Optimizing a GPU-Based All-Pairs Shortest Path Algorithm Through Partitioning and Transfer Reduction

Tomer Haham

May 16, 2025

3692313

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Adil Chhabra

Acknowledgments

First, I would like to sincerely thank Prof. Dr. Christian Schulz for the opportunity to work on this project under his supervision. His support throughout various projects during my studies has been greatly appreciated. I am also grateful to Adil Chhabra for his valuable assistance and advice throughout the course of this thesis. His feedback and availability were especially helpful in addressing questions and overcoming challenges along the way.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, May 16, 2025

Tomer Haham

Abstract

The All-Pairs Shortest Path (APSP) problem is a fundamental challenge in graph theory with extensive applications in various domains. FASTAPSP is a state-of-the-art GPU-accelerated algorithm designed to efficiently solve APSP on large-scale graphs. However, its performance is limited by the quality of graph partitioning and excessive CPU–GPU data transfers. In this thesis, we introduce two complementary enhancements to the FASTAPSP algorithm. First, we optimize the GPU execution pipeline by restructuring data transformations into GPU kernels and minimizing costly CPU–GPU data transfers. This architectural optimization yields substantial performance improvements, with an average speedup of 2.41× and peak gains of up to 6.7× on the largest graphs. Second, we evaluate and integrate advanced graph partitioners, including KAHIP and the GPU-based JET partitioner, to improve partition quality and reduce computational overhead. This leads to an additional 2.7% runtime improvement when using the KAHIP social variant compared to the original implementation with the default METIS partitioner.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The All-Pairs Shortest Path (APSP) problem is a fundamental challenge in graph theory with widespread applications across a variety of domains. It plays a crucial role in transportation and navigation systems, where shortest path distances between all intersections or cities are used to precompute efficient routing schemes [18]. In social and web network analysis, APSP enables the computation of centrality measures such as betweenness centrality, which depends on the number of shortest paths that pass through each node [3]. In biological network modeling, APSP-based methods have been applied to identify signaling pathways in protein-protein interaction networks by finding optimal paths between functional regions [47]. In web search and information retrieval, APSP-style computations support the evaluation of structural similarity between documents or entities across large-scale information networks [56]. Likewise, in parallel and distributed computing, APSP workloads, particularly those used in centrality analysis, serve as benchmarks and algorithmic foundations for graph processing on modern multicore and multithreaded architectures [35]. Many of these applications require computing shortest paths not just between a few vertex pairs but between all pairs of vertices in large, sparse graphs with millions of vertices and edges.

Traditional APSP approaches such as the Floyd-Warshall algorithm [17] are computationally intensive and consume large amounts of memory. With time complexity $O(n^3)$ and space complexity $O(n^2)$, they are often unsuitable for large-scale graphs, even when parallelized. Repeatedly applying Dijkstra's algorithm [12] for every source vertex can introduce computational redundancy and irregular memory access, making it difficult to parallelize efficiently.

Recent research has increasingly leveraged GPU acceleration to enhance APSP computations. An important recent advancement in this direction is the FASTAPSP algorithm [54], which integrates Single-Source Shortest Path (SSSP) computations,

localized Floyd-Warshall kernels, and Min-Plus matrix operations within a GPU framework. Nevertheless, the performance of FASTAPSP remains constrained by graph partition quality and excessive CPU–GPU data transfers. Low-quality graph partitions can increase execution time, while high data transfer overhead further limits scalability and reduces overall efficiency.

This thesis directly addresses these performance bottlenecks by pursuing two complementary research directions: (1) evaluating and integrating advanced graph partitioners that improve edge cut and the boundary-to-interior vertex ratio, and (2) optimizing the GPU execution pipeline to minimize CPU–GPU data transfers and fully exploit GPU parallelism. These optimizations significantly improve the algorithm's efficiency and scalability, enhancing its applicability to practical, large-scale APSP problems across various domains.

1.2 Our Contribution

This thesis presents a practical and performance-driven enhancement of the FASTAPSP algorithm. The two central contributions are: First, we optimize the CPU–GPU execution strategy by reducing data transfers and restructuring data transformation routines as dedicated GPU kernels. These changes significantly reduce transfer overhead and unlock additional parallelism. The optimized version achieves an average speedup of **2.41**× over the original implementation across a wide variety of graphs, with peak gains of **6.7**× observed on the largest inputs.

Second, we evaluate and integrate a set of alternative graph partitioners into the FASTAPSP pipeline, including multiple variants of the Karlsruhe High Quality Partitioning (KAHIP) algorithm and the GPU-based JET partitioner. We analyze the trade-offs between partition quality and runtime, demonstrating that stronger partitioners like some of the KAHIP variants yield tangible runtime benefits. Notably, the KAHIP social variant achieves an average runtime improvement of **2.7%** compared to the default METIS partitioner, while the KAHIP economical variant provides improvements of up to **7%** when excluding the partitioning overhead.

Together, these improvements make FASTAPSP run faster. This is especially true for large-scale graphs, where partition quality and data transfer overhead are critical performance factors.

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 presents the necessary background on graph theory, partitioning algorithms, and GPU programming models. Chapter 3 reviews relevant prior work in parallel APSP algorithms and graph partitioning methods. Chapter 4 presents our contributions, which include the integration of alternative graph partitioners and optimizations to the GPU execution pipeline. Chapter 5 provides a detailed

experimental evaluation of both contributions, including runtime comparisons and partition quality metrics. Finally, Chapter 6 concludes the thesis with a summary of findings and suggestions for future research.

CHAPTER 2

Fundamentals

In this chapter, we introduce the fundamental concepts required to understand the optimization of the All-Pairs Shortest Path (APSP) problem. Key terms related to graph theory, GPU-accelerated computation, and graph partitioning are explained, with a focus on their relevance to APSP and the performance optimizations explored in this thesis.

2.1 Graph Terminology

We define a **directed graph** $G = (V, E)$ as a set of vertices V and a set of directed edges E, where each edge $(u, v) \in E$ represents a one-way relationship from vertex u to vertex v. In contrast, an **undirected graph** is one where the edges do not have a direction, meaning that if $(u, v) \in E$, then $(v, u) \in E$ as well.

A **weighted graph** assigns a numerical weight w to each edge (u, v), representing the cost or distance of traversing that edge. If all edges are assigned the same weight or no weight at all, the graph is referred to as an **unweighted graph**.

2.2 Shortest Path Problems

The **single-source shortest path (SSSP) problem** involves finding the minimum weight path from a source vertex s to all other vertices $t \in V \setminus \{s\}$ in a weighted graph $G = (V, E)$. Common algorithms like Dijkstra's algorithm are used to solve the SSSP problem.

The **all-pairs shortest path (APSP) problem** extends this idea by finding the shortest path between every pair of vertices $u, v \in V$. Solutions to the APSP problem are critical in applications like network routing, city planning, and optimization tasks where all shortest path distances need to be computed.

2.3 Graph Partitioning

In parallel computing, **graph partitioning** refers to the process of dividing a graph $G = (V, E)$ into smaller subgraphs or **components** $V_1, V_2, \ldots, V_k$, where $V_1 \cup V_2 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The goal is to balance computational load and minimize communication overhead, particularly in parallel and distributed systems.

An important consideration in graph partitioning is the **edge cut**, which is the set of edges that connect vertices in different components. The objective is often to minimize the edge cut, as it directly influences the amount of inter-process communication required when solving graph problems in parallel.

A **boundary vertex** of a component V_i is one that has an edge connecting it to a vertex in another component, whereas an **interior vertex** has no such connections. The boundary vertices are crucial in parallel algorithms because they typically form the "interface" through which different components communicate.

2.4 Computational Frameworks: CPU vs. GPU

The **Central Processing Unit (CPU)** is the main processing unit in a computer, designed for sequential execution of tasks. While CPUs excel at handling a wide variety of tasks, they are typically limited by the number of cores and are less suited for highly parallelized computations.

On the other hand, the **Graphics Processing Unit (GPU)** is specialized for parallel computation. GPUs consist of many small cores that allow them to perform thousands of operations simultaneously. This makes GPUs ideal for tasks like graph traversal and shortest-path computation, where large amounts of independent work can be parallelized.

Execution on the GPU is organized hierarchically: individual **threads** are grouped into **warps** (typically 32 threads), which are further organized into **thread blocks**. A **GPU kernel** is a function that is launched across many threads in parallel, often numbering in the thousands or millions. Threads within a warp execute instructions in a lockstep manner, which has implications for performance when branching occurs. Programming frameworks such as CUDA [7] and OpenCL [49] expose this execution model to developers, allowing them to fine-tune parallel algorithms to fully exploit the GPU's hierarchical concurrency.

2.5 CPU–GPU Data Transfers and Optimization

In GPU-accelerated algorithms, one of the key performance bottlenecks is the time spent on **CPU–GPU data transfers**. This refers to the movement of data between the CPU's main memory and the GPU's memory. These transfers can be slow compared to the computational power of the GPU, and minimizing them is essential for optimizing the

performance of GPU-based algorithms. This thesis focuses on reducing the frequency and volume of CPU–GPU data transfers. By keeping relevant data on the GPU for as long as it is needed, we reduce the overhead associated with frequent data transfers, thus enhancing computational efficiency.

2.6 Compressed Sparse Row (CSR) Format

Efficient graph representation is crucial for handling large-scale sparse graphs, as encountered throughout this thesis. Sparse graphs, characterized by a relatively small number of edges compared to the total possible connections between vertices, can lead to significant inefficiencies if represented using dense structures. Therefore, to optimize memory efficiency and computational performance, we represent graphs using the **Compressed Sparse Row (CSR)** format.

In the CSR format, the graph is compactly represented using three arrays. The first array stores the non-zero edge weights, capturing only the edges that explicitly exist. The second array contains the column indices, identifying the destination vertex for each corresponding edge weight. Finally, a third array holds row pointers, which indicate the positions in the first two arrays where each vertex's edge list begins, effectively mapping the vertex indices to their respective edges.

In our context, this choice is especially important because FASTAPSP replicates the entire graph data structure on each process. To keep memory usage manageable, especially when working with large graphs and multiple GPUs, it is essential to store the graph as compactly as possible. The CSR format supports this by representing only the non-zero edges, thereby avoiding the overhead associated with dense representations. In addition, its layout enables efficient sequential memory access, which matches well with how GPU threads consume data during graph traversal and computation.

CHAPTER 3

Related Work

This chapter surveys related work on the two main components of our approach: graph partitioning and all-pairs shortest path (APSP) algorithms. We first review partitioning methods, focusing on multilevel techniques used by METIS, KAHIP, and the GPU-based JET partitioner. We then discuss classical and GPU-based APSP algorithms, including PART APSP, its decentralized variant, and FASTAPSP, which serves as the baseline for our optimizations.

3.1 Graph Partitioning

Graph partitioning is a critical problem in parallel computing, with numerous approaches designed to optimize the computation and minimize the communication between subgraphs. There has been extensive research on graph partitioning, and we refer the reader to the relevant literature [4, 6]. Many high-quality graph partitioners employ a multilevel approach, such as KAHIP [44], JET Partitioner [19], METIS [26], SCOTCH [8], HMETIS [27, 25], and (MT)-KAHYPAR [20, 21, 22]. In this chapter, we review the key literature on graph partitioning, focusing on the multilevel partitioning approaches used by KAHIP, JET Partitioner, and METIS, which are central to this work.

3.1.1 Multilevel Graph Partitioning

Several state-of-the-art partitioners, including KAHIP, JET, and METIS, employ the **multilevel graph partitioning** (MGP) approach. This technique is widely used for efficiently partitioning large-scale graphs, as it reduces complexity by simplifying the problem across hierarchical levels. The process consists of three main phases: coarsening, initial partitioning, and refinement.

In the **coarsening** phase, the graph is iteratively reduced by recursively contracting vertices and edges. Each contraction step merges vertices based on a heuristic, such as edge

matching or label propagation, to form a smaller, coarser representation of the original graph. This phase aims to reduce the graph size while preserving its essential structure, making it more tractable for partitioning.

Once the graph has been sufficiently coarsened, the algorithm proceeds to the **initial partitioning** phase. At this stage, the graph is small enough to be efficiently partitioned. Depending on the partitioner, either a recursive bisection method or a direct k-way partitioning approach is used to divide the graph into parts. This initial partition serves as a starting point for refinement in the next phase.

The final phase is **uncoarsening** (also known as refinement). Here, the graph is gradually restored to its original size by recursively applying the inverse of the coarsening process. At each level, the partitioning is improved using local search methods, such as Kernighan–Lin [30] or Fiduccia–Mattheyses [16], to minimize edge cuts and balance the partitions as the graph is uncoarsened back to its finer levels.

The strength of the multilevel approach lies in its ability to simplify the partitioning process by breaking the problem into smaller, more manageable pieces. This significantly reduces the complexity of partitioning large graphs and ensures scalability across large datasets. Consequently, the multilevel partitioning approach allows KAHIP, JET, and METIS to handle large graphs efficiently while maintaining high-quality partitions.

3.1.2 Partitioning Algorithms

METIS

METIS employs a multilevel k-way partitioning strategy that reduces the graph size through coarsening, computes an initial partition at the coarsest level, and then progressively refines the solution during uncoarsening. Earlier versions of METIS used a recursive bisection approach, but the current default is a direct k-way partitioning scheme [26]. In its coarsening phase, METIS recursively contracts the graph using a greedy heavy-edge matching (HEM) strategy, merging vertex pairs connected by high-weight edges to form super-vertices. This process preserves key structural properties while significantly reducing the graph size. During uncoarsening, METIS refines partitions using efficient Kernighan-Lin-inspired heuristics, including Greedy Refinement (GR) and Global KL Refinement (GKLR). These methods adjust vertex assignments along partition boundaries to reduce edge cuts while respecting balance constraints.

KAHIP

KAHIP also follows the multilevel approach but incorporates significantly more sophisticated refinement techniques compared to METIS. In particular, KAHIP supports various coarsening schemes including edge rating, high-quality matchings via the Global Path Algorithm (GPA), and size-constrained label propagation used to guide coarsening and preserve balance. It offers multiple configurations, from fast to high-quality modes, such as

`eco`, `strong`, `fast` and `social`, each tuned for specific application domains. During coarsening, KAHIP applies edge contraction heuristics and balance-aware matching schemes that aim to preserve partition balance early on.

What sets KAHIP apart is its use of advanced global search techniques during Uncoarsening, including *F-cycles* and *W-cycles* [43]. These strategies repeatedly coarsen and uncoarsen the graph to escape local optima and improve partition quality beyond what is typically achievable with a single V-cycle. Refinement in KAHIP combines multiple techniques applied at different granularities: localized Kernighan–Lin-style search and flow-based improvements are used between pairs of blocks in the quotient graph, while size-constrained label propagation is employed both in pairwise and global (k-way) refinement phases. This multi-strategy approach enables KAHIP to produce high-quality cuts while strictly maintaining balance constraints.

JET

JET also follows a multilevel approach, but it significantly diverges from METIS and KAHIP in its methods. The coarsening step in JET uses a two-hop matching technique, which initially matches edges with high weights and then extends this matching to include two-hop matches if many vertices remain unmatched. This approach helps preserve more graph structure in the coarse representation. After coarsening, JET uses the METIS algorithm to partition the coarsest graph. This ensures that JET leverages the high-quality partitioning capabilities of METIS for the coarsened graph. In the Uncoarsening (refinement) phase, JET differs from both METIS and KAHIP by using label propagation to improve the partition quality. JET refines partitions using a modified label propagation scheme that prioritizes cut reduction without enforcing balance in each step. It allows temporarily unbalanced moves, including negative-gain ones, through a custom afterburner filter, and restores balance in a separate rebalancing phase. This decoupling enables larger, more effective vertex migrations while maintaining overall partition quality and balance. The descriptions above provide an overview of how METIS, KAHIP, and JET differ in their overall multilevel partitioning strategies. To highlight the design choices more clearly, we now break down the multilevel process into its three main phases, coarsening, initial partitioning, and refinement, and compare how each partitioner implements them.

Coarsening

METIS performs coarsening by recursively contracting vertices using a *greedy heavy-edge matching (HEM)* strategy [26]. In this approach, edges with the largest weights are preferentially matched, contracting connected vertices into super-vertices. This reduces the graph size rapidly while preserving important structural characteristics. The HEM technique aims to maintain the quality of the partitioning throughout the multilevel process.

In contrast, KAHIP employs a multilevel coarsening scheme based on edge rating functions and high-quality matchings. Instead of simple heavy-edge matching, it uses advanced

strategies such as the Global Path Algorithm (GPA) [44], which maximizes the overall edge rating. KAHIP also integrates size-constrained label propagation to improve balance early in the coarsening process. These mechanisms produce structurally representative coarse graphs and maintain partition balance from the outset.

JET, on the other hand, introduces a GPU-parallel *two-hop matching* strategy for coarsening [19]. Initially, it matches high-weight edges (similar to HEM), but then expands to match two-hop neighbors, such as twins, leaves, and relatives, ensuring more unmatched vertices are contracted. This leads to coarse graphs that retain more structural detail. The design is optimized for GPU throughput using hash-based matchmaker selection and warp-level concurrency.

Initial Partitioning

At the coarsest level, METIS computes a direct k-way partition of the graph. This is typically done when the graph is reduced to a small number of vertices. Earlier versions used recursive bisection, but modern METIS performs direct k-way partitioning at the coarsest level and then uncoarsens the solution through refinement.

KAHIP takes a more flexible approach to initial partitioning. It supports multiple strategies, including recursive bisection, spectral bisection, and the use of external partitioners such as Scotch. The selected method depends on the configuration preset (e.g., `eco`, `strong`) and may involve internal multilevel preprocessing followed by initial refinement using flow-based or FM-style methods.

Rather than implementing its own strategy, JET delegates the initial partitioning of the coarsest graph to the METIS library. This is typically done once the graph has been reduced to a few thousand vertices. By offloading this step to a well-tested CPU-based method, JET ensures high-quality initial partitions while keeping the coarsening and refinement stages on the GPU.

