# Scaling and Selecting GPU Methods for All Pairs Shortest Paths (APSP) Computations

Yang Xia
*Computer Science and Engineering*
*The Ohio State University*
xia.425@osu.edu

Peng Jiang
*Computer Science Department*
*University of Iowa*
peng-jiang@uiowa.edu

Gagan Agrawal
*Computer and Cyber Sciences*
*Augusta University*
gagrawal@augusta.edu

Rajiv Ramnath
*Computer Science and Engineering*
*The Ohio State University*
ramnath@cse.ohio-state.edu

***Abstract*— All Pairs Shortest Path (APSP) is one of the graph problems where the output size is significantly larger than the input size. This paper examines the issues in scaling GPU implementations for this problem beyond the memory limits. Because the existing (in-core) methods offer a complex trade-off between the overall computation complexity and the available parallelism, choosing the best out-of-core version for a given matrix is challenging. We develop three efficient out-of-core implementations, which are based on the blocked Floyd-Warshall algorithm, Johnson's algorithm, and the boundary algorithm, respectively. Next, we develop a methodology to select the best implementation for a given graph. Experimental results show that compared with an efficient multi-core APSP implementation, the out-of-core version achieves speedups of 8.22 to 12.40 for graphs with a small separator, and speedups of 2.23 to 2.79 for other sparse graphs, and our models can select the best implementation in most cases.**

## I. Introduction

As more computing power is shifting to accelerators, a large volume of recent efforts (see representative examples [16], [17], [22], [26], [27], [37], [25], [3]) have shown that it is promising to exploit GPUs to accelerate graph processing. However, together with the high peak processing power of GPUs, comes a significant limitation, which is that the device memory size on GPUs is only a fraction of advanced CPUs' memory sizes. Thus, GPU processing is often viewed as a solution only for small and medium sized graphs. One approach, however, can be developing *out-of-core* GPU implementations. There do exist several such efforts [30], [23], [32], though focus on (more common) graph problems where the input size is larger than the output size, and the out-of-core version processes an input graph that cannot fit in memory.

All Pair Shortest Path (APSP) belongs to a different class of graph problems, where the output distance (dense) matrix is larger (often by orders of magnitude!) than the size of the adjacency (sparse) matrix representing the input graph. APSP involves finding the shortest paths between all pairs of vertices in a weighted graph. It is an important problem with applications in diverse areas such as traffic simulation [2], databases [9], Internet routing [29], sensor networks [18], and others. The time complexity of APSP algorithms can be very high, especially when parallelism has to be exposed (for example, the time complexity is cubical in terms of the number of vertices with the Floyd-Warshall algorithm [10]). Thus, the major overhead of APSP can either be the data transfers or the computation itself, depending upon the problem size, characteristics of the matrix, and the algorithm used.

We observe that not only it is challenging to create efficient out-of-core versions of algorithms, but the optimal choice may depend upon the nature of the graph itself. Previous research works on GPU acceleration of APSP mostly proposed to adopt the Floyd-Warshall algorithm [16], [35], [20] and demonstrated high efficiency with small graph sizes. However, the Floyd-Warshall algorithm's high complexity can be detrimental to overall efficiency when large graphs are involved. On the other hand, using another approach, Johnson's algorithm [10], as the basis is also challenging because we are typically only able to run a small batch of SSSP instances in parallel due to the GPU memory size limits, limiting parallelism.

In this work, we explore the design space of out-of-core processing solutions to the APSP problem. We first present three out-of-core implementations for APSP. We also introduce several optimizations to improve their performance – specifically, using dynamic parallelism for Johnson's algorithm, batching small data transfers in the boundary algorithm, and exploiting asynchronous data transfers for all implementations. Our analysis on understanding the relative performance of these three versions leads to the following observations. For graphs with a small separator, the boundary algorithm usually achieves the best performance because of regular computation patterns and resulting memory accesses. For other sparse graphs, the maximal number of components allowed in out-of-core boundary algorithm implementation is small. Thus, Johnson's algorithm achieves a better performance. However, as the densities of the graphs increase, the workloads of Johnson's algorithm also increase. When graphs are sufficiently dense, the blocked Floyd-Warshall algorithm delivers the best performance.

