner that includes edges not present in the original graph, an additional preprocessing step is applied to eliminate those edges. However, this may lead to situations where the resulting spanner is no longer a spanning tree. Conceptually, the edge bundling algorithm does not require a spanning tree to function. The potential drawbacks, such as increased ambiguity and a different graph structure compared to a greedy spanner, are discussed in Chapter 9.

4.3.3 Edge Bundling

The first step is to compute the set of edges present in the original graph but absent from the t-spanner. Next, the Floyd–Warshall algorithm is applied to the t-spanner to compute shortest paths between all pairs. For each edge (u, v) in this difference set, its precomputed shortest-path distance $d(u, v)$ is retrieved. Since these distances are already available, this lookup is efficient and can be done on the CPU. If

$$d(u, v) \;\leq\; t * w(u, v)$$

where t is the maximum-distortion parameter and $w(u, v)$ the original edge weight, the edge gets bundled. The coordinates of all vertices along that path then serve as control points for a Bézier curve.

4.3.4 Rendering Edges

For the sake of simplicity, once the edges are bundled and control points are calculated, they are rendered with the 2D Canvas API [CAN25]. Rendering the edges directly with WebGPU would be the obvious and faster solution, but at the same time, also be more complex. As the goal of this thesis was to speed up the edge bundling algorithm rather than the drawing, this compromise was taken.

In the UI one can choose between two rendering modes, fast and quality. The fast mode tries to approximate the bundled edge into multiple quadratic curves. Because the canvas API has a native method for drawing such curves, it is fast but also not as

accurate. The quality rendering mode, on the other hand, uses the same approach to how edges are drawn by Wallinger et al. [WAA+23]. It approximates the bundled edge as a Bezier curve with 50 segments. Depending on the dataset, calculating and drawing these segments is generally five to eight times slower compared to the fast rendering mode, but the end result is a much smoother curve. The quality rendering mode was used for all visualizations shown in this thesis.

CHAPTER 5

Evaluation and Results

The goal of this chapter is to compare the original S-EPB algorithm to the proposed P-EPB algorithm using various performance and quality metrics. Multiple variations of the P-EPB algorithm, each employing different implementations of the spanner construction, are evaluated against the fastest variation of the S-EPB algorithm without biconnected component decomposition

5.1 Experimental Setup

Experiments were conducted on a MacBook Air, equipped with an Apple M4 chip featuring a 10-core CPU and a 10-core GPU, along with 24 GB of unified memory and running the operating system version 15.5 Sequoia. It is important to note that the GPU core count on modern (post circa 2020) Apple machines cannot be directly compared to the thousands of CUDA cores typically found in dedicated GPUs from manufacturers like NVIDIA, as their architectures differ significantly. For both performance and quality, except for the interactivity experiment, the values for maximum distortion and the edge weight factor were chosen to be 2, as these values produced visually appealing output.

5.1.1 Performance Measures

The P-EPB algorithm was evaluated on Chromium v138 using the Vitest [VIT25a] framework with Playwright [PLA25] for browser integration. Each metric calculation was implemented as a separate test. Tests were executed sequentially, and all buffers were disposed of after each run. Vitest executes tests in a real browser, isolating each test in its own iframe to ensure independent browser contexts and prevent unintended resource sharing.

The C++ implementation of the S-EPB algorithm from Wallinger et al. [WAA+23] was used for the performance experiments because this version is tuned for performance.

Source code and compilation instructions are provided by Wallinger et al. in an open source repository [Wal21]. This implementation requires the OGDF library [OGD25]. OGDF version 2022.02 (Dogwood), the latest compatible release, was compiled according to the official documentation.

Furthermore, as a baseline, a JavaScript CPU implementation of the was also tested. This implementation, however, is very slow and does not include the same optimizations as the original version by Wallinger et al. [WAA+23]. Since the goal of this thesis is to run the S-EPB algorithm on the GPU, it was not considered necessary to optimize the JavaScript CPU implementation.

The following results, unless otherwise noted, are the average values obtained by running each test 25 times. To mitigate the potential impact of thermal throttling, the entire experiment suite, each test executed 25 times, was repeated three times consecutively, and the results from the final run are reported in the next section. The rationale behind this approach is that the system warms during prolonged testing, which could unfairly degrade the performance of tests that were run later in the suite. By selecting the potentially worst results from the last of the three runs, a fair comparison can be ensured.