Refinement (Uncoarsening)

METIS refines the partition during uncoarsening using lightweight heuristics inspired by the Kernighan–Lin algorithm. These include Greedy Refinement (GR), which adjusts boundary vertices if they yield gain without violating balance, and Global KL Refinement (GKLR), which uses gain-based queues and limited hill-climbing. These strategies are designed to be fast and effective, making METIS scalable even for large graphs [26].

KAHIP employs a hybrid refinement strategy that operates at multiple levels. Between pairs of blocks, it uses local search techniques such as flow-based refinement and a localized Fiduccia–Mattheyses heuristic. At the global level, it performs size-constrained label propagation in both pairwise and full k-way refinement modes. Additionally, it supports advanced global search mechanisms like *F-cycles* and *W-cycles* [43], which allow it to repeatedly coarsen and refine the graph to escape local minima.

Finally, JET performs refinement using a GPU-parallel label propagation algorithm designed to operate in two stages [19]. The first stage (JETLP) focuses on unconstrained cut improvement using gain-based label updates. The second stage (JET) restores balance through a dedicated rebalancing pass. JET also includes an *afterburner* heuristic that enables negative-gain moves by filtering and reordering vertex adjustments, leading to further cut improvements. This decoupled two-phase approach allows for both aggressive optimization and efficient GPU utilization.

Each of these partitioners has its own strengths, and the choice of partitioner depends on the target platform and the specific requirements of the graph partitioning task. In this thesis, we evaluate the performance of the All-Pairs Shortest Path algorithm using these three partitioners.

The efficiency of All-Pairs Shortest Path (APSP) algorithms on large-scale graphs is closely tied to graph partitioning techniques. Methods like METIS, KAHIP, and JET partition the graph into smaller subgraphs, minimizing communication overhead and improving parallel processing. This allows APSP algorithms to scale better by reducing computational load and enhancing performance, especially for large graphs. As we now turn to the APSP algorithms, it's clear that effective partitioning plays a crucial role in optimizing their execution.

3.2 APSP Algorithms

The All-Pairs Shortest Path (APSP) problem is a fundamental challenge in graph theory and has been extensively studied using various approaches. The two most commonly employed methods are the Floyd-Warshall (FW) [17] algorithm and the repeated application of the Single-Source Shortest Path (SSSP) algorithm.

The FW algorithm, which utilizes dynamic programming, computes the shortest paths between all pairs of vertices in a systematic, iterative manner. Thanks to its matrix-based structure, the Floyd-Warshall algorithm can be parallelized effectively. However, its cubic time and quadratic space complexities render it impractical for large-scale graphs. Kannan et al. [24] were able to compute 6 million vertices in about 80 minutes by engaging 4,096 computing nodes of 24,576 GPUs in the Summit supercomputer system [53].

On the other hand, the SSSP-based approach, which solves the shortest path problem from each vertex, often employs algorithms like Dijkstra's [12] or Bellman-Ford [10]. SSSP-based methods often scale better on sparse graphs due to their lower time complexity of $O(n^2 \log n + nm)$, which can outperform the FW algorithm under certain conditions. However, the SSSP method faces challenges in parallel computing environments due to the need for complex data structures and a lack of optimizations for modern hardware features, such as SIMD or cache utilization [46]. Several attempts have been made to improve the parallelism of SSSP, such as the delta-stepping algorithm [37] and task-level parallelization for handling multiple SSSP tasks [45], but they still struggle with scalability, particularly on very large graphs. Notably, the largest graph tested by the SSSP algorithm consisted

of 1,024 vertices and required approximately 10 hours to process using two GPUs [41]. While both FW and SSSP have significant limitations, recent advancements have explored GPU-based solutions to APSP. The PART APSP algorithm [14] and its decentralized variant [13] employ a divide-and-conquer strategy, leveraging parallel computing resources to independently process subgraphs before combining the results. These methods significantly reduce the computational bottlenecks associated with the traditional algorithms, especially when large-scale data sets are involved. The FASTAPSP further builds upon these techniques by integrating both SSSP and FW algorithms, improving upon the scalability of previous methods while also eliminating inter-process communication overhead.

In this section, we provide an overview of the PART APSP and Decentralized PART APSP algorithms, followed by a detailed explanation of FASTAPSP. The latter is the focus of our work, where we optimize its performance.

3.2.1 PART APSP Algorithm

The PART APSP algorithm is designed to compute the shortest paths between all pairs of vertices for large graphs using parallel computing resources. It is based on the parallel Floyd-Warshall (FW) algorithm, which is described in more detail later in Section 3.3, and utilizes a partition strategy. The algorithm consists of four main steps 1:

- **Step 1: Preprocessing**: The input graph is divided into multiple components (subgraphs). The vertices of each component are categorized into interior and boundary vertices. The graph partitioning is done by the master node, which then broadcasts the result to all worker nodes.

- **Step 2: Local APSP Computation**: Each process applies the Floyd-Warshall algorithm to compute the APSP within its own component. This step involves updating the distance matrix for both interior and boundary vertices. The results are then sent to the master node.

- **Step 3: Boundary APSP Computation**: The boundary vertices from all components are gathered to form a boundary subgraph. The master node applies the Floyd-Warshall algorithm on this boundary subgraph to compute the shortest paths between boundary vertices. The results are then broadcast to the workers, which run the FW to update their local distance matrices to cover the case where the shortest path goes through boundary vertices.

- **Step 4: Global APSP Computation**: The final shortest path results for the entire graph are computed using the MIN-PLUS operation on the intermediate results from the local and boundary APSP computations. The MIN-PLUS operation efficiently combines the partial results from each component and the boundary subgraph. For each vertex in the subgraph it finds the minimum among the distances to

each of its boundary vertices plus the distance to each of the target subgraph boundary vertices and the distance from that boundary vertex to the target, as shown in the following formula.

$$C_{ij} = \min_{k,p} (A_{ik} + B_{kp} + D_{pj})$$

The pseudo code for the Partitioned All-Pairs Shortest Path Algorithm is provided in Algorithm 1. This algorithm is suitable for parallel systems where multiple workers compute the APSP for graph components independently. The time complexity of the PART APSP algorithm is $O(n^{2.25})$, which is better than applying the Floyd-Warshall algorithm directly to the entire graph. However, scalability can be limited due to the global communication required between processes.

Algorithm 1: Partitioned All-Pairs Shortest Path Algorithm

Input: A graph $G(V, E)$, where the weights of E are non-negative
Output: The distances between the vertices in G

Step 1: Preprocessing
Master: Read and partition the graph
Master: Scatter the partition component $C(i)$ to worker i processor;
Step 2: Local APSP Computation
for *all worker processors* p *in parallel* **do**
    Worker p: Floyd-Warshall($C(p)$);
    Worker p: Send computed boundary distances to the master processor;
Step 3: Boundary APSP Computation
Master: Extract the boundary graph G_{BG};
Master: Compute APSP on G_{BG};
Master: Send the boundary graph G_{BG} to all worker processors;
for *all worker processors* p *in parallel* **do**
    Worker p: Floyd-Warshall($C(p)$);
    Worker p: Send $C(p)$ distances to all worker processors;
Step 4: Global APSP Computation
for *all worker processors* p *in parallel* **do**
    for *all components* C_2 *in* G **do**
        Worker p: compute_APSP_between_components($C(p), C_2$);

3.2.2 DECENTRALIZED PART APSP Algorithm

The DECENTRALIZED PART APSP algorithm is an optimization of the PART APSP algorithm. It follows a similar four-step approach, but with a decentralized design to reduce global communication. The steps of the algorithm are as follows:

- **Step 1: Preprocessing**: Each process reads the original graph and partitions it independently. The graph is divided into components, and the vertices are categorized into interior and boundary vertices, similar to the PART APSP algorithm.
- **Step 2: Solving Local APSP**: Each process computes the APSP for its local vertices using the Floyd-Warshall algorithm. Afterward, the distance and path matrices for boundary vertices are broadcast to other processes.
- **Step 3: Solving the Boundary SSSP Problem**: Each process calculates the shortest paths from its boundary vertices to all other boundary vertices using a Single-Source Shortest Path (SSSP) algorithm. Then each process runs the FW to update the local distance matrices in case the shortest path goes through boundary vertices. This step reduces the communication required compared to the PART APSP algorithm.
- **Step 4: Solving the Global APSP**: The final step is the same as in the PART APSP algorithm, where the MIN-PLUS operation is used to combine the intermediate results from the local and boundary APSP computations to compute the global APSP.

The Decentralized PART APSP algorithm reduces global communication by performing boundary calculations independently and using the SSSP algorithm to compute shortest paths between boundary vertices. This leads to improved scalability, particularly in large parallel systems. The time complexity of the Decentralized PART APSP algorithm is also $O(n^{2.25})$, but the optimization leads to better performance in scenarios with many processing nodes. The pseudo code for the Decentralized PART APSP algorithm is shown in Algorithm 2.

3.3 Building Block Algorithms

We now turn to the specific algorithms used in FASTAPSP, focusing on the GPU-based building blocks that are employed in its second, third, and fourth steps. The FASTAPSP algorithm consists of four main steps, as illustrated in Figure 3.1. First, the algorithm partitions the graph into equal-sized components. In the second step, it uses the global single-source shortest path (SSSP) algorithm by Harish and Narayanan [23] to calculate the shortest paths from the boundary vertices to all other vertices in the graph. In the third step, it applies the local blocked Floyd-Warshall algorithm [28] within each component. In the last step, it calculates the distances from the interior vertices in each component to all vertices, resulting in the complete all-pairs shortest path matrix. In this section, we provide a detailed explanation of the algorithm used in the second, third and fourth steps of the FASTAPSP.

Algorithm 2: Decentralized Partitioned All-Pairs Shortest Path Algorithm

Input: A graph $G(V, E)$, where the weights of E are non-negative
Output: The distances between the vertices in G

Step 1:
for *all processor* p *in parallel* **do**
    read and decompose the graph
    get own component $C(p)$
Step 2:
for *all processor* p *in parallel* **do**
    Floyd-Warshall($C(p)$) send computed boundary distances to all other processors
Step 3:
for *all processor* p *in parallel* **do**
    G_{BG} = extract_boundary_graph(G)
    for *all vertex* v *in* G_{BG} **do**
        solve_SSSP(v, G_{BG})
for *all processor* p *in parallel* **do**
    Floyd-Warshall($C(p)$) send computed distances to all other processors
Step 4:
for *all processor* p *in parallel* **do**
    for *all component* C_2 *in* G **do**
        compute_APSP_between_components($C(p), C_2$)

3.3.1 Harish and Narayanan's algorithm

This algorithm follows a pattern similar to Dijkstra's algorithm [12], but with a key difference: instead of processing only the vertex at the top of the priority queue as in Dijkstra's algorithm, Harish and Narayanan's algorithm processes all vertices whose paths were updated in the previous step. These are then processed in parallel. As shown in Figure 3.3, in the first step, only the source vertex is active. In the second step, all its children become active, and in the third step, the children of these children are activated.

This approach significantly improves upon Dijkstra's algorithm because it leverages the parallel processing capabilities of GPUs. Each GPU thread is assigned to a vertex. This allows the algorithm to compute shortest paths in parallel and achieve substantial performance gains.

After providing the intuition, we will delve deeper into the details. Initially, each GPU thread is assigned to a single vertex. The algorithm maintains, for each vertex, its cost, temporary cost, path, and temporary path, where the cost represents the best known distance from the source vertex.

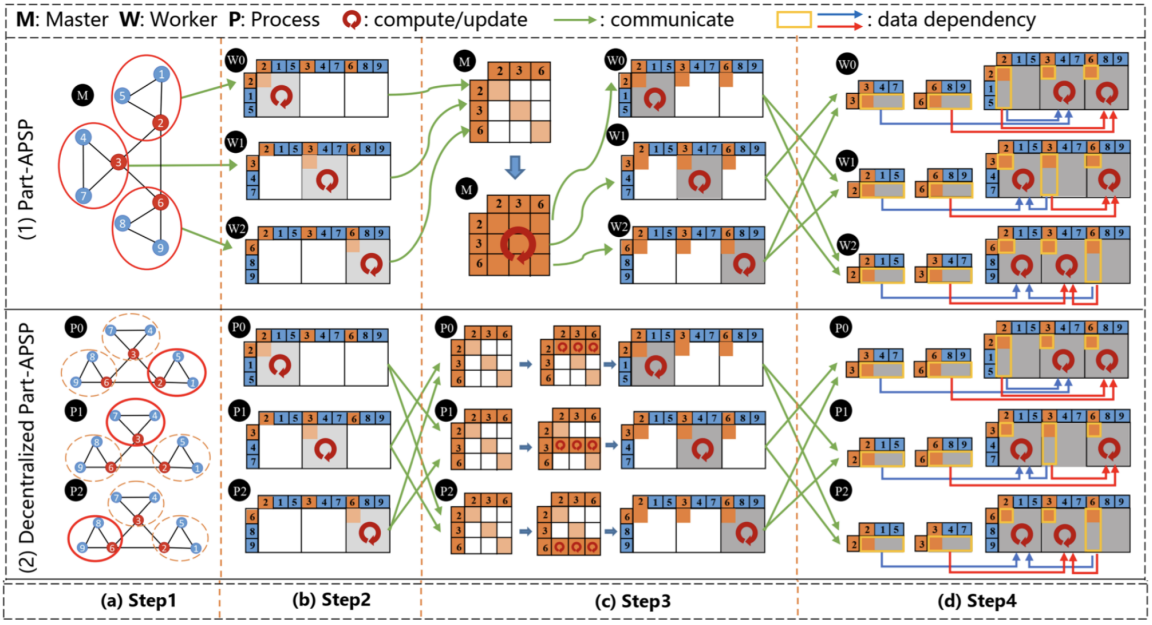

Figure 3.1: An illustration of the two algorithms: (1) PART APSP, and (2) DECENTRALIZED PART APSP. Adapted from Yang et al. [54].

The algorithm proceeds in iterations, each comprising three synchronized phases. During the first phase, every thread verifies the activity status of its assigned vertex. Upon confirming that it is active, the thread evaluates all of its neighboring vertices. It computes the sum of its vertex cost and the edge weight to each neighbor, comparing it against the neighbor's temporary cost. If this sum is smaller, the neighbor's temporary cost is updated using atomic operations, as described by Kemp [29].

In the second phase, when a vertex is active, its corresponding thread verifies for each of its neighbors, whether the temporary cost of its neighbor equals the sum of its own cost and the edge weight to that neighbor. Essentially, this checks if the shortest path to the neighbor goes through this specific active vertex. If this condition holds true, the active vertex is appended to the neighbor's temporary path.

During the third phase, each thread examines whether the cost of its assigned vertex exceeds its temporary cost. If so, it signifies that the vertex's cost has been updated in the current iteration. Consequently, the thread updates the vertex's cost and path to match its temporary cost and temporary path and marks the vertex as active. The algorithm proceeds to iterate until no vertices remain active, indicating that further improvement to any vertex's path is not possible. At this point, the algorithm concludes.

Algorithms 1, 2, and 3 represent the different kernel functions (phases) of the algorithm. The boolean array M_a indicates whether a vertex is active, while P_a stores the path and TP_a information for each vertex. The arrays C_a and TC_a hold the current and temporary distances, respectively, ensuring efficient distance updates during computation. In addition,

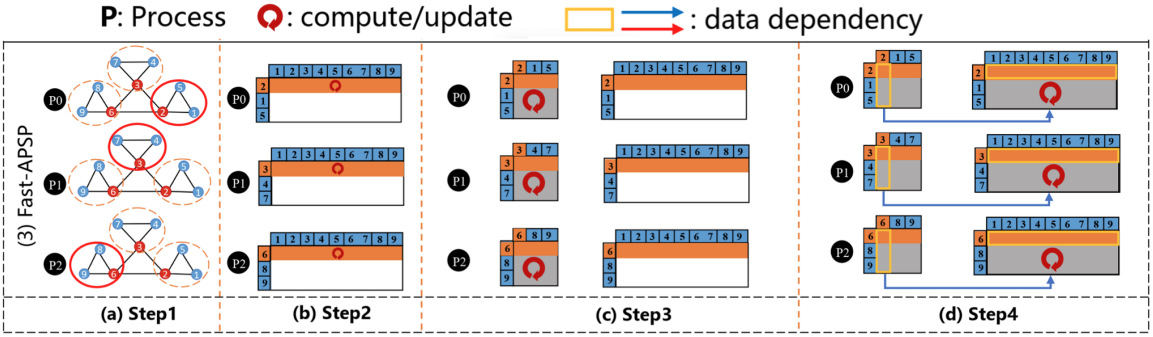

Figure 3.2: Demonstrations of the four different steps in the FASTAPSP algorithm. Adapted from Yang et al. [54].

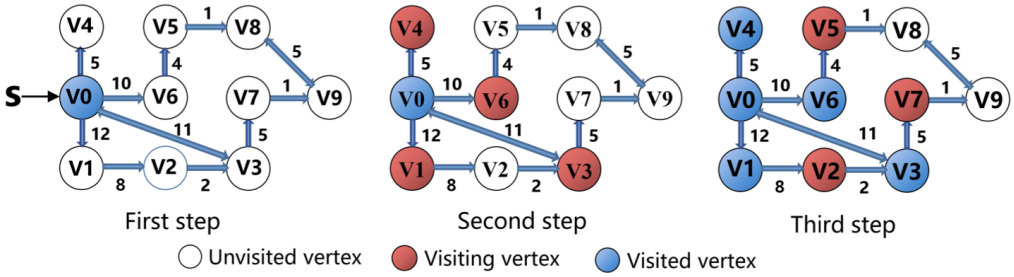

Figure 3.3: Overview of Harish and Narayanan's algorithm executing the first three steps. Reproduced from Yang et al. [54].

several improvements have been made to Harish's GPU algorithm. Okuyama et al. [39] leverage coarse-grained parallelism by implementing a task parallelization strategy that associates each task with an SSSP problem. The delta stepping algorithm [40], proposed as a compromise solution, demonstrates good performance on GPUs. However, the authors also note that for planar graphs such as road networks, which typically have high diameters and low degrees, the delta stepping algorithm is not as efficient. Furthermore, a number of other works [11, 36, 52] address parallel solutions to the SSSP problem.