These observations are further coded into a methodology to automatically select the best version for each graph. For Johnson's algorithm, we observe that the execution times of different SSSP instances or different batches are similar. Thus,

execution times can be predicted by running several (randomly selected) batches of SSSP instances. For the blocked Floyd-Warshall algorithm and the boundary algorithm, we estimate the execution times based on the number of vertices. We also propose simple (low overhead) heuristics based on the density values of the graphs. Overall, to the best of our knowledge, this paper is the first in reporting out-of-core versions of APSP and a methodology for choosing the best among multiple candidate versions.

Our experiments evaluate different versions using a set of matrices from SuiteSparse matrix collection [12] and other graphs generated using the R-MAT [8], and lead to the following observations. First, our out-of-core versions can achieve significant performance improvement over an efficient multi-core CPU implementation. For graphs with a small separator, the speedups are from 8.22 to 12.40, whereas for other sparse graphs, it delivers speedups in the range from 2.23 to 2.79. Thus, we demonstrate that GPU processing can be attractive even for large graphs. Second, compared with other efficient implementations from the literature [31], our implementations achieve speedups in the range 4.70-69.2 in almost all cases. Finally, we show that our selector can always select the most efficient implementation for our set of graphs based on our cost models.

## II. Background

Let $G = (V, E)$ be a weighted graph, where $V$ is the set of vertices and $E$ is the set of edges. Suppose that $n = |V|$ is the number of vertices, $m = |E|$ is the number of edges. Let $w[i][j]$ be the weight of the edge from vertex $v_i$ to vertex $v_j$. The goal of the all-pairs shortest paths (APSP) problem is to compute an $n \times n$ matrix $dist$, where $dist[i][j]$ denotes the shortest distance from vertex $v_i$ to vertex $v_j$. There are two families of algorithms to solve the APSP problems, building on the Floyd-Warshall algorithm and Johnson's algorithm, respectively [10].

### A. The Floyd-Warshall Algorithm

This algorithm initializes the matrix $dist$ with the input weights for the edges (which will be $\infty$ when there are no edges between the two vertices). The algorithm has an outer loop with the number of iterations equal to the number of vertices in the graph. During the $k^{th}$ iteration of this outer loop, we check each pair of vertices $v_i$ and $v_j$. If there is a shorter path between them via the intermediate vertex $v_k$, then the algorithm updates $dist[i][j]$. It has been shown that the algorithm gets the final shortest distances between each pair of vertices after $n$ iterations [10].

To improve the performance of the Floyd-Warshall algorithm, existing works [20] applied tiling techniques to get a blocked version. Initially, the matrix $dist$ is partitioned into $num\_b \times num\_b$ blocks. Each block is a sub-matrix with size $b \times b$, where $b = n/num\_b$. The algorithm takes $num\_b$ iterations to compute the results of all blocks. During the $k^{th}$ iteration, it follows three major steps. Let $A(i, j)$ denote the $(i, j)$ block of the matrix $dist$. In the first step, it computes APSP on a diagonal block, i.e. $A(k, k)$ using the original Floyd-Warshall algorithm because its computation is independent of any other block. In the second step, it updates blocks $A(k, i)$ and $A(i, k)$, where $i \neq k$. The computation of these blocks depends on the diagonal block $A(k, k)$, and they are updated by applying *MinPlus Multiply* [21] with $A(k, k)$. In the final step, it updates the remaining blocks $A(i, j)$, where $i \neq k$ and $j \neq k$.