Additionally, a two second pause was inserted between successive test runs to ensure that all allocated hardware resources were released and that each test began under identical initial conditions.

Quality Measures

All quality metrics were computed using the experimental suite provided in Wallinger et al. [WAA+23]. This approach ensures direct comparability of results and avoids errors that may arise from differences in metric implementations. The experimental suite is written in Python, and version 3.11.13 was used to produce the results. The Python implementation of the S-EPB algorithm was used because the C++ version does not support quality metric calculation. Apart from the programming language used and performance differences, both implementations should produce the same results. The performance difference, however, can be ignored for the quality experiments. The JavaScript implementation of the algorithm produces the exact same output as the P-EPB (greedy) variant. Therefore, the quality experiments were only run for the P-EPB (greedy) implementation.

5.1.2 Datasets

Table 5.1 shows the six datasets that were used for testing. For the sake of comparability, three of the six datasets were taken directly from Wallinger et al. [WAA+23] [WAA+21]. The other three datasets are synthetic and were not created to test visual clarity but to test how the new algorithm scales in dense graphs. These synthetic datasets are fully connected graphs with 256, 529 and 1024 nodes, respectively. Because of the limitation that the whole graph needs to be contained in a single WebGPU buffer, the two larger datasets, Amazon200k and PanamaPapers, with 192,976 and 743,253 vertices,

respectively, were not tested. The minimum buffer size supported by all WebGPU-enabled browsers is 256 MiB, as described in the WebGPU specification [WEB25b], which is rather small but large enough to fit graphs with up to around 8200 vertices. It is up to the operating system and the browser to determine this limit. For example, the maximum buffer size on the machine where the experiments are run, which is described in chapter 5.1, is around 4 GB. The implications of this limitation are discussed in chapter 5.5.

Dataset	$\|V\|$	$\|E\|$
Airlines	235	2101
Migrations	**1702**	9,726
Airtraffic	1533	16,494
FC 256	256	32,640
FC 529	529	139,656
FC 1024	1024	**523,776**

Table 5.1: All six datasets with their vertex and edge counts.

5.2 Performance Evaluation

5.2.1 Runtime Comparison

Table 5.2 presents the average runtime in milliseconds over 25 iterations, excluding the time spent rendering. Due to the long runtime of this implementation, the experiments were run only five times for the airlines dataset and once for the migration and airtraffic datasets. The FC256, FC529, and FC1024 datasets were not tested at all because their runtimes were excessively long. It is clear that the P-EPB algorithm with the greedy spanner implementation is slower across all datasets compared to the other two algorithms. This is likely due to the fact that both, the spanner construction and the edge bundling steps, involve a three-level nested loop, which increases computational complexity. On the other hand, the P-EPB algorithm with the theta spanner implementation produces comparable results for the airlines and migration datasets compared to the S-EPB algorithm. However, this implementation excels with dense graphs, such as the airtraffic dataset, where it achieves nearly a 2x speedup. Notably, the speedup factor appears to increase with the graph's density. For instance, the FC1024 dataset, which contains over half a million edges, runs more than seven times faster than the S-EPB algorithm. The naive S-EPB JavaScript implementation is orders of magnitude slower than any other implementation.

5.2.2 Interactivity

Because the P-EPB algorithm requires setting up GPU buffers and transferring data—operations that introduce latency—the time measured to recompute the bundling after each parameter change is measured. Depending on which parameter is modified, some buffers can be

Algorithm	Airlines	Migration	Airtraffic	FC 256	FC 529	FC 1024
S-EPB (Wallinger)	**16**	**292**	563	185	1250	6607
S-EPB (JavaScript)	15,8 sec	44 min	75 min	/	/	/
P-EPB (greedy)	116	1,224	1,236	1,262	5,664	24,388
P-EPB (theta)	38	348	**287**	**34**	**150**	**924**

Table 5.2: Average runtime, unless otherwise noted, in milliseconds.

reused, reducing the recomputation time. This metric is critical because the outcome of this thesis is an interactive web application in which users adjust parameters on the fly, making render latency far more important than in a scenario where just one visualization is generated.

Tables 5.4 and 5.3 show the runtime results after modifying the edge weight factor and maximum distortion parameter, respectively, each adjusted twice. Additionally, Table 5.5 presents the runtime results when both parameters are changed simultaneously.