Algorithm 3: SSSP-KERNEL1

Input: $V_a, E_a, W_a, M_a, C_a, TC_a, P_a, TP_a$
Output: Updated values of C_a, P_a

```
1 t_i ← threadID ;
2 if M_a[t_i] then
3     for all neighbors nid of t_i do
4         atomic:
              if TC_a[nid] > C_a[t_i] + W_a[nid] then
5                 TC_a[nid] ← C_a[t_i] + W_a[nid] ;
```

Algorithm 4: SSSP-KERNEL2

Input: $V_a, E_a, W_a, M_a, C_a, TC_a, TP_a$
Output: Updated values of M_a, TP_a

```
1 t_i ← threadID ;
2 if M_a[t_i] then
3     M_a[t_i] ← false ;
4     for all neighbors nid of t_i do
5         if TC_a[nid] == C_a[t_i] + W_a[nid] then
6             TP_a[nid] ← t_i ;
```

Algorithm 5: SSSP-KERNEL3

Input: $V_a, E_a, W_a, M_a, C_a, TC_a$
Output: Updated values of C_a, P_a, M_a

```
1 t_i ← threadID ;
2 if C_a[t_i] > TC_a[t_i] then
3     C_a[t_i] ← TC_a[t_i] ;
4     P_a[t_i] ← TP_a[t_i] ;
5     M_a[t_i] ← true ;
6 TC_a[t_i] ← C_a[t_i] ;
7 TP_a[t_i] ← P_a[t_i] ;
```

3.3.2 Sequential Floyd-Warshall as a Basis for Parallelization

The third step of the FASTAPSP algorithm uses a parallel version of the Floyd-Warshall [28] algorithm to solve the all-pairs shortest path problem in each of its components. Before explaining the parallel version, we will first explain the original version [17] of the algorithm, as understanding the sequential version is crucial for comprehending the parallel one.

The Floyd-Warshall algorithm uses a dynamic programming technique to solve the all-pairs shortest path problem. It performs n iterations where, n is the number of vertices in the graph, and in each iteration k, it checks for each pair of vertices $v_i, v_j \in V$ if there exists a shorter path between v_i and v_j through vertex v_k. It employs an $n \times n$ distance matrix to represent the current shortest distances between all pairs of vertices. We denote this distance matrix as *dist*. In the initialization stage, for each edge $(v_i, v_j) \in E$, the entry *dist*$[i, j]$ is set to $w(v_i, v_j)$. All entries that do not correspond to edges in the graph are set to infinity.

After the initialization stage, the algorithm performs n iterations. In the k_{th} iteration, it checks for all pairs $v_i, v_j \in V$ if *dist*$[i, j]$ > *dist*$[i, k]$ + *dist*$[k, j]$. If this is the case, it updates *dist*$[i, j]$ to *dist*$[i, k]$ + *dist*$[k, j]$. After n iterations, the algorithm terminates, and the distance matrix represents the all-pairs shortest path solution.

Algorithm 6: Floyd-Warshall Algorithm

Input: $n \times n$ matrix *dist*, where $dist[i, j]$ is the initial distance between vertices i and j, and infinity for non-edges

Output: *dist* matrix with the shortest distances between all pairs of vertices

```
1 for k = 1 to n do
2     for i = 1 to n do
3         for j = 1 to n do
4             if dist[i, j] > dist[i, k] + dist[k, j] then
5                 dist[i, j] ← dist[i, k] + dist[k, j]
6             end
7         end
8     end
9 end
```

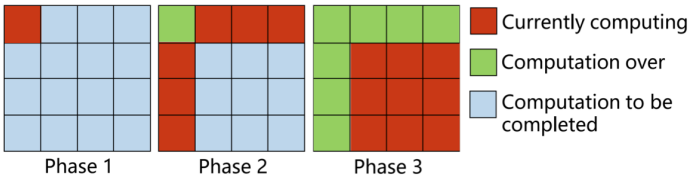

Figure 3.4: Overview of Katz and Kidar's algorithm in its first iteration. Reproduced from Katz et al. [28].

3.3.3 Blocked Floyd-Warshall algorithm

Having introduced the sequential Floyd-Warshall algorithm, we now turn to the parallel variant proposed by Katz and Kider [28], which adapts the original formulation for efficient execution on GPUs. Yang et al. referred to it as the Blocked FW algorithm. The first step is to divide the *dist* matrix into b equal-sized sub-blocks, with each sub-block assigned to a GPU thread-block.

The algorithm performs n/b iterations, where each iteration of the Blocked FW simulates n/b iterations of the original FW algorithm. In each iteration, a primary block is set. The primary block in the p_{th} iteration is the p_{th} diagonal block. An iteration consists of three phases, as illustrated in Figure 3.4. In the first phase, the primary block calculates its all-pairs shortest paths using the Floyd-Warshall algorithm. In this phase, only one thread-block is processing the primary block.

In the second phase, the blocks in the same row of the primary block and the blocks in the same column of the primary block are active. A close examination of the distance matrix accesses when running the FW algorithm reveals that to calculate a block in the same row or column as the primary block, only the entries of this specific block and the entries of the primary block are needed. This is explained as follows: the p_{th} iteration of

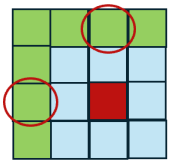

Figure 3.5: Dependency of a specific block in phase 3 of the first iteration.

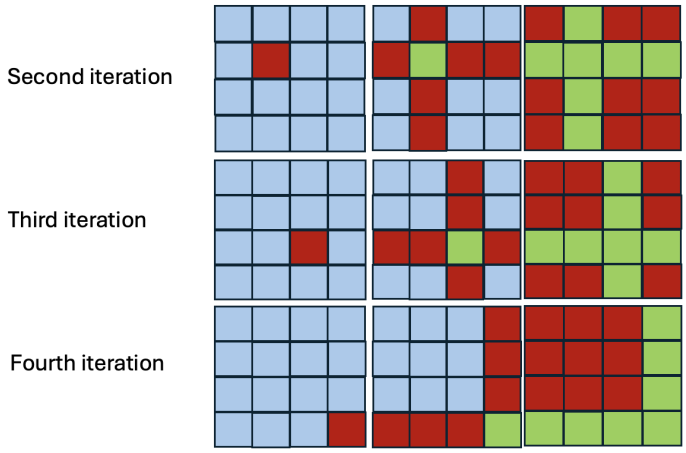

Figure 3.6: The second, third, and fourth iterations

the blocked FW represents the $k = (n/b) * ((p_{th}) - 1)$ to $k = (n/b) * (p_{th})$ iterations in the FW algorithm, where with close examination only the entries as described above are needed to calculate the block. This way the calculations of all the blocks that are active in this phase are performed in parallel.

In the third step all other blocks in the matrix are calculated for the same k values. Using the same argument as in the second phase, it can be shown that each of the remaining blocks depends only on its own entries and the entries of the blocks in its row and column that intersect with blocks in the row and column of the primary block. For instance, as seen in Figure 3.5, the red block depends only on the circled blocks and its own entries. This means that all the blocks calculated in the third phase do not depend on each other within this iteration and can, therefore, be computed in parallel. Figure 3.6 demonstrates the three phases in the second, third, and fourth iterations of the blocked FW algorithm.

3.3.4 Min-Plus

The Min-Plus operation plays a key role in the FASTAPSP algorithm and can be viewed as a variant of the **General Matrix Multiplication (GEMM)** operation, which is a core operation in matrix computations. In standard matrix multiplication, the entry C_{ij} in the

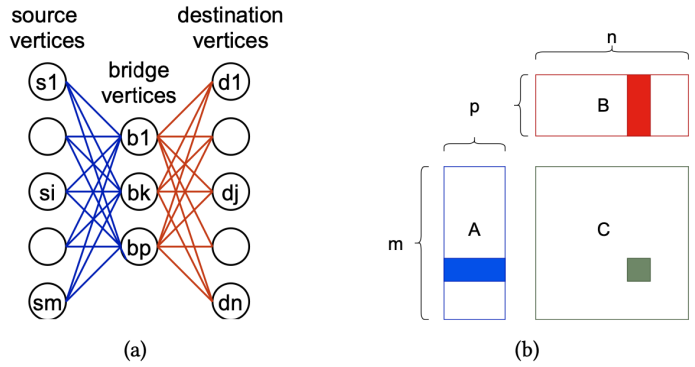

Figure 3.7: An illustration of the Min-Plus operation. Reproduced from Yang et al. [54].

product of two matrices A and B is computed as:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

where n represents the number of columns in A (and equivalently, the number of rows in B). In contrast, the **Min-Plus** operation modifies the traditional matrix multiplication by replacing the addition and multiplication operations. Specifically, it replaces the summation with the **minimum** operation and multiplication with **addition**, resulting in the following formulation for the elements of the matrix C:

$$C_{ij} = \min_{k} (A_{ik} + B_{kj})$$

Here, instead of summing products, the Min-Plus operation calculates the **minimum** over the sum of corresponding elements from the rows of A and the columns of B.

Although the Min-Plus operation is a modification of matrix multiplication, it retains the core structure of GEMM, making it computationally similar. This similarity enables the Min-Plus operation to benefit from optimization techniques commonly used for GEMM [9, 32, 50, 55], as demonstrated by Yang et al., particularly in the context of parallel computing.

3.4 FASTAPSP

As mentioned above, the FASTAPSP algorithm consists of four steps. In this section, we will explain each of these steps in more detail.

3.4.1 Step 1: Partition the graph

In the first step, the algorithm uses the graph partitioning tool METIS [26] to partition the graph into k sub-graphs, called components, of nearly the same size. Each component is

assigned to a process, and the set of boundary vertices B_i and the set of interior vertices In_i are identified. Additionally, the complete graph is stored in each of the processes in CSR (Compressed Sparse Row) matrix format.

3.4.2 Step 2: Solving SSSP problem of boundary vertices

Next, each process P_i (where i ranges from 1 to k) calculates the single-source shortest paths using Harish and Narayanan's algorithm. This is done from each boundary vertex of its component to all other vertices in the graph, see step 2 in Figure 3.1. This is feasible because each process has access to the complete graph, allowing this computation to proceed without any process communication.

3.4.3 Step 3: Computation of the APSP in each component

In the third step, each process computes the all-pairs shortest paths within its component using the blocked Floyd-Warshall algorithm, as shown in step 3 in Figure 3.1. Yang et al. proved that the distances calculated within each component are optimal. The following provides an intuitive explanation:

Shortest paths from boundary to interior vertices: The shortest paths from the boundary vertices to the interior vertices are optimal (see Step 2).

Paths between interior vertices: There are two cases for paths between interior vertices:

- **Paths within the component:** If the optimal path between two interior vertices includes only vertices within the same component, the blocked FW algorithm will compute the optimal path.
- **Paths involving external vertices:** If the optimal path includes vertices outside the component, the path must involve at least two boundary vertices of the component, see Figure 3.8(a). The path from the first boundary vertex to the target interior vertex is already optimal (as established in Step 2). Therefore, the distance from the source interior vertex to the target interior vertex, passing through the first boundary vertex, will also be optimal.

3.4.4 Step 4: Computing the shortest path from internal vertices to vertices in other components

Lastly, each component uses the knowledge obtained from Steps 2 and 3 to compute the shortest paths from each of its interior vertices to all vertices outside its component, by applying MIN-PLUS operations. The MIN-PLUS operation computes, for each interior vertex, the minimum of the distances from the vertex to one of the boundary vertices plus the distance from that boundary vertex to the target vertex.

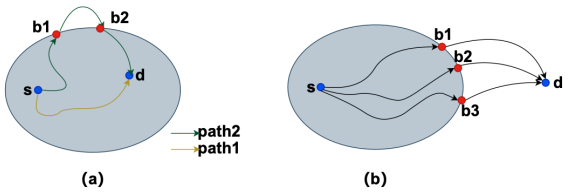

Figure 3.8: An illustration of paths involving external vertices (a). Path to vertex outside the component (b). Reproduced from Yang et al. [54].

Yang et al. provide a proof of the optimality of the MIN-PLUS operation. The intuition is straightforward: any path from a source interior vertex to a target vertex outside its component must traverse a boundary vertex of its component. The distance from the source vertex to its boundary vertex is computed in Step 3, and the distance from the boundary vertex to the target vertex is computed in Step 2. Thus, selecting the minimum among all possible paths, as shown in Figure 3.8(b), guarantees the optimal solution.

One key advantage of the FASTAPSP algorithm is that it eliminates the need for interprocess communication during the computation stages by storing the original graph in each process. This feature distinguishes it from similar algorithms such as the Part-APSP [14] and decentralized Part-APSP [13] algorithms, which require communication between the processes.

Algorithm 7: Fast All-Pairs Shortest Path Algorithm

Input : A graph $G(V, E)$, where the weights of E are non-negative
Output: The distances between the vertices in G

```
1  for all processor p in parallel do
2      read and decompose graph G;
3      get own component C(p);
4  for all processor p in parallel do
5      for all boundary vertex v in C(p) do
6          solve_SSSP(v, G);
7  for all processor p in parallel do
8      Floyd-Warshall(C(p))  // compute_APSP(C(p));
9  for all processor p in parallel do
10     // 1 MIN-PLUS operator;
11     compute_APSP_components(C(p), G);
```

3.5 Time Complexity Analysis

The time complexity of the FASTAPSP algorithm is strongly dependent on the ratio of boundary vertices to all vertices in the graph, which is denoted as $\overline{a}$. In each component, this ratio is $a_i = \frac{|B_i|}{|B_i|+|In_i|}$. In the first step, the graph partitioning using METIS takes $O(n \log n)$ time. In the second step, the single-source shortest path from all boundary vertices to all other vertices in the graph is computed, resulting in time complexity of $O(\overline{a}n(n \log n + m))$. In the third step, the algorithm calculates the all-pairs shortest path in each of the components. Since there are k tasks, the time complexity for this step is $O\left(k\left(\frac{n}{k}\right)^3\right)$. In the fourth step, there are k tasks that need to perform the MIN-PLUS operations. Each MIN-PLUS operation involves all the boundary vertices of its component and it is done for each interior vertex to all other vertices in the graph. This leads to a time complexity of $O\left(((\overline{a} * \frac{n}{k}) * ((1-\overline{a}) * \frac{n}{k}) * n) * k\right) = O\left(\overline{a}(1-\overline{a})\frac{n^3}{k}\right)$. The overall time complexity of the algorithm is $O\left(nlogn + \overline{a}n^2 \log n + \overline{a}nm + \frac{n^3}{k^2} + \overline{a}(1-\overline{a})\frac{n^3}{k}\right)$. Yang et al. were able to show that the time complexity for planar graphs can be simplified to $O(n^{9/4})$ and to $O(n^3)$ for non-planar graphs.

CHAPTER 4

Optimizing FASTAPSP Through Advanced Graph Partitioning and GPU Memory Management

In this section, we present the main contributions of our research. The goal of our project was to build upon the work of Yang et al. and identify ways to enhance the performance of their algorithm. Specifically, we focused on two main aspects. The first is the graph partitioning process in the algorithm's initial step. The second is the implementation flow, with an emphasis on optimizing CPU–GPU data transfers.

To improve the partitioning process, we explored alternative partitioners and integrated KAHIP and JET as replacements for METIS, as discussed in Section 4.1. In addition, we refined the implementation by minimizing CPU–GPU data transfers, ensuring that necessary data remains on the GPU for the duration of dependent computations. The details of this optimization and its impact are elaborated in Section 4.2.

4.1 The Graph Partitioning

The FASTAPSP algorithm utilizes the METIS graph partitioner to divide the input graph into k subgraph components. Graph partitioning plays a crucial role in determining the algorithm's efficiency, as it influences how well the workload will be parallelized across computing processes. As demonstrated by Yang et al., the quality of the partition, particularly the boundary-to-interior vertex ratio, greatly influences the performance of the FASTAPSP algorithm. A higher number of boundary vertices increases computational overhead. This is because more operations are required in the second (3.4.2) and fourth (3.4.4) steps, which involve calculating shortest paths and performing Min-Plus operations.

In the second step, each process computes the Single-Source Shortest Path (SSSP) from all boundary vertices of its assigned component to every other vertex in the graph. More

boundary vertices result in more SSSP computations, which increases the number of times the shortest path algorithm must be executed and impacts the overall runtime, particularly with large graphs.

Similarly, in the fourth step, each process performs Min-Plus operations to propagate distances from the interior vertices of its assigned subgraph through the boundary vertices of the same subgraph, which serve as bridge vertices. By reducing the number of boundary vertices, the number of required Min-Plus operations decreases, resulting in more efficient execution.

To optimize the partitioning, we explored alternative graph partitioners, namely KAHIP and JET. The experiments we conducted (see Chapter 5) demonstrated that these partitioners were able to improve the boundary-to-interior vertex ratio, and we investigated how this improvement influences the overall performance.

Implementation Details of Graph Partitioners. To integrate alternative partitioners into FASTAPSP, we modified the graph decomposition step to support both JET and KAHIP while retaining the existing METIS-based approach.

KAHIP Integration: KAHIP follows the same interface as METIS, making its integration straightforward. The implementation first transforms the input adjacency list into a format suitable for KAHIP's main partitioning function. The partitioning process is configured using a balance constraint of 3%, ensuring near-even distribution of vertices across partitions. Additionally, a fixed random seed of 42 is used for reproducibility. Different KAHIP modes, such as "FAST", "ECO", "STRONG", and "SOCIAL", were integrated to explore trade-offs between partitioning speed and quality. The resulting partition labels, computed for each vertex by KAHIP, are then used to construct subgraph assignments within FASTAPSP. Each vertex is assigned to a subgraph based on its partition label, and a mapping structure is populated to group vertices by subgraph ID. This assignment is subsequently used in the parallel stages of the algorithm to manage per-subgraph computation and boundary handling.

JET Integration: The JET Partitioner, being a GPU-based partitioning tool using Kokkos [15], required a different integration approach. The input graph is converted into a `KokkosSparse::CrsMatrix`, which is the format expected by JET's partitioning function. The partitioning step is performed on the GPU, leveraging Kokkos for memory management and computation. Similar to KAHIP, JET was configured to use a balance constraint of 3% to maintain partition quality. After partitioning, the resulting labels are copied back to the CPU and stored in a mapping structure that assigns vertices to their respective subgraphs. An additional integration challenge involved the explicit creation and initialization of Kokkos device views. Since JET expects its input data on the GPU, the host-side adjacency structures had to be manually mirrored using `create_mirror_view` and synchronized via `deep_copy` operations. Moreover, constant initialization of device-side vertex weights required a temporary mirror view to set values on the host before transferring them to the device.