### B. Johnson's algorithm and SSSP implementations

In a very different approach, Johnson's algorithm [10] solves APSP by running the single-source shortest path (SSSP) over each source node in the graph. In the classic Johnson's algorithm, Dijkstra's algorithm for each source node [10] is executed. The key idea is to maintain a priority queue of the vertices, prioritized by their shortest distances discovered so far. For each iteration, the method extracts the top vertex on the queue, which is also the vertex with the shortest distance from the source vertex. Then, it *relaxes* all edges leaving from this vertex – the relax operation on an edge $(u, v)$ updates the distance value of $v$ if it has a shorter distance from the vertex $u$. When the implementation is done using Fibonacci heaps or relaxed heaps, the time complexity of Dijkstra's algorithm is $O(nlogn + m)$, which is the lowest known complexity for SSSP.

Dijkstra's algorithm's only exposed parallelism is in the form of concurrent processing of the edges at the single vertex at the top of the queue. On the other hand, in Bellman-Ford algorithm, during each iteration, we check each edge $(u, v)$, and perform *relax* operations. This is repeated until the distances converge. In the worst case, this operation repeats $n - 1$ times. Obviously, the *relax* operations among different vertices are independent and easy to parallelize. However, the sequential time complexity is O($nm$), which is much higher than that of Dijkstra's algorithm.

The *delta-stepping* algorithm [24] is a generalization of both Dijkstra's algorithm and the Bellman-Ford algorithm. It allows more parallelism than Dijkstra's while providing better work efficiency than the Bellman-Ford algorithm by using a coarse-grained priority queue. The vertices whose distances lie at a specific range are grouped in the same bucket. The range in each bucket is called *delta*. Delta-stepping then processes vertices from the bucket with smallest distances in parallel. When this bucket becomes empty, the next bucket is processed, and so on.

In this work, we adopt Near-Far optimization [11] for SSSP implementation, which is a simplification of the delta-stepping algorithm that is considered efficient on GPUs. The work queue here is divided into two queues: $Near\_Queue$ and $Far\_Queue$. Initially, $Near\_Queue$ contains only the source vertex and $Far\_Queue$ is empty. During each iteration, the method traverses edges from the vertices in the $Near\_Queue$. For an edge $(u, v)$, where $u$ is a vertex from $Near\_Queue$, it not only performs the *relax* operation but also assigns vertex $v$ into the two queues – specifically, it appends $v$ to $Near\_Queue$ if the updated distance is smaller than $i\Delta$, otherwise, it assigns it to the $Far\_Queue$. Then, it begins the next iteration of processing the $Near\_Queue$. Once it runs out of vertices in the $Near\_Queue$, we swap $Near\_Queue$ and $Far\_Queue$, and continue the next iteration with an updated range $((i + 1)\Delta)$.

## III. Out-of-Core Approaches for APSP

This section shows a series of out-of-core methods to solve the APSP problem. We note that several research efforts [16],

[20] have proposed efficient implementations of the Floyd-Warshall algorithm on GPUs because of a large degree of exposed parallelism and memory efficiency. However, the time complexity of the Floyd-Warshall algorithm is as high as O($n^3$), clearly adding high overheads as the graph sizes increase. This is an important consideration as we develop our efficient out-of-core methods. On the other hand, Johnson's algorithm is work efficient, and it can also expose sufficient parallelism as different SSSP instances are independent. However, the GPU memory sizes limit the number of parallel SSSP instances.

### A. Out-of-core Floyd-Warshall Algorithm

**Algorithm 1** Out-of-Core Version of the Blocked Floyd-Warshall Algorithm

**Input**: $A(i, j)$ - weight matrix of block $(i, j)$
$n_d$-the number of blocks in each dimension.
**Output**: $A(i, j)$ - shortest distance matrix of block $(i, j)$