From the tables, the general trend that initial computation time is typically longer than the subsequent re-computations can be observed. However, there are notable outliers. For instance, in the case of the P-EPB algorithm combined with the greedy spanner, applied to both the migration and airtraffic datasets, recalculations, particularly when the maximum distortion parameter is set to one, take longer than the initial computation.

	Airlines			Migration			Airtraffic		
Algorithm	init	dist_1	dist_2	initial	dist_1	dist_2	init	dist_1	dist_2
P-EPB (greedy)	120	86	84	1242	1341	463	1243	2812	807
P-EPB (theta)	36	12	15	346	35	51	288	43	54

Table 5.3: Interactivity, maximum distortion.

	Airlines			Migration			Airtraffic		
Algorithm	init	wght_1	wght_2	init	wght_1	wght_2	init	wght_1	wght_2
P-EPB (greedy)	115	13	6	1229	216	13	1241	170	23
P-EPB (theta)	38	17	7	348	214	19	285	165	28

Table 5.4: Interactivity, edge weight factor.

5.3 Quality Evaluation

The following section describes the quality metrics used. For the sake of comparability, ink reduction, distortion and ambiguity metrics are taken from [WAA+23].

	Airlines			Migration			Airtraffic		
Algorithm	initial	dist	weight	initial	dist	weight	initial	dist	weight
P-EPB (greedy)	117	94	87	1232	1536	466	1252	3000	819
P-EPB (theta)	37	24	14	346	234	53	286	182	55

Table 5.5: Interactivity, maximum distortion and edge weight factor.

5.3.1 Visual Comparison

Figure 5.1 shows the visual outputs of the P-EPB algorithm, both greedy and theta-spanner versions, alongside those of the S-EPB algorithm. The P-EPB images were produced with the web implementation, while the S-EPB images use the Python implementation by Wallinger et al. [WAA+23].

Both methods were run with the same parameters—maximum distortion = 2 and edge-weight factor = 2, yet the visualizations differ noticeably. In the S-EPB visualization, edges form much tighter bundles than in the P-EPB versions. A possible explanation for this is that the P-EPB algorithm implements edge path bundling incorrectly. However, since the JavaScript S-EPB implementation produces the same output while being fundamentally different in its code, this seems unlikely. The more probable reason for the differing visualizations is the way each implementation renders Bézier curves.

5.3.2 Ink Reduction

"Ink reduction" is a quality metric for edge bundling that measures how much less "ink" a bundled drawing uses compared to an unbundled one. To calculate it, the "active pixels" are counted, pixels whose color is above a specific gray threshold, in both drawings. The ink reduction R_{ink} is then given by the ratio