4.2 CPU–GPU Data Transfer Optimization

In the original FASTAPSP implementation, the algorithm follows a straightforward approach where data is transferred between the CPU and GPU at each stage of the computation. Specifically, the graph data required for each step is transferred to the GPU before the computations, and the results are subsequently moved back to the CPU after each step, as illustrated in Figure 4.1.

In contrast, our optimized approach reduces the frequency of these back-and-forth data transfers by retaining the necessary data on the GPU for as long as it is required for subsequent steps, as depicted in Figure 4.2.

These transfers are particularly costly due to the significant latency and bandwidth limitations of the communication channel between the CPU and GPU. Moving data between the two involves explicit memory copies and synchronization overhead, introducing a bottleneck that can significantly impact performance, particularly when frequent transfers disrupt the continuity of GPU computations. By minimizing these transfers, our optimized approach eliminates unnecessary memory operations, reduces synchronization delays, and significantly improves overall computational efficiency.

At first glance, one might assume that this approach would lead to an increase in the memory usage on the GPU. However, as demonstrated in Section 5.3.3, the peak memory usage on the GPU remains nearly unchanged. Instead, memory usage increases in other areas, with the peak staying almost the same, and only rising during stages where memory utilization is below the peak limit. In the following sections, we will provide a detailed explanation of the data structures used in both approaches and describe the processes of each approach in turn.

4.2.1 Data Structures

In this section, we describe the main data structures used throughout the algorithm. The input graph is initially provided in the Matrix Market (.mtx) format and is internally converted to the Compressed Sparse Row (CSR) format. CSR is a widely used representation for sparse graphs, consisting of three compact arrays: the row pointer array, which indicates the start of each vertex's adjacency list; the column index array, which lists the target vertices of all edges; and the values array, which stores the corresponding edge weights. This format is particularly space efficient for sparse graphs and enables fast traversal and indexing on the GPU. Once constructed, the CSR representation is transferred to the GPU and reused across various computation stages.

For subgraph-specific operations, the `subgraph_dist` and `subgraph_path` matrices are used to store the shortest distances and the corresponding paths between all the vertices of a subgraph and all vertices of the complete graph. During the Floyd-Warshall algorithm, additional intermediate matrices like `inner_to_inner_dist` and `inner_to_inner_path` are created to compute the shortest paths between vertices

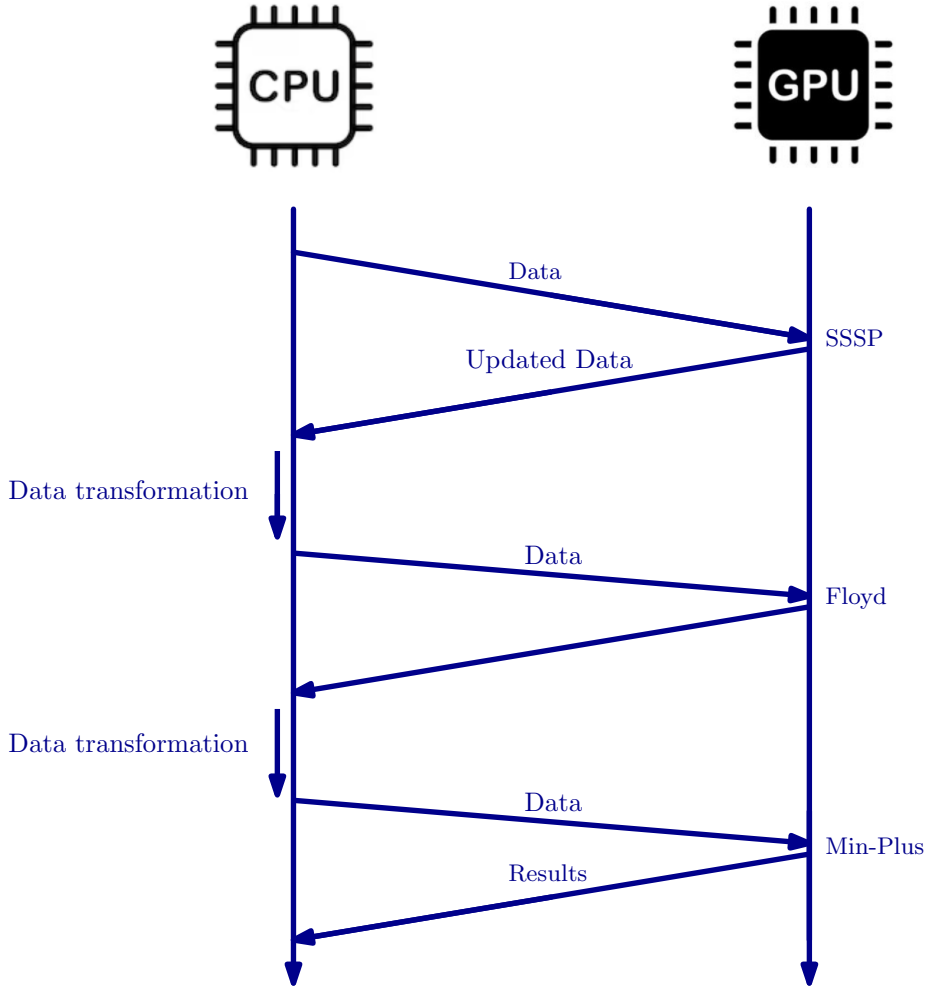

Figure 4.1: CPU–GPU Data Transformation Workflow in the Original Version

within the subgraph. Similarly, the `inner_to_boundary_dist` matrix holds the shortest distances between the interior vertices of the subgraph and the boundary vertices, which is used in the Min-Plus operation later in the process. Additionally, the `id_to_index` structure maps each vertex ID to its corresponding index in the subgraph, while `index_to_id` performs the inverse mapping, converting indices back to their original vertex IDs. Table 4.1 presents an overview of the key data structures used in the algorithm, detailing their memory requirements, computational complexity, and usage across different processing stages. In this table, b_p denotes the number of boundary vertices in subgraph p, n the total number of vertices, m the total number of edges, and k the number of subgraph components. Algorithm 8 presents the pseudo code for FASTAPSP, providing a detailed breakdown of each function's data structures and signatures.

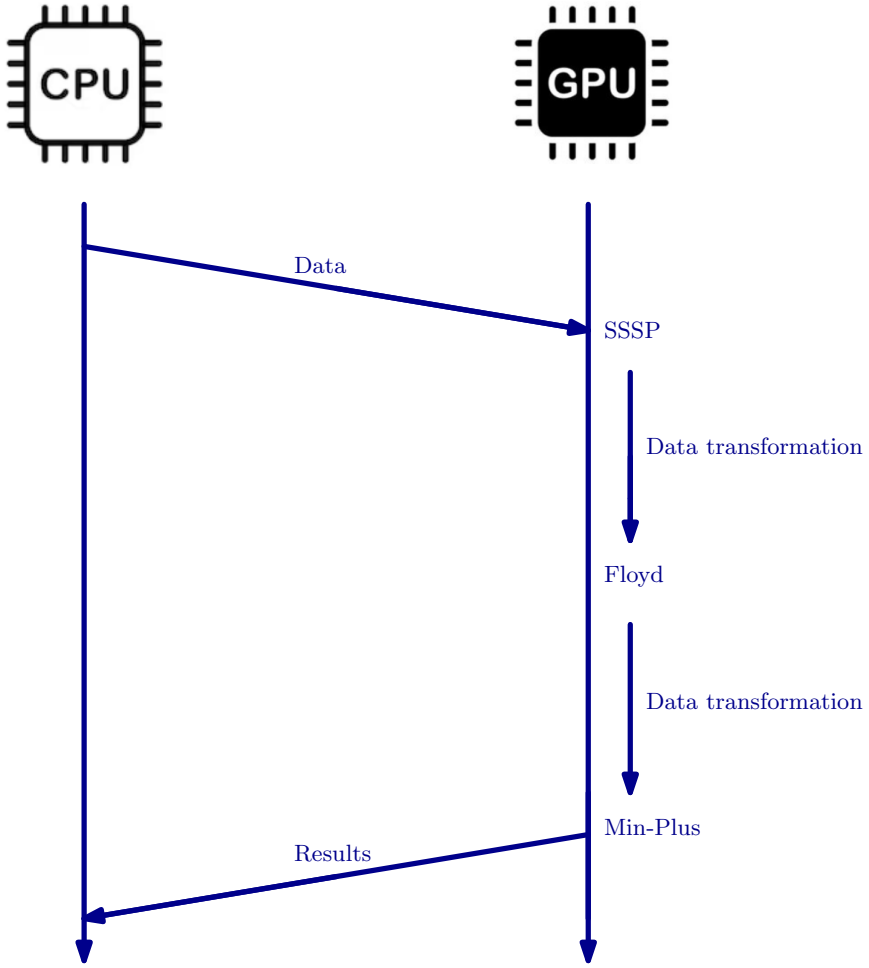

Figure 4.2: CPU–GPU Data Transformation Workflow in the Optimized Version

4.2.2 The Original Approach

The original FASTAPSP algorithm progresses through a series of stages, with data being transferred back and forth between the CPU and GPU at each step. This continuous data movement introduces significant CPU–GPU data transfer overhead, which can become a bottleneck, particularly when processing large graphs. The chronological flow of events, as depicted in Figure 4.1, is as follows:

Step 1: Graph Partitioning

In this step, the graph partitioning algorithm is applied to the input graph, with each vertex being assigned to a subgraph component. This operation is entirely performed on the CPU.

Data	Size	O-Notation	Used In
G_{CSR}	$\|V\| + 2\|E\|$	$\mathcal{O}(m)$	SSSP, Min-Plus
`subgraph_dist`, `subgraph_path`	$\|V\|^2/k$	$\mathcal{O}(n^2)$	SSSP, Floyd-Warshall, Min-Plus
`inner_to_inner_dist`, `inner_to_inner_path`	$\|V\|^2/k^2$	$\mathcal{O}(n^2)$	Floyd-Warshall
`inner_to_boundary_dist`	$(\|V\| - b_{\text{p}})/k * b_{\text{p}}$	$\mathcal{O}(n * b_{\text{p}})$	Min-Plus
`id_to_index, index_to_id`	$\|V\|$	$\mathcal{O}(n)$	SSSP, Min-Plus
Final APSP Results of $C(p)$	$\|V\|^2/k$	$\mathcal{O}(n^2)$	Output

Table 4.1: Data Structures and Their Memory Usage

Algorithm 8: FASTAPSP Algorithm with data structures and signatures

Input : A graph $G(V, E)$, where the weights of E are non-negative
Output: The distances between all vertices in G

for *all processor* p *in parallel* **do**
    `// Step 1: Graph Partitioning (CPU)`
    Read and decompose graph G on CPU;
    Get own component $C(p)$;

for *all processor* p *in parallel* **do**
    `// Step 2: SSSP Computation on Boundary Vertices (GPU)`
    for *each boundary vertex* v *in* $C(p)$ **do**
        `// Run SSSP kernel on GPU`
        Solve_SSSP_GPU(G_{CSR}, $subgraph\_dist$, $subgraph\_path$, id_to_index);

for *all processor* p *in parallel* **do**
    `// Step 3: Floyd-Warshall Algorithm on Subgraphs (GPU)`
    `// Prepare subgraph distance matrices`
    build_subgraph_matrix_CPU(G_{CSR} $subgraph\_dist$, $subgraph\_path$, $inner\_to\_inner\_dist$, $inner\_to\_inner\_path$, id_to_index, $index\_to\_id$)
    `// Run Floyd-Warshall on GPU`
    floyd_warshall_GPU($inner\_to\_inner\_dist$, $inner\_to\_inner\_path$,);

for *all processor* p *in parallel* **do**
    `// Step 4: Min-Plus Operation (GPU)`
    `// Construct intermediate matrices`
    construct_inner_to_boundary_dist_CPU($inner\_to\_inner\_dist$, $inner\_to\_boundary\_dist$);
    `// Execute Min-Plus kernel`
    min_plus_GPU($subgraph\_dist$, $subgraph\_path$, $inner\_to\_boundary\_dist$);

Step 2: SSSP Algorithm on Boundary Vertices

In this step, the Single-Source Shortest Path (SSSP) algorithm is applied to all boundary vertices in the subgraph. The graph's CSR adjacency matrix, along with the `subgraph_dist` and `subgraph_path` matrices, are transferred to the GPU for computation. After the computation is completed, the results are returned to the CPU, and the matrices are removed from the GPU memory.

Step 3: Floyd-Warshall Algorithm within Subgraphs

As part of the preprocessing, the `inner_to_inner_dist` and `inner_to_inner_path` matrices are constructed from the updated `subgraph_dist` and `subgraph_path` matrices. This matrix construction is carried out on the CPU, as it involves data transformations that are essential for preparing the data for the subsequent steps of the algorithm. The blocked Floyd-Warshall algorithm is then used to compute all pairs of shortest paths within the subgraph. The `inner_to_inner_dist` and `inner_to_inner_path` matrices are transferred to the GPU for computation. Once the calculations are complete, the results are transferred back to the CPU.

Step 4: Min-Plus Algorithm

Following the completion of the blocked Floyd-Warshall algorithm, the `inner_to_boundary_dist` matrix is constructed from the `inner_to_inner_dist` matrix. This matrix stores the shortest distances between the interior and boundary vertices. The construction of this matrix is performed on the CPU, as it involves transforming the computed data from the previous step into the format required for Min-Plus operations.

The final step involves the Min-Plus algorithm, which computes the shortest paths from the interior vertices to the boundary vertices and then from the boundary vertices to all other vertices. During this step, the relevant matrices `inner_to_boundary_dist`, `subgraph_dist`, and `subgraph_path` are transferred to the GPU for computation. Once the Min-Plus operation is complete, the results are transferred back to the CPU.

4.2.3 Optimized Approach in FASTAPSP

In our optimized version of FASTAPSP, the algorithm reduces CPU–GPU data transfers by keeping data on the GPU for as long as it is required for subsequent computations, minimizing the need for frequent and costly transfers. As illustrated in Figure 4.2, this approach avoids unnecessary memory movements by retaining intermediate results on the GPU. Our optimized version continues to use the same core data structures as the original approach while refactoring data transformations previously handled by the CPU into GPU kernels, ensuring that only the final results are transferred back to the CPU after computation is completed.

Step 1: Graph Partitioning

In the optimized version of FASTAPSP, the graph partitioning step was extended to support alternative partitioners beyond the original METIS based approach. While the conceptual goal of assigning vertices to subgraphs remains unchanged, the implementation differs depending on the selected partitioner.

When using KAHIP, partitioning is performed on the CPU in a manner similar to METIS, with only minor differences in the configuration interface and partitioning modes. In contrast, JET executes the partitioning step on the GPU. The graph is first transferred to the device, where partitioning is carried out using Kokkos based kernels. After computation, the resulting partition labels are transferred back to the CPU and used to construct the subgraph mapping. As a result, this step does not reduce CPU–GPU data transfers but provides an alternative partitioning strategy that may yield improved subgraph quality. For additional integration details, see Section 4.1.

Step 2: SSSP Algorithm on Boundary Vertices

In this step, the graph's CSR adjacency matrix, along with the `subgraph_dist` and `subgraph_path` matrices, are transferred to the GPU for computation. After the SSSP computations, the `subgraph_dist` and `subgraph_path` matrices are updated and retained on the GPU, avoiding unnecessary transfers back to the CPU.

Step 3: Floyd-Warshall Algorithm within Subgraphs

The `inner_to_inner_dist` and `inner_to_inner_path` matrices are constructed from the updated `subgraph_dist` and `subgraph_path` matrices. Unlike the original approach, these matrices are now created directly on the GPU, eliminating the need for data transfers to the CPU, which significantly reduces overhead.

The blocked Floyd-Warshall algorithm is then executed entirely on the GPU, as in the original approach. However, in our optimized version, the `inner_to_inner_dist` and `inner_to_inner_path` matrices remain on the GPU, preventing unnecessary transfers to the CPU and reducing data movement bottlenecks.

Step 4: Min-Plus Algorithm

The `inner_to_boundary_dist` matrix is constructed entirely on the GPU using the `inner_to_inner_dist` matrix. This avoids unnecessary transfers to the CPU and fully utilizes the parallelism provided by the GPU.

Next, the Min-Plus operation is executed on the GPU using the matrices `subgraph_dist`, `subgraph_path`, and `inner_to_boundary_dist`. Once the operation is complete, the final results are transferred back to the CPU.

Our optimized approach significantly reduces CPU–GPU data transfer frequency. By refactoring data transformation functions into GPU kernels, it takes full advantage of GPU

parallelism, reducing transformation time and improving computational efficiency. As demonstrated in 5.3.3, the peak memory consumption occurs during the fourth step in both approaches, with only a slight difference between them. However, in the second and third steps, our optimized approach exhibits considerably higher memory usage, though still lower than the peak observed in the fourth step.

Implementation Details of the Optimized GPU Data Transformations. To optimize the performance of FASTAPSP, we restructured key data transformation operations into independent GPU kernels. These transformations involve constructing, extracting, and decoding the subgraph distance matrices, all of which were previously handled on the CPU. The new GPU implementation eliminates redundant data transfers and leverages parallel execution for efficient computation. This section describes the three key transformation steps and their corresponding GPU kernels, which are detailed in Algorithm 9, Algorithm 10, and Algorithm 11.

Subgraph Matrix Construction and Floyd-Warshall Preparation The first transformation initializes the `inner_to_inner_dist` and `inner_to_inner_path` matrices from the `Compressed Sparse Row (CSR)` format and the `subgraph_dist` and `subgraph_path` matrices.