**for** $k = 0; k < n_d; k++$ **do**
  ▷ Stage 1
  A(k,k) = Blocked Floyd-Warshall(A(k,k))
  Transfer results of A(k,k) back to CPU
  ▷ Stage 2
  **for** $i = 0; i < n_d; i++$ **do**
    A(k,i) = min(A(k,i), A(k,k) + A(k,i))
    Transfer results of A(k,i) back to CPU
  **end for**
  **for** $i = 0; i < n_d; i++$ **do**
    A(i,k) = min(A(i,k), A(i,k) + A(k,k))
    Transfer results of A(i,k) back to CPU
  **end for**
  ▷ Stage 3
  **for** $i = 0; i < n_d; i++$ **do**
    **for** $j = 0; j < n_d; j++$ **do**
      A(i,j) = min(A(i,j),A(i,k)+A(k,j))
      Transfer results of A(i,j) back to CPU
    **end for**
  **end for**
**end for**

A straightforward implementation of APSP is to adapt the blocked version of the Floyd-Warshall algorithm to the out-of-core case. Algorithm 1 briefly shows the procedure of this implementation. In this case, the matrix is partitioned into $n_d \times n_d$ blocks, so that each block is as large as possible while it can still fit into GPU's memory. Then, for each iteration, we apply the same three stages to update the values of blocks. For diagonal blocks that are updated during the first stage, we apply the blocked version of the Floyd-Warshall algorithm recursively. For the remaining blocks, we implement matrix multiplication (*MinPlus Multiplication*) with tiling techniques to utilize shared memory efficiently [14]. We transfer the data back to the CPU after updating each block – the amount of data transferred is $n_d \times n^2$ because we transfer all blocks during each iteration.

### B. Out-of-Core Johnson's Algorithm

A naive out-of-core implementation of Johnson's algorithm involves running an SSSP kernel for each source vertex. Here, we propose a *batch* implementation to make efficient use of parallelism on GPUs. The overall procedure is shown in Algorithm 2. It first determines the maximal number of concurrent runs of SSSP on GPUs based on the GPU's memory limits. We denote it as $bat$. Let $L$ be the GPU memory size, the memory required for storing the graph be $S$, and further, each SSSP instance requires $c \times m$ storage for the work queues, where $c$ is a constant, $m$ is the number of edges. Then

$$bat = (L - S)/(c \times m).$$

Assume that the number of batches is $n_b$, where $n_b = n/bat$, then the procedure of the *batch* implementation is composed of $n_b$ iterations. During each iteration, we run $bat$ SSSPs in a single kernel. We denote the kernel function as ***MSSP*** since it essentially computes the shortest paths from multiple sources. We assign each SSSP instance to one thread block so that we are able to synchronize threads within one thread block, using $\_\_syncthreads()$ primitive provided by CUDA.

**Algorithm 2** Out-of-Core Implementation of Johnson's Algorithm

__global__ void MSSP( ){
**for** $source\_id$ = $block\_id$ ; $source\_id$ $< bat$; $source\_id$ += $num\_blocks$ **do**
  $Near\_Far\_TB(source\_id)$
**end for**
}
$n_b = n/bat$
**for** $i = 0; i < n_b; i++$ **do**
  MSSP<<<>>>( )
  Transfer results back to CPU.
**end for**

However, with the above method, when the number of edges gets larger, $bat$ would decrease. Now, if $bat$ is smaller than the maximal number of active thread blocks on GPUs, the parallelism provided by GPUs is under-utilized. To utilize the parallelism of GPUs effectively, we proposed to utilize the *Dynamic Parallelism* feature to launch child kernels to process vertices that have a large out-degree. *Dynamic Parallelism* is an extension of the CUDA programming model to enable a CUDA kernel to create and synchronize with new kernels [14]. As launching child kernels causes additional overheads, we only launch kernels for vertices with a large out-degree. When we traverse graphs inside the MSSP kernel, we first launch a child kernel to find all vertices with large out-degrees and push the edge lists of these vertices together to a single queue. Because the sizes of these edge lists are different, we partition edge lists into equal sizes. Then, we launch another child kernel to traverse these edge lists, with each partition assigned to one thread block.