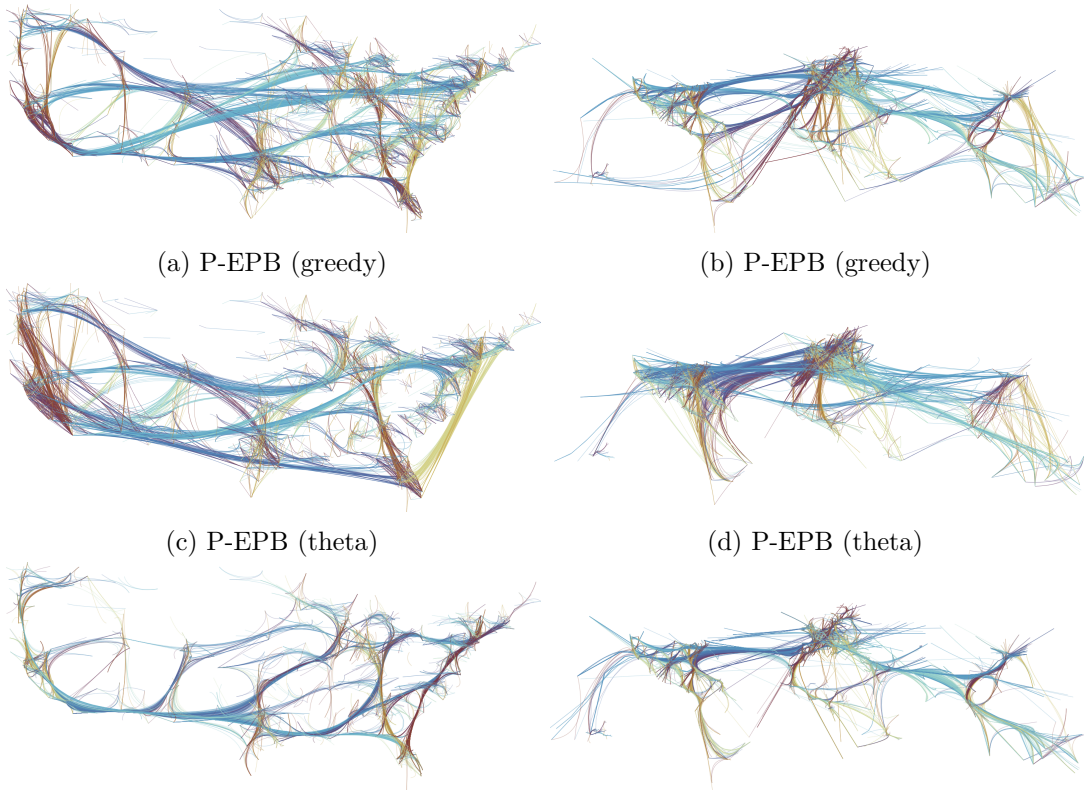
$$R_{\text{ink}} = \frac{P_b}{P_u}$$

where P_b is the number of active pixels in the bundled drawing and P_u is the number in the unbundled drawing. A smaller R_{ink} indicates better bundling, as it reflects less visual clutter.

The results in the Tables 5.6, 5.7, and 5.8 show that the P-EPB implementations generally achieved comparable or superior ink reduction versus the S-EPB algorithm, especially on denser datasets. However, on the airlines dataset, other methods like CUBu and Winding Roads resulted in the greatest reduction.

5.3.3 Distortion

The quality metric distortion quantifies how much edge-bundling elongates connections compared to their original straight-line distances. For each edge e, one measures the



(a) P-EPB (greedy) (b) P-EPB (greedy)

(c) P-EPB (theta) (d) P-EPB (theta)

(e) S-EPB (Wallinger) (f) S-EPB (Wallinger)

Figure 5.1: Comparing the different bundling algorithms. (a) (b) P-EPB (greedy), (c) (d) P-EPB (theta) and (e) (f) S-EPB (Wallinger).

bundled length $\ell_b(e)$ and the Euclidean straight-edge length $\ell_s(e)$, then computes the average ratio

$$D = \frac{1}{|E|} \sum_{e \in E} \frac{\ell_b(e)}{\ell_s(e)}$$

where $|E|$ is the total number of edges. A distortion value D close to 1 indicates that bundling has preserved the original geometry, whereas larger values reveal increasing detours and potential visual artefacts introduced by the bundling process.

The results in the Tables 5.6, 5.7, and 5.8 show that across most datasets, the P-EPB algorithms produced distortion values similar to S-EPB and other methods. For the migration dataset, the CUBu and Winding Roads algorithms yielded the lowest distortion.

5.3.4 Ambiguity

The ambiguity of a bundled layout measures the proportion of false neighbors to all visually implied neighbors in the graph, capturing how many incorrect adjacencies are perceived due to ambiguous edge renderings. For each edge $e = (u, v)$, reachable neighbor sets $N(u, e)$ are defined based on intersections and flat angles between edges. True neighbors ($N_t(u, e)$) are those within a graph distance threshold δ, while false neighbors ($N_f(u, e)$) are the remaining vertices. The ambiguity metric calculates the ratio of false neighbors to total neighbors, with lower values indicating less ambiguity and therefore being better. Theoretically, a value of 0 would be ideal, but even a straight-line drawing—as shown in Tables 5.6, 5.7, and 5.8—has an ambiguity greater than 0. In the tables, the ambiguity is denoted as amb_n, where n is the value of δ. The P-EPB (theta) version often achieved the lowest ambiguity, significantly outperforming others on the airtraffic dataset. On the migration dataset, traditional algorithms like Winding Roads and Force-directed Edge Bundling produced better ambiguity scores.

5.3.5 Summary

The Tables 5.6, 5.7, and 5.8 show the quality metrics for the airlines, migration, and airtraffic datasets. Furthermore, as a lower bound comparison quality metrics for the CUBu, Winding Roads and FDEB algorithms were taken from Wallinger et al. [WAA+21]. Overall, both P-EPB implementations and the S-EPB algorithm yield comparable results.

Notably, for the airtraffic dataset, both P-EPB implementations achieve better ink reduction, and the theta-spanner implementation even has a smaller ambiguity value. However, for the airlines dataset, the P-EPB theta-spanner implementation has a noticeably worse ink reduction score.