The original CPU implementation iterated sequentially over all vertices to extract and store connectivity information and distances, leading to inefficient memory accesses and high overhead. In contrast, our optimized GPU implementation launches three independent kernels. The first kernel constructs the `inner_to_inner_dist` and `inner_to_inner_path` matrices using the `CSR` graph structure, allowing each vertex to process its neighbors in parallel. The second kernel copies subgraph-related distances from `subgraph_dist` into `inner_to_inner_dist`, ensuring that all necessary subgraph information is transferred. Finally, the third kernel initializes diagonal entries in `inner_to_inner_dist`, correctly setting all self-distances to zero. This transformation is detailed in Algorithm 9.

Once these transformations are completed, the matrices `inner_to_inner_dist` and `inner_to_inner_path` are ready for Floyd-Warshall execution on the GPU, eliminating unnecessary CPU transfers.

Extraction of Inner-to-Boundary Distances Once the subgraph structure is built, the Min-Plus computation requires a thin matrix containing only distances from interior vertices to boundary vertices. In the CPU-based approach, this transformation was executed with a nested loop that iterated over each interior vertex and extracted its connections to boundary vertices.

The GPU-based version significantly reduces overhead by using Kernel 4 to assign one thread per interior vertex. Each thread extracts the relevant entries in parallel and copies them into the new matrix. This transformation is detailed in Algorithm 10.

Algorithm 9: Build the subgraph matrices for the Floyd-Warshall algorithm

Input : G_{CSR}, $subgraph\_dist$, $subgraph\_path$
Output: $inner\_to\_inner\_dist$, $inner\_to\_inner\_path$

```
// Step 1: Process Graph Structure (Kernel 1)
```

foreach $v \in V_p$ ***(in Kernel 1) in parallel*** **do**
  foreach *neighbor* u *of* v **do**
    if $u \in V_p$ **then**
      $inner\_to\_inner\_dist[v, u] \leftarrow subgraph\_dist[v, u]$;
      $inner\_to\_inner\_path[v, u] \leftarrow v$;

```
// Step 2: Copy Subgraph Data (Kernel 2)
```

foreach *boundary vertex* $v_b \in V_p$ ***(in Kernel 2) in parallel*** **do**
  foreach $u \in V_p$ **do**
    $inner\_to\_inner\_dist[v_b, u] \leftarrow subgraph\_dist[v_b, u]$;
    $inner\_to\_inner\_path[v_b, u] \leftarrow subgraph\_path[v_b, u]$;

```
// Step 3: Initialize Diagonal Entries (Kernel 3)
```

foreach $v \in V_p$ ***(in Kernel 3) in parallel*** **do**
  $inner\_to\_inner\_dist[v, v] \leftarrow 0$;

Algorithm 10: Extraction of Inner-to-Boundary Distances

Input : $inner\_to\_inner\_dist$
Output: $inner\_to\_boundary\_dist$

```
// Step 1: Extract Inner-to-Boundary Distances (Kernel
   4)
```

foreach *interior vertex* $v_i \in V_p$ ***(in Kernel 4) in parallel*** **do**
  foreach *boundary vertex* $v_b \in V_p$ **do**
    $inner\_to\_boundary\_dist[i, v_b] \leftarrow inner\_to\_inner\_dist[v_i, v_b]$;

Final Decoding of Floyd-Warshall Results After executing the Floyd-Warshall algorithm on each subgraph, the computed shortest path distances need to be transferred back to the global distance matrix. The CPU version used a nested loop to update each subgraph's distances and paths in the global structure, leading to expensive sequential memory accesses.

To accelerate this process, Kernel 5 assigns a separate thread to each vertex pair, enabling efficient parallel copying of the Floyd-Warshall results into the final matrix. The kernel also maps subgraph indices to their global counterparts, avoiding unnecessary data transfers. This process is described in Algorithm 11.

Summary By parallelizing these data transformations, we achieve significant improvements over the CPU implementation. First, data transfers are reduced, as the subgraph matrices are constructed and retained on the GPU, eliminating the need for frequent memory exchanges. Additionally, parallelism is enhanced by fully parallelizing each transformation step, leading to a substantial reduction in computational time.

Algorithm 11: Decoding of Floyd-Warshall Results

Input : $inner\_to\_inner\_dist$, $inner\_to\_inner\_path$, $subgraph\_dist$, $subgraph\_path$

Output: $subgraph\_dist$, $subgraph\_path$

```
// Step 1: Transfer Processed Results to Global Graph
   Matrix (Kernel 5)
```

1 **foreach** $(v_i, v_j) \in V_p \times V_p$ ***(in Kernel 5) in parallel*** **do**

2 $\quad subGraph[v_i, v_j] \leftarrow inner\_to\_inner\_dist[v_i, v_j]$;

3 $\quad subGraph\_path[v_i, v_j] \leftarrow inner\_to\_inner\_path[v_i, v_j]$;

CHAPTER 5

Experimental Evaluation

This chapter presents an in-depth experimental evaluation of the contributions proposed in this thesis. Our goal is to assess the practical impact of the two main enhancements to the FastAPSP algorithm: (1) the integration of alternative graph partitioners and (2) the optimization of the CPU–GPU execution pipeline to reduce data transfers. We aim to answer key questions regarding the runtime performance, memory behavior, and partition quality of the modified algorithm compared to the original baseline.

To guide our evaluation, we formulate the following research questions:

Partitioning Impact (P):

P1 Does improving the boundary-to-interior vertex ratio lead to better overall runtime performance of FastAPSP?

P2 What is the trade-off between partitioning time and total runtime when using stronger partitioners such as KaHIP or Jet?

Execution Optimization (O):

O1 How does reducing CPU–GPU data transfers affect overall runtime performance?

O2 How is total runtime distributed across algorithmic stages, and what does this reveal about the source of performance improvements?

O3 What is the effect of the optimized execution strategy on GPU memory consumption across different graph sizes?

O4 How much of the runtime improvement can be attributed to restructuring data transformations as GPU kernels?

To address these questions, we perform a series of experiments on a collection of real-world and synthetic graphs. Each experiment is designed to isolate specific aspects of the algorithm, allowing us to compare runtime behavior, partitioning quality, and resource usage across different configurations.

The remainder of this chapter is organized as follows: Section 5.1 introduces the experimental setup. Section 5.2 presents results related to the impact of graph partitioning strategies, addressing questions **P1-P2**. Section 5.3 analyzes the optimized execution strategy, focusing on questions **O1-O4**.

5.1 Experimental Setup

This section presents the experimental setup used to evaluate the performance and scalability of the proposed improvements to FASTAPSP. We first describe the hardware and software environment. Then, we introduce the baseline implementation and the graph partitioning tools used for comparison. Next, we summarize the benchmark datasets and partitioning parameters. The section concludes with the evaluation methodology.

5.1.1 Hardware and Software Environment

All experiments were conducted on the bwUniCluster 2.0 HPC system [5], using Ice Lake + GPUx4 nodes. Each node is equipped with two Intel Xeon Platinum 8358 processors (2 sockets $\times$ 32 cores, 2.6 GHz), 512 GB of RAM, and four NVIDIA H100 GPUs (80 GB each). Nodes are interconnected via dual InfiniBand HDR200 links and run Red Hat Enterprise Linux 8.4. The Lustre parallel file system was used, with high-performance temporary storage provided via `$TMPDIR` (backed by NVMe SSDs).

Experiments were scheduled using SLURM, with one MPI process per GPU. Small and medium graphs were processed on a single GPU; larger graphs used four. The project was compiled with GCC 11.4.0, OpenMPI 5.0.2, and CUDA 12.2 using `nvcc_wrapper`, with full optimization enabled (`-O3`). The main executable was run with 1 or 4 MPI ranks, corresponding to the number of GPUs used per experiment.

5.1.2 Baselines

We compare our optimized implementation of FASTAPSP against the original version by Yang et al. [54], which we refer to as the baseline. In their work, Yang et al. demonstrated that FASTAPSP significantly outperforms prior state-of-the-art APSP algorithms across a wide range of datasets, including PART APSP [14], DECENTRALIZED PART APSP [13], a GPU-based Dijkstra implementation [39], and a variation of a CPU-based Dijkstra implementation [48]. Accordingly, we treat their implementation as the standard reference for evaluating our contributions.

To assess the effect of partitioning quality, we compare several alternative partitioners, *KaHIP eco*, *KaHIP social eco*, *KaHIP social fast*, and the *Jet* partitioner, which we refer to as KAHIP_ECO, KAHIP_SOC_ECO, KAHIP_SOC_FAST, and JET, respectively, against the default METIS baseline.

To evaluate the impact of our architectural optimizations independently, we also compare the total runtime of the original and optimized versions using identical graph instances and METIS partitions. This ensures a fair and controlled assessment focused exclusively on the implementation-level improvements.

5.1.3 Graph Instances

Table 5.1 lists the graph instances used in our experiments. The collection spans both synthetic and real-world networks, which are commonly used to benchmark shortest path algorithms and graph partitioning methods. All graphs were processed as undirected and unweighted and were converted into the format required by the FASTAPSP framework.

The graphs were obtained from either the SuiteSparse Matrix Collection [31] or the Network Data Repository with Interactive Graph Analytics and Visualization [42]. We preserved original attributions to the graph creators where possible.

A large subset of the mesh and scientific graphs originates from the 10th DIMACS Implementation Challenge [1], including delaunay_n16, wing, fe_tooth, fe-ocean, and 598a. Several OpenStreetMap-derived road networks, such as luxembourg_osm, belgium_osm, and netherlands_osm, were also used. Additional scientific and infrastructure graphs, including onera_dual, usroads-48, and sc-pwtk, were sourced from SuiteSparse.

The dataset also includes real-world web and social graphs. Notably, web-sk-2005 and web-it-2004 were created by Boldi et al. [2], while soc-youtube originates from the work of Mislove et al. [38]. The com-amazon graph was obtained from the SNAP collection [33], and the road networks roadnet-pa and roadnet-ca are credited to Leskovec et al. [34].

To ensure a fair and meaningful comparison, our dataset includes all graphs used in the original FASTAPSP evaluation by Yang et al. [54]. This allows us to directly assess the impact of our modifications under the same conditions. In addition, we augment this set with several larger real-world graphs, ny-sorted, com-amazon, belgium_osm, road-roadnet-ca, and netherlands_osm, to extend the experimental coverage and evaluate scalability on more demanding instances.

5.1.4 Methodology

Our evaluation focuses on three primary criteria: total runtime, runtime breakdown by algorithmic stage, and partitioning quality. Runtime is measured as wall-clock time, with results collected separately for core computational phases such as SSSP, Floyd–Warshall, and Min–Plus. Partition quality is assessed through structural metrics including edge cut and the boundary-to-interior vertex ratio.

Table 5.1: Graph instances used in our APSP experiments. For each graph, we show the number of nodes n, the number of edges m, and the graph type.

Graph	Nodes (n)	Edges (m)	Type
delaunay_n16	65,536	196,575	Mesh / Scientific
wing	62,032	121,544	Mesh / Scientific
fe_tooth	78,136	452,591	Mesh / Scientific
onera_dual	85,567	252,384	Mesh / Scientific
598a	110,971	741,934	Mesh / Scientific
luxembourg_osm	114,599	119,666	Road Network
web-sk-2005	121,422	334,419	Web Graph
usroads-48	126,146	161,950	Road Network
fe-ocean	143,437	409,593	Mesh / Scientific
ny-sorted	264,346	365,050	Road Network
com-amazon	334,863	925,872	Web Graph
sc-pwtk	217,891	5,653,221	Mesh / Scientific
web-it-2004	509,338	7,178,413	Web Graph
soc-youtube	495,957	1,936,748	Social Network
roadnet-pa	1,090,920	1,541,898	Road Network
belgium_osm	1,441,295	1,549,970	Road Network
road-roadnet-ca	1,957,027	2,760,388	Road Network
netherlands_osm	2,216,688	2,441,238	Road Network

Partitioning performance is evaluated by running the optimized implementation of FASTAPSP with each integrated partitioner. All configurations are tested on multiple graphs and across a range of k values (i.e., the number of partitions), selected based on graph size and structure to reflect realistic parallel workloads and GPU memory constraints. To enable a deeper evaluation of partitioner behavior under varying conditions, and to simulate scenarios with different numbers of available GPUs, each graph was tested with multiple k values. Smaller graphs were assigned lower k values to preserve subgraph granularity, while larger graphs required higher k values to remain within device memory limits. A complete breakdown of the k values used per graph instance is provided in Appendix A.1.

Architectural improvements are assessed independently by comparing the original and optimized versions of FASTAPSP on identical graph inputs, all using the same METIS-generated partitions. This setup allows for a controlled analysis of the impact of reduced CPU–GPU data transfers and the restructuring of data transformation routines as CUDA kernels.

Unless otherwise noted, all experiments were conducted on identical hardware under consistent conditions. Performance metrics were aggregated using the geometric mean to ensure a balanced view across graph sizes, with both absolute and relative runtimes reported where appropriate.

To account for runtime variability, each configuration was executed 20 times on small and medium graphs, and 10 times on large graphs. We classify as large those instances exceeding one million vertices, specifically roadnet-pa, belgium_osm, road-roadnet-ca, and netherlands_osm. Reported values reflect the average across these repetitions.

We imposed a uniform partitioning time limit of two hours to maintain fair and consistent comparisons across all partitioners. In two cases, KAHIP_ECO on soc-youtube and JET on onera_dual, this limit was exceeded and no result was produced. These two graphs were therefore excluded from all geometric mean computations to avoid skewing aggregate metrics. This approach ensures that comparisons remain both fair and methodologically sound.

5.2 Evaluation of Graph Partitioning Strategies

5.2.1 The Partition Quality

To evaluate the impact of partition quality on runtime, we compare two structural metrics: edge cut and the boundary-to-interior vertex ratio. These metrics are chosen to evaluate the connection between structural partition quality and runtime behavior, as framed in research questions **P1** and **P2**. The edge cut and the boundary-to-interior vertex ratio are measured for KAHIP_ECO, KAHIP_SOC_ECO, KAHIP_SOC_FAST, and JET, using METIS as a baseline. We analyze how these metrics correlate with the total runtime of the APSP algorithm across a range of graph instances.

As shown in Table 5.2, KAHIP_ECO yields the lowest edge cuts overall, with an average reduction of **11%** compared to METIS. Its performance is notably impacted by one outlier, the web-sk-2005 graph, where the edge cut is significantly worse, skewing the geometric mean. Meanwhile, KAHIP_SOC_ECO and JET achieve improvements of **11%** and **7%**, respectively, without such regressions.

While edge cut is a commonly used proxy for partition quality, it does not reliably correlate with computational cost in FASTAPSP. Instead, the boundary-to-interior vertex ratio proves to be a more reliable predictor, as it governs the volume of computation and inter-partition communication in the SSSP and Min-Plus stages 4.1.

Table 5.3 shows that KAHIP_ECO and KAHIP_SOC_ECO reduce the boundary-to-interior vertex ratio by **6%** and **11%**, respectively, compared to METIS, while the JET partitioner performs substantially worse. This discrepancy is notably evident on the web-it-2004 graph: although JET achieves a **32%** reduction in edge cut, it produces a boundary-to-interior vertex ratio that is over **110%** higher, resulting in a **42%** increase in total runtime relative to METIS.

To further reinforce the connection between partition quality and algorithmic efficiency, Figure 5.1 presents the average runtime of the SSSP phase for each partitioner. As the most computationally expensive component of FASTAPSP, as demonstrated in Figure 5.6, the SSSP stage is particularly sensitive to the number of boundary vertices.

Consistent with the boundary-to-interior vertex ratio results, KAHIP_ECO achieves

the best performance, reducing SSSP runtime by **12.5%** compared to METIS, with KAHIP_SOC_ECO also showing a reduction of **3.5%**.

These improvements highlight how better partition quality translates directly into runtime benefits, particularly in the most performance-critical stage of the algorithm. This supports the hypothesis in **P1**, confirming that the boundary-to-interior vertex ratio significantly impacts end-to-end performance. While lower boundary ratios generally lead to shorter runtimes, the relationship is not strictly linear. We explore this nuance further in the next subsection.

Table 5.2: Edge cut comparison across partitioners for each graph instance. Lower values indicate fewer inter-partition edges, which can reduce communication overhead. Bold values indicate the best result per row

Graph	METIS	KAHIP_SOC_ECO	KAHIP_ECO	KAHIP_SOC_F	JET
delaunay_n16	2,449.00	2,170.00	**2,129.50**	2,474.50	2,405.15
wing	5,103.67	4,561.00	**4,100.33**	5,298.67	4,416.23
fe_tooth	20,733.63	19,158.25	**18,456.33**	22,146.52	19,596.16
onera_dual	5,193.33	4,288.00	**4,068.00**	4,520.67	-
598a	38,504.75	37,214.50	**35,098.00**	40,482.75	35,675.87
luxembourg_osm	276.50	265.00	**241.00**	284.00	278.37
web-sk-2005	119.00	85.00	326.00	**80.00**	81.75
usroads-48	334.00	**283.00**	286.00	342.00	346.50
fe-ocean	17,357.67	15,173.33	**13,917.67**	18,834.44	15,100.15
ny-sorted	2,140.00	1,731.00	**1,624.00**	1,929.00	1,939.60
com-amazon	75,582.84	59,715.19	**53,832.00**	63,753.00	64,720.80
sc-pwtk	673,703.33	653,115.00	**616,131.67**	743,767.67	631,800.73
web-it-2004	150,608.50	165,188.50	118,474.50	147,356.50	**101,639.90**
soc-youtube	1,047,247.67	1,031,893.33	-	1,105,530.00	**1,002,470.03**
roadnet-pa	31,481.50	28,366.00	**26,220.50**	32,455.00	33,386.85
belgium_osm	12,829.50	12,273.50	**11,126.00**	13,971.50	14,220.15
road-roadnet-ca	67,307.00	62,481.00	**57,704.00**	70,787.00	72,045.00
netherlands_osm	27,734.00	26,945.00	**24,056.00**	30,775.00	31,133.25
Arithmetic Mean	**70,391.56**	**68,045.33**	**61,482.72**	**74,671.10**	**64,299.15**
Geometric Mean Improvement	**1.00x**	**1.11x**	**1.11x**	**1.00x**	**1.07x**

Table 5.3: Boundary-to-interior vertex ratio across partitioners. This metric reflects the proportion of vertices on partition boundaries relative to internal vertices, with lower values generally correlating with better runtime performance. Bold values indicate the best result per row

Graph	METIS	KAHIP_SOC_ECO	KAHIP_ECO	KAHIP_SOC_F	JET
delaunay_n16	0.039	0.034	**0.033**	0.039	0.038
wing	0.167	0.157	**0.141**	0.175	0.148
fe_tooth	0.183	0.167	**0.161**	0.196	0.170
onera_dual	0.117	0.106	**0.098**	0.112	-
598a	0.225	0.215	**0.200**	0.239	0.203
luxembourg_osm	0.005	0.005	**0.004**	0.005	0.005
web-sk-2005	0.001	0.001	0.003	0.001	**0.001**
usroads-48	0.005	**0.004**	0.005	0.005	0.005
fe-ocean	0.213	0.221	0.213	0.240	**0.205**
ny-sorted	0.016	0.013	**0.012**	0.014	0.014
com-amazon	0.333	0.276	**0.244**	0.295	0.293
sc-pwtk	0.745	0.706	**0.640**	0.870	0.683
web-it-2004	0.140	0.094	0.146	**0.088**	0.296
soc-youtube	**0.959**	0.966	-	1.042	1.372
roadnet-pa	0.056	0.052	**0.047**	0.059	0.060
belgium_osm	0.018	0.017	**0.016**	0.020	0.020
road-roadnet-ca	0.068	0.064	**0.058**	0.072	0.073
netherlands_osm	0.025	0.025	**0.022**	0.028	0.029
Arithmetic Mean	**0.14**	**0.13**	**0.12**	**0.15**	**0.14**
Geometric Mean Improvement	**1.00x**	**1.11x**	**1.06x**	**0.992x**	**0.98x**

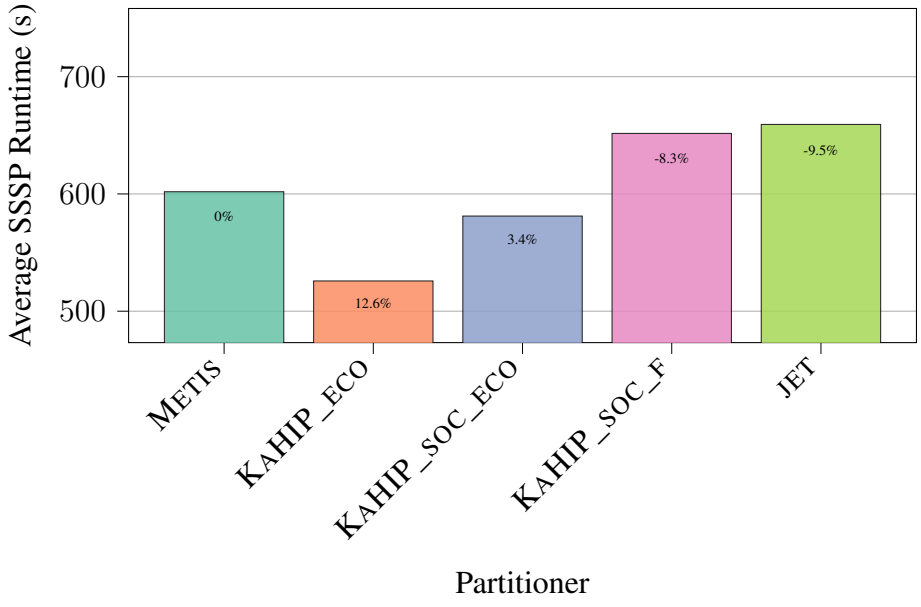

Figure 5.1: Average runtime of the SSSP phase across all graph instances for each partitioner. Lower runtimes indicate more efficient processing.

The boundary-to-interior vertex ratio emerged as a stronger predictor of runtime than edge cut, supporting hypothesis **P1**. Partitioners like KAHIP_ECO and KAHIP_SOC_ECO achieved up to **11%** reductions in this ratio, resulting in SSSP speedups of **12.5%** and **3.5%**, respectively. While the JET partitioner reduced edge cuts, its boundary ratio was significantly worse on certain graphs. For example, on web-it-2004, it increased the boundary-to-interior vertex ratio by over **110%**, leading to a **42%** runtime slowdown. These results emphasize that minimizing boundary vertices, not just edge cuts, is more critical for runtime efficiency.

5.2.2 Total Runtime Comparison

While structural partition quality provides valuable insights, the ultimate criterion for evaluating a partitioner's utility in FASTAPSP is its effect on end-to-end runtime. Figure 5.2 presents a boxplot comparison of total runtimes across all tested partitioners.

KAHIP_SOC_ECO achieves consistently strong performance on most graphs, slightly outperforming METIS in terms of average runtime across the majority of datasets 5.4, with an overall **2.7%** improvement.

JET performs well on small to medium-sized graphs, but its efficiency degrades with input size, suggesting that its partitioning strategy does not scale as effectively. In contrast, KAHIP_ECO excels on large graphs, despite its longer partitioning times. This is attributable to its ability to minimize both edge cut and boundary ratios, factors that become increasingly important as graph size and communication overhead grow.

These results highlight the trade-off described in **P2** between partitioning time and runtime efficiency. While KAHIP_ECO offers the best partition quality and scales well with input size, its long partitioning time increases the total runtime on small and medium graphs, making it less favorable in those scenarios. In contrast, KAHIP_SOC_ECO provides the most balanced performance overall.

KAHIP_SOC_ECO achieved the best overall performance with a **2.7%** speedup over METIS. Better partitioners like KAHIP_ECO required more time but produced higher-quality partitions, leading to the best results on large graphs. JET was significantly faster but yielded lower-quality partitions, making it more effective on small graphs. Overall, no single partitioner performed best across all inputs.

5.2.3 Runtime Excluding Partitioning Overhead

In many real-world scenarios, the partitioning of a graph is performed once and reused across multiple executions of the same algorithm. This is particularly relevant in settings where the input graph structure changes only slightly, such as during incremental updates, or where multiple executions are needed for parameter tuning or scenario analysis.

Table 5.4: Total runtime of FASTAPSP for each partitioner across all graphs. Results reflect wall-clock time and include partitioning. The geometric mean is calculated only over graphs for which all partitioners successfully completed.

Graph	METIS	KAHIP_SOC_ECO	KAHIP_ECO	KAHIP_SOC_F	JET
delaunay_n16	26.89	**25.35**	28.32	27.36	27.00
wing	44.93	43.75	51.56	46.74	**42.03**
fe_tooth	41.99	40.97	56.13	42.99	**40.58**
onera_dual	52.73	**51.61**	58.93	52.84	-
598a	71.45	71.69	98.51	73.53	**68.59**
luxembourg_osm	65.99	66.67	**65.64**	68.58	67.97
web-sk-2005	51.17	51.29	63.65	50.98	**50.36**
usroads-48	79.84	**77.67**	80.17	80.45	80.35
fe-ocean	286.87	297.68	314.93	311.78	**285.41**
ny-sorted	247.49	**223.64**	229.17	236.79	239.76
com-amazon	384.08	382.60	3,184.09	**367.70**	368.91
sc-pwtk	512.02	505.63	598.81	558.40	**498.84**
web-it-2004	170.83	153.16	267.51	**146.15**	243.52
soc-youtube	**364.18**	1,678.88	-	381.62	415.08
roadnet-pa	1,310.89	**1,252.77**	1,328.41	1,356.21	1,385.84
belgium_osm	1,773.47	1,738.34	**1,718.87**	1,875.39	1,904.24
road-roadnet-ca	3,449.99	3,309.36	**3,306.65**	3,541.40	3,619.33
netherlands_osm	5,290.01	5,272.84	**4,983.79**	5,725.00	5,785.25
Arithmetic Mean	**862.99**	**844.59**	**1,023.51**	**906.84**	**919.25**
Geometric Mean Speedup	**1.00x**	**1.027x**	**0.8x**	**0.983x**	**0.975x**

For example, in a traffic navigation system like Waze [51], the underlying road network remains largely static, while edge weights (e.g., travel times) vary dynamically. In such cases, it is practical to compute the partitioning once and reuse it, as the partitioning quality is not significantly affected by changes in edge weights.

To capture this practical setting, we evaluate the total runtime of FASTAPSP excluding the partitioning step. By removing this cost, we isolate the impact of partition quality on the core computational workload and more clearly highlight the efficiency gains provided by different partitioners. This perspective also addresses the trade-off posed in **P2**, demonstrating that partitioners with higher upfront costs may still yield better overall efficiency in repeated or long-running workloads.

The results, presented in Table 5.5, show that when the cost of partitioning is removed, performance differences between partitioners become more pronounced. Both KAHIP_ECO and KAHIP_SOC_ECO consistently outperform METIS, with average improvements of **7%** and **4%**, respectively, across nearly all graph sizes. Their advantage is especially evident on large-scale graphs, where better boundary management and reduced communication overhead have the greatest impact on SSSP and Min-Plus performance. In contrast, the JET partitioner shows only modest changes under this measurement. Since its partitioning time is relatively low, removing this cost does not significantly shift its standing.

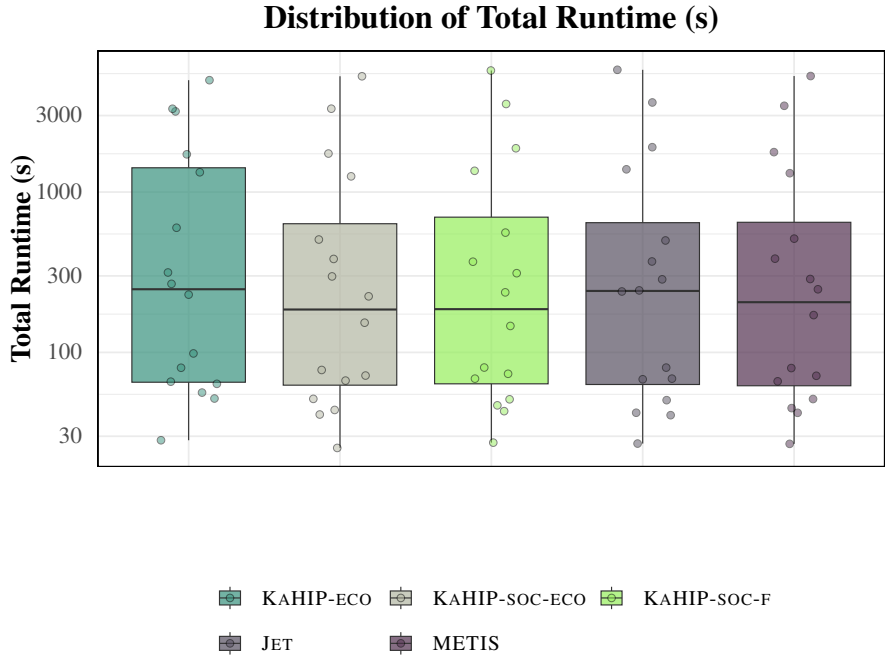

Figure 5.2: Distribution of Total Runtime Across All Algorithms. Each box represents the average runtime per graph.

Overall, these results confirm that investing in high-quality partitions, especially using KAHIP_ECO or KAHIP_SOC_ECO, yields lasting benefits that persist beyond the initial partitioning step. This is particularly relevant for iterative workloads, multi-run tuning, or applications where the partitioning cost can be amortized over many executions.

However, it is important to note that no single partitioner consistently delivers the best runtime across all graph instances. As shown in our results, performance can vary significantly depending on the structure of the input graph. In scenarios where partitioning is performed infrequently, such as in batch preprocessing or long-lived deployment settings, it may be beneficial to benchmark multiple partitioners for each graph and select the one that yields the best runtime performance for that instance.

Excluding partitioning time, high-quality partitioners consistently yield better performance. KAHIP_ECO and KAHIP_SOC_ECO achieved average runtime reductions of **7%** and **4%**, respectively, compared to METIS. These results suggest that in scenarios with repeated executions or infrequent re-partitioning, investing in stronger partitions delivers better amortized performance.

Table 5.5: Runtime of FASTAPSP excluding the partitioning phase. This isolates the impact of partition quality on core computation

Graph	METIS	KAHIP_SOC_ECO	KAHIP_ECO	KAHIP_SOC_F	JET
delaunay_n16	26.87	24.90	**24.63**	27.22	26.93
wing	44.90	43.05	**39.77**	46.54	41.96
fe_tooth	41.94	39.49	**38.70**	42.82	40.50
onera_dual	52.69	49.22	**47.69**	51.21	-
598a	71.37	69.49	**66.87**	73.27	68.51
luxembourg_osm	65.95	66.31	**63.23**	68.38	67.91
web-sk-2005	51.13	50.91	52.88	50.83	**50.29**
usroads-48	79.80	**77.24**	77.30	80.24	80.27
fe-ocean	286.80	296.53	289.81	311.45	**285.33**
ny-sorted	247.39	222.18	**214.68**	236.09	239.66
com-amazon	383.73	359.61	**351.72**	366.28	368.76
sc-pwtk	511.59	498.66	**472.98**	557.17	498.68
web-it-2004	170.17	146.80	174.38	**144.37**	243.36
soc-youtube	**360.66**	374.47	-	372.65	414.69
roadnet-pa	1,309.77	1,237.18	**1,174.26**	1,351.90	1,385.58
belgium_osm	1,772.40	1,723.21	**1,632.90**	1,871.05	1,903.97
road-roadnet-ca	3,447.72	3,283.23	**3,065.69**	3,534.29	3,618.52
netherlands_osm	5,287.34	5,233.47	**4,799.79**	5,717.03	5,784.70
Arithmetic Mean	**862.43**	**835.77**	**783.72**	**904.93**	**919.06**
Geometric Mean Speedup	**1.00x**	**1.04x**	**1.07x**	**0.99x**	**0.98x**

5.3 Optimized Execution Strategy

5.3.1 Overall Runtime Improvements

To evaluate the impact of our optimization strategy, we compare the total execution time of the original and optimized versions of FASTAPSP, both using the METIS partitioner. As shown in Figure 5.3, our optimized implementation achieves substantial performance improvements across nearly all graph sizes. The corresponding absolute runtimes for both versions are provided in Table 5.6. Most notably, we observe speedups of 6.74× and 6.68× on the two largest graphs, road-roadNet-CA and netherlands_osm, each with about two million vertices. These results support our hypothesis that minimizing CPU–GPU data transfers becomes increasingly critical as data volume grows. Smaller graphs such as luxembourg_osm and onera_dual still show improvements (1.22× and 1.15×, respectively), though the relative gains are less pronounced, as expected. This observation directly addresses research question **O1**, confirming that reducing data transfers improves runtime performance, particularly on large-scale graphs where communication overhead is a dominant cost. The only exception is web-sk-2005, which shows a slight performance regression (15% slower). Notably, this graph has an exceptionally low boundary-to-interior vertex ratio (122 boundary vs. 121,300 interior vertices),

indicating highly effective partitioning. As a result, the original version already incurs minimal data transfer overhead, limiting the benefits of our optimization. In contrast, graphs like fe_tooth, with a much higher ratio (7,921 boundary vs. 70,215 interior), experience significantly more data transfer and thus benefit more from transfer reduction. Overall, the average speedup across all tested graphs is 2.41×, validating the scalability and impact of our optimization strategy.

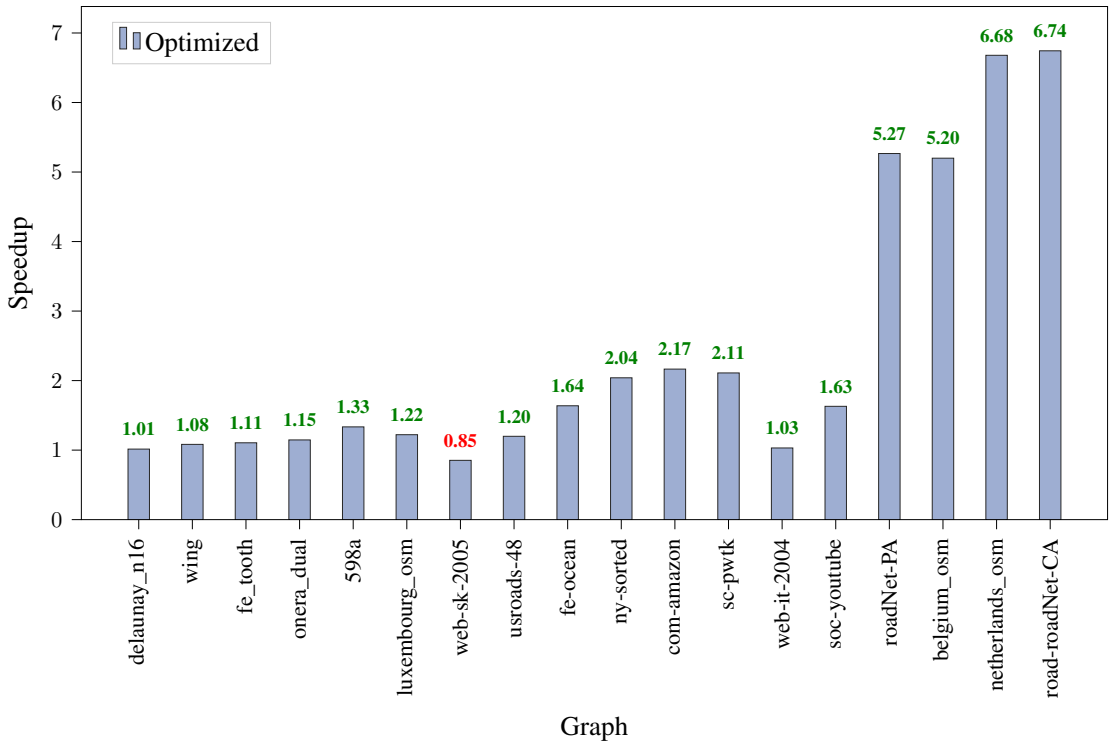

Figure 5.3: Speedup achieved by the optimized version of FASTAPSP over the original implementation across all graphs using the METIS partitioner.