When comparing the edge path bundling results to traditional algorithms, one might expect the traditional methods to yield higher ambiguity values. This is generally true, except in the migration dataset, where the Winding Roads and FDEB algorithms achieve better ambiguity scores. In the airlines dataset, the CUBu and Winding Roads algorithms show the greatest ink reduction compared to other methods. Finally, for the migration dataset, CUBu and Winding Roads yield the lowest distortion values. Interestingly, the CUBu algorithm has a distortion value of 1.00 for the airtraffic dataset, which is the same as a straight-line drawing of the graph.

5.4 Discussion

The evaluation results highlight the potential of WebGPU to enhance the performance of complex graph visualization algorithms, such as edge path bundling. The proposed P-EPB algorithm, particularly the theta-spanner variant, demonstrated promising results, significantly improving upon the CPU-based S-EPB implementation on dense graphs. This section explores these findings, delves into the factors contributing to the observed

Algorithm	ink	dist	amb_1	amb_2
Straight	1.00	1.00	0.66	**0.03**
S-EPB (Wallinger)	0.58	1.08	0.80	0.05
P-EPB (greedy)	0.59	1.13	0.77	0.04
P-EPB (theta)	0.70	**1.02**	**0.62**	0.04
CUBu	**0.47**	1.08	0.86	0.05
Winding Roads	**0.47**	1.08	0.86	0.05
Force-directed	0.77	1.04	0.73	0.04

Table 5.6: Quality metric comparison Airlines.

Algorithm	ink	dist	amb_1	amb_2	amb_3	amb_4	amb_5
Straight	1.00	1.00	0.67	0.39	0.13	0.05	0.038
S-EPB (Wallinger)	0.59	1.07	0.69	0.34	0.10	0.05	0.03
P-EPB (greedy)	**0.51**	1.13	0.77	**0.04**	**0.00**	**0.00**	**0.00**
P-EPB (theta)	0.56	1.21	0.46	0.25	0.09	0.06	0.05
CUBu	0.64	**1.06**	0.77	0.43	0.15	0.06	0.04
Winding Roads	0.58	**1.06**	**0.42**	0.15	0.07	0.06	0.05
Force-directed	0.77	1.12	0.44	0.15	0.07	0.06	0.05

Table 5.7: Quality metric comparison Migration.

Algorithm	ink	dist	amb_1	amb_2	
Straight	1.00	1.00	0.54	0.10	**0.00**
S-EPB (Wallinger)	0.63	1.10	0.58	0.14	**0.00**
P-EPB (greedy)	**0.44**	1.13	0.77	**0.04**	**0.00**
P-EPB (theta)	**0.44**	1.10	**0.35**	0.07	**0.00**
CUBu	0.71	**1.00**	0.59	0.11	0.01
Winding Roads	0.56	1.06	0.58	0.12	0.01
Force-directed	0.79	1.06	0.54	0.10	**0.00**

Table 5.8: Quality metric comparison Airtraffic.

performance gains, identifies areas where bottlenecks persist, and reflects on the trade-offs involved in the chosen approach.

The success of the P-EPB (theta) algorithm can largely be attributed to its effective utilization of the GPU's massively parallel architecture. The decision to use the Floyd-Warshall algorithm for all pairs shortest path computation played a key role. While its $O(|V|^3)$ time complexity is theoretically high, its structure lends itself well to GPU execution. Each of the n^2 distance updates in the innermost loops can be processed independently by separate GPU threads. This turned a complex task into a data-parallel problem, using thousands of GPU cores to process the graph's adjacency matrix at the

same time. By comparison, the CPU-based S-EPB implementation relies on sequential single-source shortest path calculations using Dijkstra's algorithm.

The choice of spanner construction also played a pivotal role. The greedy spanner algorithm, while effective, is inherently sequential, as each step depends on the results of the previous one, making it difficult to parallelize. This limitation is reflected in the relatively poor performance of the P-EPB (greedy) variant shown in Table 5.2. In contrast, the theta-graph spanner is a geometric construction that can be built in a highly parallel manner, making it a better fit for GPU execution and a key factor in the observed performance improvements.

An interesting finding is that, for the migration dataset, the Winding Roads and algorithms achieve the best and second-best ambiguity scores for $\delta = 1$, outperforming the S-EPB algorithm. Overall, in terms of ambiguity, the non edge path and edgepath bundling algorithms perform very similarly. In some cases, the non edge path bundling algorithm even achieves a better ambiguity score.

A glaring shortcoming of the current P-EPB algorithm is that the visual output differs significantly from the visualization generated by the original S-EPB algorithm. This is likely due to differences in how both implementations render Bézier curves. Unfortunately, this means that the P-EPB algorithm cannot act as a drop-in replacement for the S-EPB algorithm in situations where visual output similarity is important. On the upside, this does not invalidate the technical improvements made to the algorithm, as rendering performance was not factored into the performance measures of either this thesis or the paper by Wallinger et al. [WAA+23]. Theoretically, once this visual discrepancy is resolved, the result would be a comparably fast edge path bundling algorithm on sparse graphs and a multiple-times-faster algorithm on dense graphs that runs entirely in the browser.

5.5 Remaining Bottlenecks and Limitations

Despite the notable speedup, several bottlenecks remain that constrain performance, particularly during the initial computation phase:

1. **CPU-GPU Data Transfer:** Before computation begins on the GPU, the graph data must be formatted into matrices and transferred from CPU memory to GPU buffers. While this is a preprocessing step, it represents a significant portion of the initial load time, as evidenced by the differences between initial and subsequent runtimes in Tables 5.4 5.3 5.5.

2. **Sequential CPU-side Logic:** The implementation is not fully GPU-based. The main loop of the Floyd-Warshall algorithm is still managed by the CPU, which dispatches a new compute shader for each of the n iterations. Additionally, after the shortest paths are computed, the resulting matrix is read back to the CPU, where

the final path reconstruction and bundling logic are executed. This round-trip and CPU-side processing introduce overhead.

3. **Rendering:** The final rendering of the Bézier curves is handled by the 2D Canvas API. While this approach is convenient, it is not the most performant. A full WebGPU rendering pipeline could potentially improve performance, but would require significant additional implementation effort.

4. **Processing large graphs:** Currently, processing larger graphs is constrained by the requirement that the entire graph must fit into a single WebGPU buffer, whose maximum size depends on the system running the program [WEB25b]. The minimum buffer size supported by all WebGPU-enabled browsers is 256 MiB, which is sufficient for processing a graph with up to around 8,200 vertices. However, this limit can be higher on certain machines. For example, on the machines used to run the experiments described in Section 5.1, the buffer size could be increased to around 4 GiB. Depending on the use case, this limitation can become problematic. For instance, the datasets *Amazon200k* and *PanamaPapers* from Wallinger et al. [WAA+23] each contain over 100,000 vertices. Furthermore, in other fields, graphs with millions of vertices and edges, such as the social graphs studied in Zhao et al. [ZLK+25], are not uncommon. For such large-scale graphs, the current P-EPB implementation is not suitable.

CHAPTER 6

Conclusion and Future Work

In conclusion, this thesis showed that the web based P-EPB algorithm achieves comparable results in both performance and quality compared to the highly optimized C++ implementation of the S-EPB algorithm. However, even though the quality metrics of both algorithms, when tested on the same dataset, are comparable, the visual output is quite different.

1. **To what extent can the computationally expensive components of the S-EPB algorithm, such as spanner construction and shortest path searches, be effectively parallelized for a GPU architecture?**

   The core components of the S-EPB algorithm can be effectively parallelized, but it requires substituting the original algorithms with alternatives better suited for a GPU architecture. The thesis demonstrates that the inherently sequential Dijkstra's algorithm for shortest-path calculation can be successfully replaced by the Floyd-Warshall algorithm, which is highly parallelizable. Similarly, while the original greedy spanner construction performs poorly on a GPU due to its sequential dependencies, an alternative geometric method like the theta-graph spanner can be constructed in a highly parallel manner, enabling significant performance gains.

2. **How does the performance of a WebGPU-based implementation compare to the CPU based S-EPB implementation across graph datasets of varying size and density?**

   This thesis showed that, in terms of performance, both the S-EPB and the P-EPB algorithms produce similar results on smaller graphs such as the Airlines and Migration datasets. Although the P-EPB algorithm performed slightly worse on these datasets, the performance gap was not significant. Where the P-EPB algorithm truly shines is with dense datasets. From the Airtraffic dataset up to the FC1024 dataset, the P-EPB algorithm consistently outperforms the S-EPB,

achieving a speedup of more than eight on the FC529 dataset. The goal of achieving computation times as low as 33ms, however, was not met. No silver bullet for eliminating the many inherently sequential parts of the EPB and S-EPB algorithms was found.