Reducing CPU–GPU data transfers significantly improved total runtime, supporting hypothesis **O1**. The optimized version of FASTAPSP achieved an average speedup of **2.41**×, with gains reaching up to **6.74**× on the largest graphs. Smaller graphs showed more modest improvements (e.g., **1.22**× for luxembourg_osm). These results highlight transfer reduction as a critical factor for scalable graph processing.

5.3.2 Stage-wise Runtime Breakdown

To better understand where the observed performance improvements occur, Figures 5.4 and 5.5 present a breakdown of the total runtime by algorithm stage for representative small

Table 5.6: Absolute runtimes (in seconds) of the original and optimized versions of FASTAPSP on all graphs, using the METIS partitioner.

Graph	Old Time (s)	New Time (s)	Speedup
delaunay_n16	30.56	30.12	1.01x
wing	47.69	44.06	1.08x
fe_tooth	45.14	40.84	1.11x
onera_dual	59.99	52.32	1.15x
598a	96.99	72.69	1.33x
luxembourg_osm	81.37	66.60	1.22x
web-sk-2005	44.65	52.36	0.85x
usroads-48	85.64	71.45	1.20x
fe-ocean	477.10	291.29	1.64x
ny-sorted	482.79	236.64	2.04x
com-amazon	807.72	373.03	2.17x
sc-pwtk	1,521.32	720.95	2.11x
web-it-2004	250.78	243.27	1.03x
soc-youtube	839.38	514.98	1.63x
roadNet-PA	8,184.37	1,554.10	5.27x
belgium_osm	9,200.81	1,769.67	5.20x
road-roadNet-CA	32,677.72	4,845.67	6.74x
netherlands_osm	35,261.54	5,279.40	6.68x
Overall Average	5,010.86	903.30	2.41x

and large graphs, respectively. These results address research question **O2**, which asks where in the algorithmic pipeline the performance gains originate. As expected, the runtime of the three core computational phases-SSSP, Floyd-Warshall, and Min-Plus-remains virtually unchanged between the original and optimized versions, since the underlying GPU kernels are identical.

Among these stages, the SSSP phase consistently accounts for the largest share of total runtime. This dominance is already evident in small graphs (Figure 5.4), and becomes increasingly pronounced in larger inputs (Figure 5.5). Figure 5.6 further illustrates this behavior by highlighting the scaling trends of each stage across the entire dataset.

This trend can be attributed to two key factors. First, larger graphs are partitioned into a greater number of subgraphs, which increases the boundary-to-interior vertex ratio. Since SSSP is executed from each boundary vertex to all other vertices, a higher number of boundary vertices leads to more shortest-path computations. Second, the overall graph size increases, leading to longer traversal paths and higher memory demand during execution. In contrast, the Floyd-Warshall and Min-Plus stages operate on local subgraphs, and their runtime scales less aggressively with global graph size.

These findings reinforce that the observed performance improvements stem not from modifications to the GPU kernels themselves, but from architectural changes-namely, the restructuring of data transformation routines into dedicated GPU kernels and the reduction of redundant data transfers between CPU and GPU. These aspects are examined in more detail in the following sections.

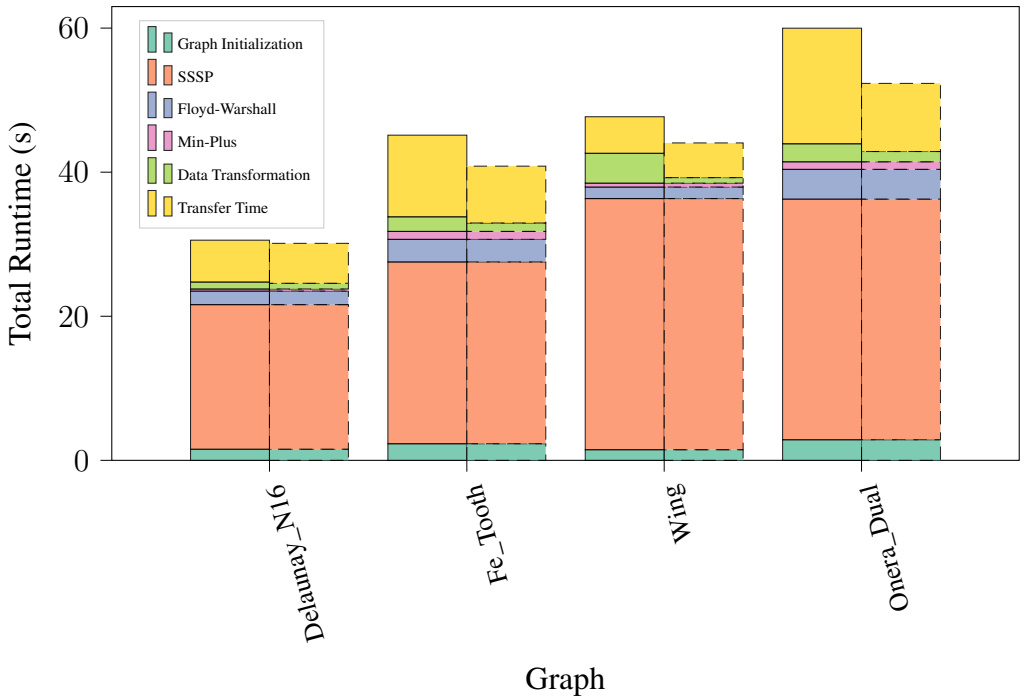

Figure 5.4: Runtime breakdown by processing stage for four representative small graphs. Bars are stacked by component: Graph Initialization (1), SSSP (2), Floyd-Warshall (3), Min-Plus (4), Data Transformation (5), and Transfer Time (6). The SSSP phase dominates overall runtime even at small scales, while other components remain comparatively minor.

Runtime analysis reveals that the optimization gains in FastAPSP stem not from changes in the GPU kernels, but from reduced data transfers and improved stage transitions, confirming **O2**.

5.3.3 Impact of Reduced Data Transfers

To isolate and quantify the impact of reduced data movement between host and device, Figure 5.7 presents a comparison of the time spent on CPU to GPU data transfers in both the original and optimized implementations of FastAPSP.

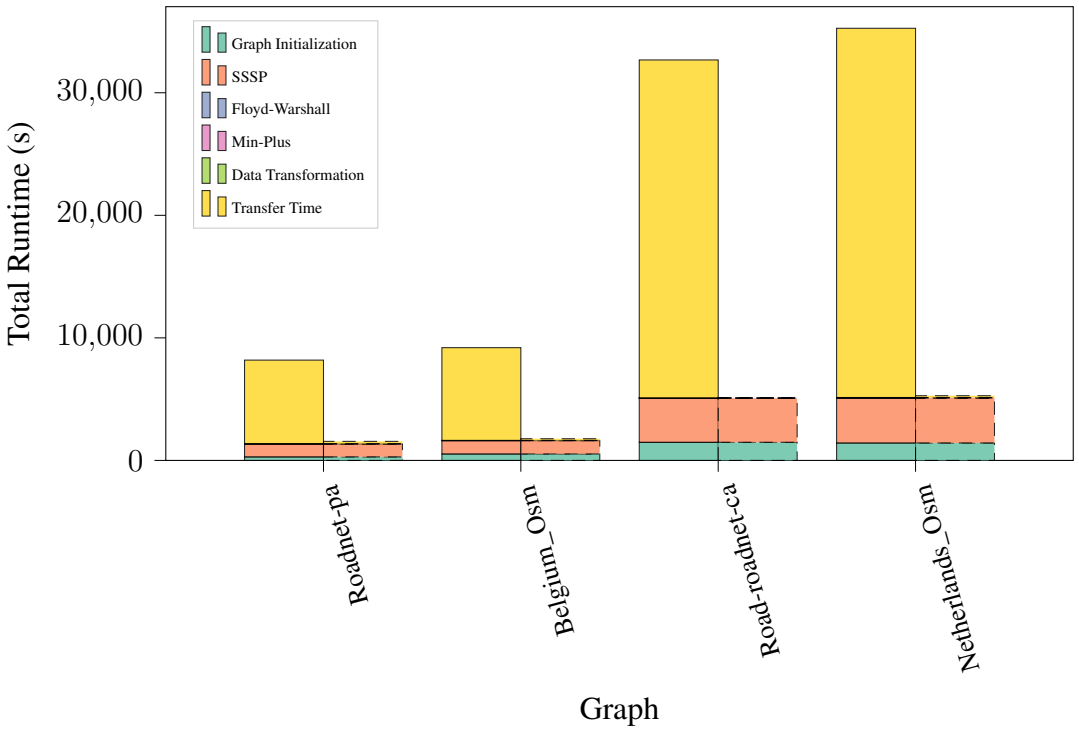

Figure 5.5: Runtime breakdown by processing stage for four representative large graphs. Bars are stacked by component: Graph Initialization (1), SSSP (2), Floyd-Warshall (3), Min-Plus (4), Data Transformation (5), and Transfer Time (6). On large graphs, Transfer Time is the most dominant component, followed by SSSP, highlighting the critical cost of CPU–GPU communication at scale.

A clear trend emerges: the performance gap between the two versions grows with graph size. While the difference is relatively small for smaller graphs, it increases substantially on larger graphs, where high data volumes make memory transfers a dominant cost factor. This observation aligns with the overall runtime improvements shown in Figure 5.3, and highlights the critical role of communication overhead in large scale graph processing.

In our optimized version, intermediate data is retained on the GPU across computation stages, which avoids redundant data transfers between the host and device. As a result, the total time spent on data transfers is significantly reduced, particularly for large graphs where communication overhead would otherwise dominate. Although transfer time remains a substantial portion of overall runtime, it is dramatically lower compared to the original implementation and constitutes the primary source of the observed speedup.

In addition, addressing **O3**, the performance improvements come at a modest memory cost: on average, the peak GPU memory usage of the optimized version is only 4% higher than that of the original implementation. This small overhead validates the feasibility of our design and demonstrates that reducing communication costs does not require sacrificing device memory efficiency.

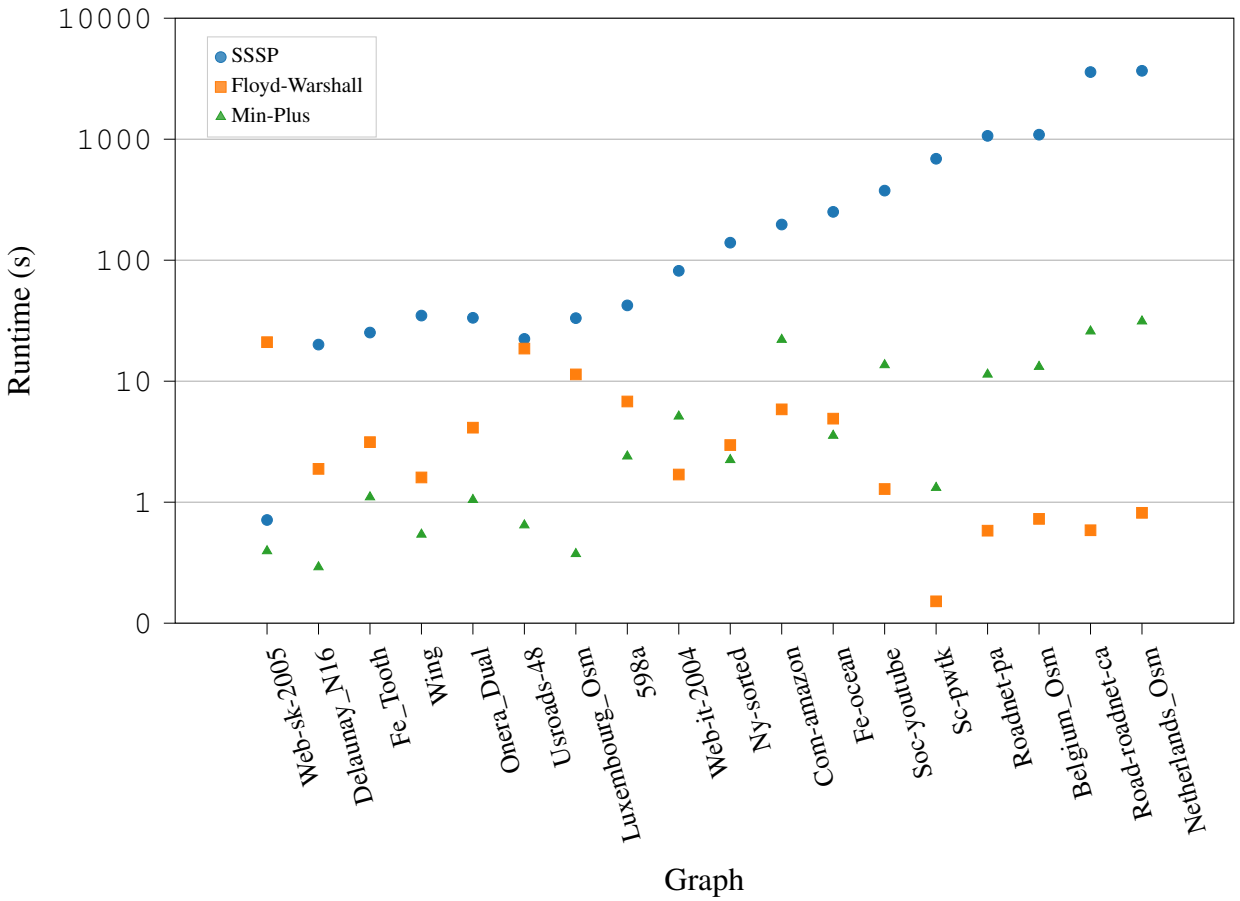

Figure 5.6: Scaling behavior of the SSSP, Floyd-Warshall, and Min-Plus stages across all input graphs, sorted by graph size. The y-axis uses a logarithmic scale to better reflect differences across several orders of magnitude. Runtime for each stage increases differently with input scale.

Optimizing data locality by retaining intermediate results on the GPU reduced data transfers substantially, directly addressing **O3**. For large graphs, transfer time was reduced by up to two orders of magnitude. Despite this, our optimized version used only **4%** more peak GPU memory on average, confirming that significant transfer reduction can be achieved with minimal memory overhead.

5.3.4 Acceleration via GPU Kernel Transformations

A further source of performance improvement in the optimized version stems from restructuring previously CPU-based data transformation routines into dedicated CUDA kernels. This change yields two main benefits: first, it eliminates unnecessary data transfers by

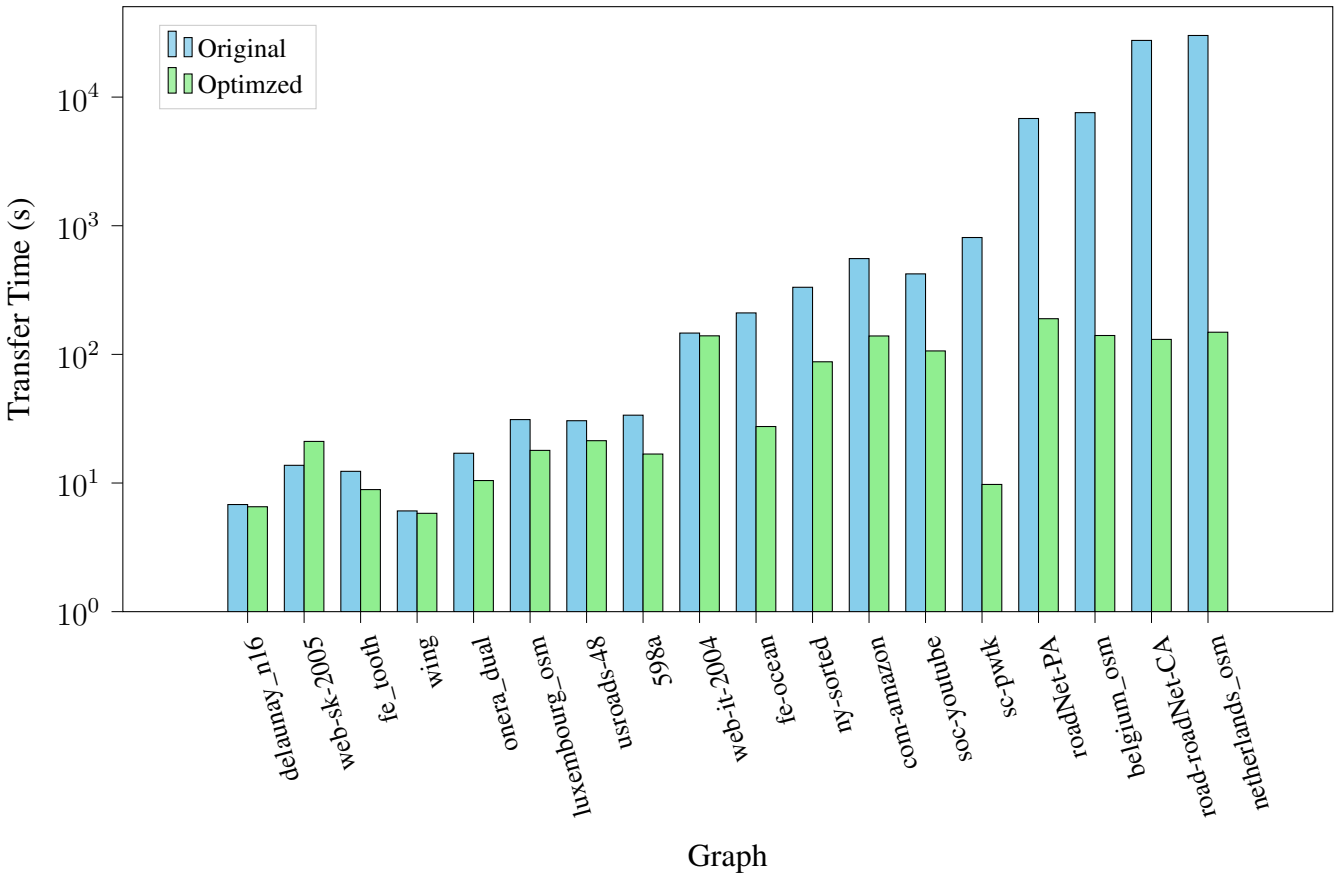

Figure 5.7: Comparison of time spent on CPU–GPU data transfers in the original and optimized implementations. The y-axis is shown on a logarithmic scale to better visualize the wide range of transfer times across graph sizes. The optimized version significantly reduces transfer overhead, especially on larger graphs.

executing transformations directly on the GPU; second, it enables parallel execution of these routines, thereby leveraging the computational capabilities of the GPU more effectively. As illustrated in Figure 5.8, the runtime of these transformation steps is significantly reduced in the optimized implementation. While each kernel contributes only a modest speedup in isolation, their cumulative impact is substantial and plays a central role in reducing total execution time. To better reflect the factors influencing transformation behavior, the graphs in Figure 5.8 are sorted by average subgraph size.