3. **What are the primary challenges and architectural trade-offs involved in adapting and implementing graph algorithms for in-browser GPU execution?**

   Key challenges stem from data management and architectural constraints. The initial overhead of transferring graph data from the CPU to the GPU is significant. Furthermore, the need for the entire graph to fit within a single CPU memory buffer limits the maximum graph size. Finally, many established algorithms like Dijkstra's are inherently sequential and ill-suited for GPU parallelization, which restricts the available choices.

There is still plenty of room to speed up the P-EPB algorithm as discussed in 5.5. Some parts of the algorithm still run on the CPU. Moving them all to the GPU should give a big speed boost by cutting extra data transfers and synchronization between CPU and GPU.

Another interesting topic is handling graphs of arbitrary size. To do this, the algorithm must process the graph in smaller chunks rather than all at once. This would overcome the WebGPU buffer size limit, though it may introduce new challenges.

WebGPU only became available in most browsers in 2025, so it's still very new. Shared graph processing libraries for WebGPU are needed so experiments like the ones in this thesis are easier to run. Everything from memory management to basic algorithms like Dijkstra had to be implemented from scratch, which takes a lot of time away from focusing on the actual algorithmic problems. However, all code from this thesis is open source.

WebGPU is a powerful new technology that has the potential to enable new ways of interactive data visualization on the most portable and shareable platform in existence, the Web.

Overview of Generative AI Tools Used

The used generative AI tools can be categorized into three groups:

1. **Research:** Various LLMs from OpenAI and Google, such as o4 and Gemini 2.5 Pro, were used initially to gain a broad understanding of the field and identify potential algorithms and data structures. For more detailed literature research, the tool Consensus [CON25] was used.
2. **Implementation:** The Integrated development environment (IDE) Cursor [CUR25] was used in two ways. First, as a general AI-assisted autocompletion tool and second, for more mundane tasks such as blueprinting page layouts or generating shader boilerplate.
3. **Writing:** Finally, o4 and Gemini 2.5 Pro were used as writing assistants to proofread text and improve sentence structure.

Acronyms

CPU Central Processing Unit. xi, 3, 10, 15, 17, 18, 22, 27, 29–32

CUBu CUDA-based Universal Bundling. 5, 7, 25–27

EPB Edge-Path Bundling. 2, 5, 6, 9, 10, 16, 32

FDEB Force-directed Edge Bundling. 5–7, 27

GPU Graphics Processing Unit. xi, xiii, 2, 3, 6, 7, 10–12, 14–17, 21–23, 28, 29, 31, 32

IDE Integrated development environment. 33

P-EPB Parallel Edge-Path Bundling. xi, xiii, xv, 2, 3, 15–17, 19, 21–32

PBEB Pixel-Based Edge Bundling. 7

S-EPB Spanner-based Edge-Path Bundling. xi, xiii, xv, 2, 3, 5–7, 9–14, 16, 21–23, 25–27, 29, 31, 32

WGSL WebGPU Shading Language. 15

Bibliography

[ADD+93] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9(1):81–100, January 1993.

[BRH+17] Benjamin Bach, Nathalie Henry Riche, Christophe Hurter, Kim Marriott, and Tim Dwyer. Towards Unambiguous Edge Bundling: Investigating Confluent Drawings for Network Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):541–550, January 2017.

[CAN25] Canvas API. `https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D`, 2025.

[CON25] Search - Consensus: AI Search Engine for Research. `https://consensus.app/search/`, 2025. Accessed: 2025-08-04.

[CUR25] Cursor - The AI Code Editor. `https://cursor.com/en`, 2025. Accessed: 2025-08-04.

[DTC+14] Hristo Djidjev, Sunil Thulasidasan, Guillaume Chapuis, Rumen Andonov, and Dominique Lavenier. Efficient Multi-GPU Computation of All-Pairs Shortest Paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 360–369, Phoenix, AZ, USA, May 2014. IEEE.

[Geb11] Fayez Gebali. *Algorithms and Parallel Computing*. John Wiley & Sons, March 2011. Google-Books-ID: 3g6lrxrd4wsC.

[GNS08] Joachim Gudmundsson, Giri Narasimhan, and Michiel Smid. Geometric Spanners: 2002; Gudmundsson, Levcopoulos, Narasimhan. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 360–364. Springer US, Boston, MA, 2008.

[HVW09] Danny Holten and Jarke J. Van Wijk. Force-Directed Edge Bundling for Graph Visualization. *Computer Graphics Forum*, 28(3):983–990, June 2009.

[LB96] Weifa Liang and Richard P Brent. Constructing the spanners of graphs in parallel. In *Proceedings of International Conference on Parallel Processing*, pages 206–210, 1996.

[LBA10] A. Lambert, R. Bourqui, and D. Auber. Winding Roads: Routing edges into bundles. *Computer Graphics Forum*, 29(3):853–862, June 2010.

[OGD25] Open Graph Drawing Framework. `https://github.com/ogdf/ogdf`, 2025. Accessed: 2025-07-27.

[PLA25] Playwright. `https://playwright.dev/`, 2025. Accessed: 2025-08-04.

[PS89] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, March 1989.

[Sca24] Matthew Scarpino. *The WebGPU Sourcebook: High-Performance Graphics and Machine Learning in the Browser*. CRC Press, Boca Raton, 1 edition, August 2024.

[SP24] S. Sarathi and P. S. Varshiga. Unleashing the Power of Graphics: A Comprehensive Exploration of WebGPU. *International Research Journal on Advanced Engineering and Management (IRJAEM)*, 2(05):1785–1791, May 2024.

[SVE25] Svelte • Web development for the rest of us. `https://svelte.dev/`, 2025. Accessed: 2025-07-13.

[Tor23] Ismail H. Toroslu. The Floyd-Warshall all-pairs shortest paths algorithm for disconnected and very sparse graphs. *Software: Practice and Experience*, 53(6):1287–1303, June 2023.

[UNO25] UnoCSS. `https://unocss.dev/`, 2025. Accessed: 2025-07-13.

[VDZCT16] Matthew Van Der Zwan, Valeriu Codreanu, and Alexandru Telea. CUBu: Universal Real-Time Bundling for Large Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 22(12):2550–2563, December 2016.

[VIT25a] Playwright. `https://vitest.dev/`, 2025. Accessed: 2025-08-06.

[VIT25b] Vite. `https://vite.dev`, 2025. Accessed: 2025-07-13.

[VLKS+11] T. Von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J.J. Van Wijk, J.-D. Fekete, and D.W. Fellner. Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges. *Computer Graphics Forum*, 30(6):1719–1749, September 2011.

[VR14] Andre Van Renssen. *Theta-Graphs and Other Constrained Spanners*. Doctor of Philosophy, Carleton University, Ottawa, Ontario, 2014.

[WAA+21] Markus Wallinger, Daniel Archambault, David Auber, Martin Nöllenburg, and Jaakko Peltonen. Edge-Path Bundling: A Less Ambiguous Edge Bundling Approach, August 2021. arXiv:2108.05467.

[WAA+23] Markus Wallinger, Daniel Archambault, David Auber, Martin Nöllenburg, and Jaakko Peltonen. Faster Edge-Path Bundling through Graph Spanners. *Computer Graphics Forum*, 42(6):e14789, September 2023.

[Wal21] Markus Wallinger. Edge-Path Bundling. `https://osf.io/t4h6j/`, August 2021.

[WEB25a] WebGPU API. `https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API`, 2025. Accessed: 2025-07-13.

[WEB25b] WebGPU Specification. `https://www.w3.org/TR/webgpu/`, 2025. Accessed: 2025-08-06.

[WSX+23] Jieting Wu, Jianxin Sun, Xinyan Xie, Tian Gao, Yu Pan, and Hongfeng Yu. Accelerating Web-based Graph Visualization with Pixel-Based Edge Bundling. In *2023 IEEE International Conference on Big Data (BigData)*, pages 6005–6014, Sorrento, Italy, December 2023. IEEE.

[ZLK+25] Tong Zhao, Yozen Liu, Matthew Kolodner, Kyle Montemayor, Elham Ghazizadeh, Ankit Batra, Zihao Fan, Xiaobin Gao, Xuan Guo, Jiwen Ren, Serim Park, Peicheng Yu, Jun Yu, Shubham Vij, and Neil Shah. GiGL: Large-Scale Graph Neural Networks at Snapchat, 2025.

[ZWGC12] Delu Zhu, Kaichao Wu, Danhuai Guo, and Yuanmin Chen. Parallelized force-directed edge bundling on the gpu. In *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science*, pages 52–56, 2012.