An interesting observation emerges when examining the scaling behavior of these transformations across different graph sizes. Intuitively, one might expect that larger graphs would lead to longer transformation times due to increased data volume. However, the experimental results reveal a more nuanced trend: transformation times do not grow significantly with graph size.

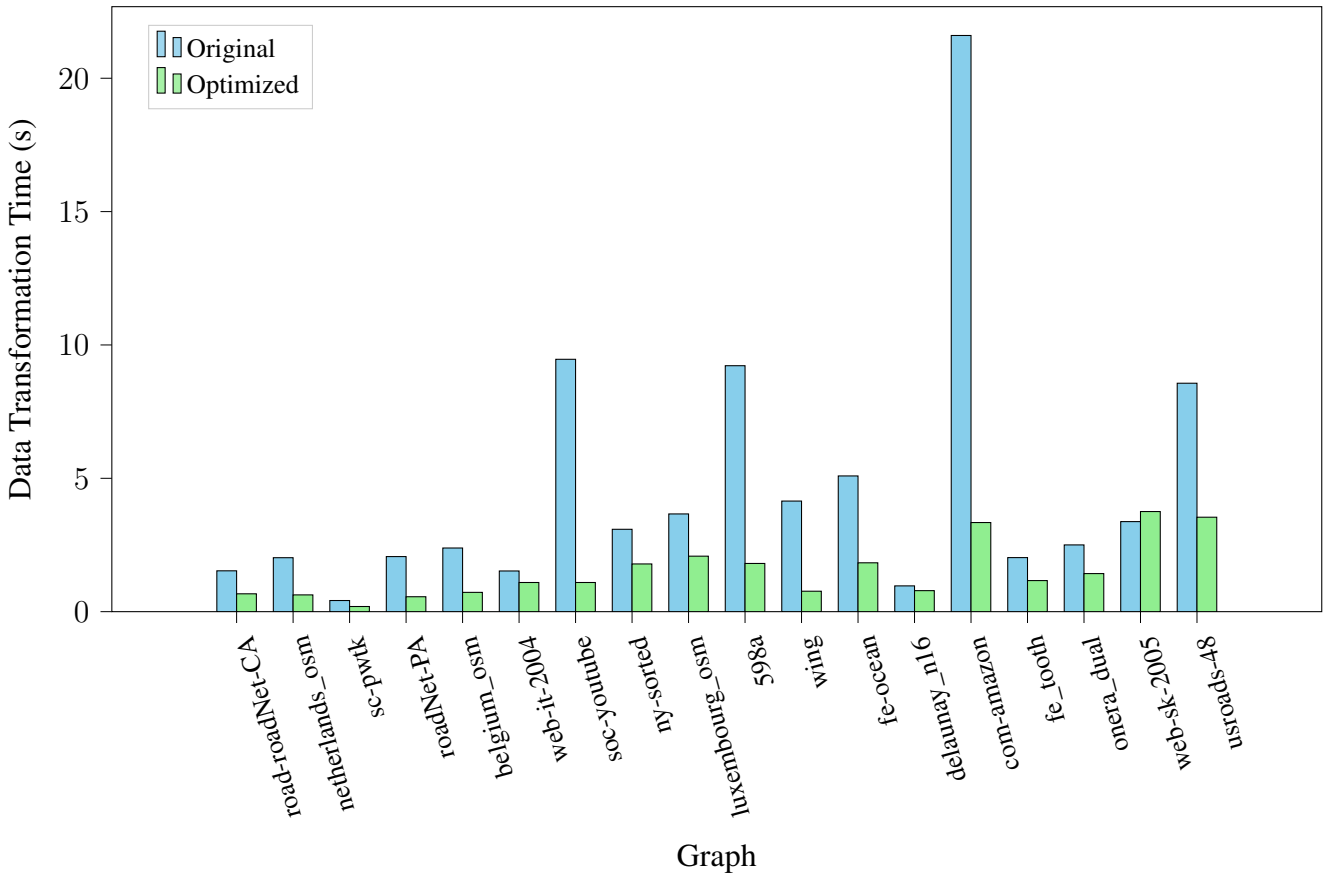

Figure 5.8: Comparison of data transformation time between the original (CPU-based) and optimized (GPU-based) implementations. GPU kernels enable parallel processing and eliminate transfer latency. Graphs are sorted by average subgraph size, which influences the number and size of kernel invocations.

This behavior was explained by the relationship between global graph size and subgraph structure. In our experiments, larger graphs are decomposed into a greater number of subgraphs in order to fit within GPU memory constraints. Conversely, smaller to medium-sized graphs can be partitioned into fewer, but larger subgraphs. Since the transformation kernels are invoked once per subgraph, the average size of a subgraph, rather than the total graph size, becomes the dominant factor influencing transformation time.

These results underscore the effectiveness of our approach. By executing transformation routines in parallel on the GPU and minimizing inter-device data movement, we eliminate a key bottleneck and achieve a scalable reduction in runtime. This directly answers research question **O4**, demonstrating that GPU-based transformation kernels contribute meaningfully to the overall speedup.

Transforming CPU-based data preparation steps into GPU kernels provided meaningful performance gains, affirming hypothesis **O4**. While each individual kernel yielded only modest improvements, executing preparation directly on the GPU eliminated the need for CPU–GPU transfers by keeping the data on the GPU. These transformations played a key role in achieving the overall speedups.

CHAPTER 6

Discussion

6.1 Conclusion

The All-Pairs Shortest Path (APSP) problem is a fundamental building block in many application domains, including transportation networks, social network analysis, and scientific simulations. As graph datasets grow in size and complexity, efficiently solving APSP at scale, especially on GPU architectures, has become increasingly critical.

This thesis introduced two key improvements to the FASTAPSP algorithm: the use of alternative partitioning strategies and the optimization of CPU–GPU memory interaction. In the first part, we achieved a **2.4**× average speedup over the original implementation of FASTAPSP by minimizing data transfers between CPU and GPU and parallelizing transformation routines using dedicated CUDA kernels. These architectural changes led to improvements of up to **6.7**× on the largest graphs. These gains were achieved with only a modest increase in GPU memory consumption, confirming the efficiency and scalability of the optimized design.

In the second part, we integrated several state-of-the-art graph partitioners into FASTAPSP, including KAHIP variants and the GPU-based JET partitioner. This resulted in a **2.7%** additional reduction in total runtime compared to METIS. The evaluation showed that partition quality, particularly the boundary-to-interior vertex ratio, has a significant impact on runtime efficiency. When partitioning time was excluded, simulating a realistic multi-run scenario, we achieved a **7%** speedup using a KAHIP variant, highlighting the amortized benefits of high-quality partitions.

Overall, the findings demonstrate that combining high-quality partitioning with a memory-efficient GPU execution strategy yields substantial and reliable performance gains for large-scale APSP computations.

6.2 Future Work

While this thesis introduces several significant optimizations to the FASTAPSP algorithm, multiple promising directions remain open for future research. One natural extension is to evaluate the scalability of the algorithm beyond the four GPUs used in our experiments. Expanding to larger multi-GPU or multi-node environments would help assess performance limits and potential bottlenecks in more distributed settings.

As graph sizes continue to grow, another major challenge is overcoming the memory limitations of a single GPU. Although increasing the number of partitions (k) can reduce per-partition memory usage, this approach offers diminishing returns. An alternative direction is to adopt a distributed GPU memory model, where graph data is partitioned across multiple GPUs or nodes. This would require implementing efficient GPU-to-GPU communication protocols such as NVLink or NCCL, and designing algorithms that minimize inter-device data movement.

In addition to scaling memory capacity, reducing CPU–GPU transfer overhead remains important. Currently, the input graph is constructed on the CPU and transferred to the GPU. Building the graph directly on the GPU could reduce initialization costs and better exploit memory locality. This is particularly relevant in scenarios where graph data is produced on the GPU or arrives in a streaming fashion.

Additionally, the partitioning strategy itself could be made adaptive. Incorporating runtime feedback into partitioner selection, such as choosing between METIS, KaHIP, or JET based on structural properties like degree distribution or sparsity, could improve performance across diverse graphs. Further, dynamically adjusting the partition count (k) at runtime in response to memory constraints or workload imbalance may lead to more efficient resource utilization and better overall throughput.

Finally, an important direction for future work is the evaluation of FASTAPSP on weighted graphs. While the current evaluation focuses on unweighted graphs, introducing edge weights would affect both shortest-path computations and the partitioning strategy. In particular, partitioners would need to account for weighted edge cuts, where minimizing total cut weight may conflict with minimizing the boundary-to-interior vertex ratio. This could lead to new trade-offs in partition quality, requiring more nuanced strategies for balancing computational load and minimizing inter-partition communication. Exploring how different partitioners adapt to weighted inputs could yield further improvements or highlight limitations of current heuristics.

APPENDIX A

Appendix

A.1 Partition Counts per Graph

Table A1 lists the number of partitions k used for each graph instance in our evaluation. These were selected based on the input size, ensuring fair comparisons between partitioners and adherence to GPU memory limitations.

Table A1: Number of partitions k used per graph.

Graph	**Partition Counts (k)**
delaunay_n16	8, 16, 32
wing	8, 16, 32
fe_tooth	8, 16, 32
onera_dual	8, 16, 32
598a	16, 32, 64
luxembourg_osm	16, 32, 64
web-sk-2005	16, 32, 64
usroads-48	16, 32, 64
fe-ocean	16, 32, 64
ny-sorted	64, 128
com-amazon	64, 128
sc-pwtk	128, 256, 512
web-it-2004	128, 256, 512
soc-youtube	128, 256, 512
roadnet-pa	1,024, 2,048
belgium_osm	1,024, 2,048
road-roadnet-ca	4,096
netherlands_osm	4,096

Zusammenfassung

Das All-Pairs-Shortest-Path-Problem (APSP) stellt eine grundlegende Herausforderung der Graphentheorie dar und findet breite Anwendung in verschiedenen Bereichen. FASTAPSP ist ein moderner, GPU-beschleunigter Algorithmus, der darauf ausgelegt ist, das APSP-Problem effizient auf großskaligen Graphen zu lösen. Seine Leistung wird jedoch durch die Qualität der Graphpartitionierung und durch umfangreiche CPU–GPU-Datenübertragungen begrenzt. In dieser Arbeit präsentieren wir zwei komplementäre Erweiterungen des FASTAPSP-Algorithmus. Erstens optimieren wir die Ausführungspipeline auf der GPU, indem wir Datenumwandlungen in GPU-Kernels überführen und kostspielige CPU-GPU-Datenübertragungen minimieren. Diese architektonische Optimierung führt zu deutlichen Leistungssteigerungen mit einer durchschnittlichen Beschleunigung um den Faktor 2,41 und einem maximalen Gewinn von bis zu 6,7 bei den größten Graphen. Zweitens evaluieren und integrieren wir fortgeschrittene Graphpartitionierer, darunter KAHIP sowie den GPU-basierten JET-Partitionierer, um die Qualität der Partitionierung zu verbessern und den Rechenaufwand zu verringern. Dies führt zu einer zusätzlichen Reduktion der Laufzeit um 2,7%, wenn die KAHIP-Variante "social" anstelle des Standardpartitionierers METIS verwendet wird.

Bibliography

[1] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[2] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[3] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[4] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. 2016.

[5] bwUniCluster 2.0. High performance computing for universities in badenwuerttemberg.

[6] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *ACM Comput. Surv.*, 55(12):253:1–253:38, 2023.

[7] Jie Cheng. CUDA by example: An introduction to general-purpose GPU programming. *Scalable Comput. Pract. Exp.*, 11(4), 2010.

[8] Cédric Chevalier and François Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *CoRR*, abs/0907.1375, 2009.

[9] Lung-Sheng Chien. Hand tuned sgemm on gt200 gpu. *Technical Report, Tsing Hua University*, 2010.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[11] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 349–359. IEEE Computer Society, 2014.

[12] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[13] Hristo N. Djidjev, Guillaume Chapuis, Rumen Andonov, Sunil Thulasidasan, and Dominique Lavenier. All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters. *J. Parallel Distributed Comput.*, 85:91–103, 2015.

[14] Hristo N. 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, Phoenix, AZ, USA, May 19-23, 2014*, pages 360–369. IEEE Computer Society, 2014.

[15] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distributed Comput.*, 74(12):3202–3216, 2014.

[16] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In James S. Crabbe, Charles E. Radke, and Hillel Ofek, editors, *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*, pages 175–181. ACM/IEEE, 1982.

[17] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[18] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Catherine C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008.

[19] Michael S. Gilbert, Kamesh Madduri, Erik G. Boman, and Siva Rajamanickam. Jet: Multilevel graph partitioning on graphics processing units. *SIAM J. Sci. Comput.*, 46(5):700, 2024.

[20] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel flow-based hypergraph partitioning. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, volume 233 of *LIPIcs*, pages 5:1–5:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[21] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable shared-memory hypergraph partitioning. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 16–30. SIAM, 2021.

[22] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Shared-memory n-level hypergraph partitioning. In Cynthia A. Phillips and Bettina Speckmann, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*, pages 131–144. SIAM, 2022.

[23] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High Performance Computing - HiPC 2007, 14th International Conference, Goa, India, December 18-21, 2007, Proceedings*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer, 2007.

[24] Ramakrishnan Kannan, Piyush Sao, Hao Lu, Drahomira Herrmannova, Vijay Thakkar, Robert M. Patton, Richard W. Vuduc, and Thomas E. Potok. Scalable knowledge graph analytics at 136 petaflop/s. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 6. IEEE/ACM, 2020.

[25] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(1):69–79, 1999.

[26] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distributed Comput.*, 48(1):96–129, 1998.

[27] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In Mary Jane Irwin, editor, *Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999*, pages 343–348. ACM Press, 1999.

[28] Gary J. Katz and Joseph T. Kider Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the EUROGRAPHICS/ACM SIGGRAPH Conference on Graphics Hardware 2008, Sarajevo, Bosnia and Herzegovina, 2008*, pages 47–55. Eurographics Association, 2008.

[29] JEREMY KEMP. *All-Pairs Shortest Path Algorithms Using CUDA*. PhD thesis, Durham University, 2012.

[30] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49(2):291–307, 1970.

[31] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandström. The suitesparse matrix collection website interface. *J. Open Source Softw.*, 4(35):1244, 2019.

[32] Junjie Lai and André Seznec. Performance upper bound analysis and optimization of SGEMM on fermi and kepler gpus. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 4:1–4:10. IEEE Computer Society, 2013.

[33] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[34] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Math.*, 6(1):29–123, 2009.

[35] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel G. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–8. IEEE, 2009.

[36] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. A pattern based algorithmic autotuner for graph processing on gpus. In Jeffrey K. Hollingsworth and Idit Keidar, editors, *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 201–213. ACM, 2019.

[37] Ulrich Meyer and Peter Sanders. [delta]-stepping: a parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[38] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In Constantine Dovrolis and Matthew Roughan, editors, *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference, IMC 2007, San Diego, California, USA, October 24-26, 2007*, pages 29–42. ACM, 2007.

[39] Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara. A task parallel algorithm for finding all-pairs shortest paths using the GPU. *Int. J. High Perform. Comput. Netw.*, 7(2):87–98, 2012.

[40] Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos, and Arturo González-Escribano. A new gpu-based approach to the shortest path problem. In *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*, pages 505–511. IEEE, 2013.

[41] Héctor Ortega Arranz, Yuri Torres de la Sierra, Diego Rafael Llanos Ferraris, Arturo González Escribano, et al. The all-pair shortest-path problem in shared-memory heterogeneous systems. *High-Performance Computing on Complex Environments*, pages 283–299, 2014.

[42] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[43] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2011.

[44] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[45] Piyush Sao, Ramakrishnan Kannan, Prasun Gera, and Richard W. Vuduc. A supernodal all-pairs shortest path algorithm. In Rajiv Gupta and Xipeng Shen, editors, *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, pages 250–261. ACM, 2020.

[46] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, Muhammad Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 979–990. ACM, 2014.

[47] Jacob Scott, Trey Ideker, Richard M. Karp, and Roded Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. *J. Comput. Biol.*, 13(2):133–144, 2006.

[48] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library - User Guide and Reference Manual*. C++ in-depth series. Pearson / Prentice Hall, 2002.

[49] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.*, 12(3):66–73, 2010.

[50] Guangming Tan, Linchuan Li, Sean Triechle, Everett H. Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of DGEMM on fermi GPU. In Scott A. Lathrop,

Jim Costa, and William Kramer, editors, *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pages 35:1–35:11. ACM, 2011.

[51] Shoshana Vasserman, Michal Feldman, and Avinatan Hassidim. Implementing the wisdom of waze. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 660–666. AAAI Press, 2015.

[52] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: a high-performance graph processing library on the GPU. In Rafael Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 11:1–11:12. ACM, 2016.

[53] Jack Wells, Buddy Bland, Jeff Nichols, Jim Hack, Fernanda Foertter, Gaute Hagen, Thomas Maier, Moetasim Ashfaq, Bronson Messer, and Suzanne Parete-Koon. Announcing supercomputer summit. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2016.

[54] Shaofeng Yang, Xiandong Liu, Yunting Wang, Xin He, and Guangming Tan. Fast all-pairs shortest paths algorithm in large sparse graph. In *Proceedings of the 37th International Conference on Supercomputing, ICS 2023, Orlando, FL, USA, June 21-23, 2023*, pages 277–288. ACM, 2023.

[55] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. Understanding the GPU microarchitecture to achieve bare-metal performance tuning. In Vivek Sarkar and Lawrence Rauchwerger, editors, *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 31–43. ACM, 2017.

[56] Peixiang Zhao, Jiawei Han, and Yizhou Sun. P-rank: a comprehensive structural similarity measure over information networks. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy Lin, editors, *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 553–562. ACM, 2009.