### C. Out-of-Core Boundary Algorithm

Djidjev *et al.* [13] proposed a *boundary algorithm* to solve APSP problems for multi-node clusters. The algorithm is composed of four major steps. During the first step, the graph $G$ is partitioned into $k$ components, i.e, $C_1, C_2 \ldots C_k$. For any edge $(u, v)$, if vertex $u$ and $v$ belong to different components, then both $u$ and $v$ are both *boundary nodes*. The goal of partitioning is to make each component roughly the same size, and the number of *boundary nodes* as small as possible.

Now, in describing the remaining three steps, $dist_i(u, v)$ denotes the current appropriate value of the shortest distance between vertex $u$ and vertex $v$ computed in step i, for i =

2, 3, 4. In the second step, the APSP problem is solved on each component independently to get $dist_2$. In the third step, the *boundary graph* $BG$ is extracted. The nodes of $BG$ are boundary nodes and the edges are edges between each pair of boundary nodes. This includes *virtual edges* computed in the step 2, i.e. if we found that $dist_2(u, v)$ is not infinity after step 2 (implying that there exists a path from $u$ to $v$), then, we add a virtual edge $(u, v)$. In the final step, we compute the shortest distances where at least one node is a non-boundary node using the $dist_2$ and $dist_3$ as follows. We define $B_i = C_i \cap BG$. For non-boundary nodes $v_i$ and $v_j$, which are from components $C_i$ and $C_j$, respectively, we obtain $dist_4$ using the following formula:

$$dist_4(v_i, v_j) = min(dist_2(v_i, b_i) + dist_3(b_i, b_j) + dist_2(b_j, v_j)) \quad (1)$$

where $b_i \in B_i$, $b_j \in B_j$.

**Algorithm 3** Out-Of-Core Implementation of the Boundary Algorithm

**Input**: $A(i, j)$ - distance matrix of block $(i, j)$,
$k$-the number of blocks in each dimension,
*bound*-the distance matrix of the boundary graph
**Output**: $A(i, j)$ - shortest distance matrix of block $(i, j)$

1: **for** $i = 0; i < k; i++$ **do**
2: blocked Floyd-Warshall($A(i, i)$))
3: **end for**
4: Update values in *bound*
5: blocked Floyd-Warshall(*bound*))
6: **for** $i = 0; i < k; i++$ **do**
7: $C2B[i] = ExtractRow(A(i, i)$
8: **end for**
9: **for** $i = 0; i < k; i++$ **do**
10: $B2C[i] = ExtractCol(A(i, i)$
11: **end for**
12: ▷ Compute $dist_4$
13: **for** $i = 0; i < k; i++$ **do**
14: **for** $j = 0; j < k; j++$ **do**
15: Extract $bound(i, j)$
16: $tmp\_1 = C2B[i] \times bound(i, j)$
17: $A(i, j) = tmp\_1 \times B2C[j]$
18: Transfer $A(i, j)$ back to CPU side.
19: **end for**
20: **end for**

Based on the description above, we implemented an out-of-core version of the boundary algorithm as shown in Algorithm 3. In the first step, we adopted a k-way partitioning method from the METIS library [19] to partition the input graphs. We use $A$ to denote the input graph and the corresponding distance matrix and *bound* to denote the matrix for the boundary graph. The structure of matrix $A$ is shown in Figure 1(a). In the following, $A(i, j)$ denotes a sub-matrix of $A$, where values are distances from component $C_i$ to $C_j$. Similarly, $bound(i, j)$ denotes sub-matrix of *bound*, where the values are the distances from boundary nodes of $C_i$ to boundary nodes of $C_j$. For step 2, we perform the blocked version of the Floyd-Warshall algorithm on each diagonal block $A(i, i)$. Then, we update the values in matrix *bound*, i.e. which is to add virtual edges to the boundary graph and apply the blocked version of the Floyd-Warshall algorithm on it to get $dist_3$. Before we compute $dist_4$, we need to

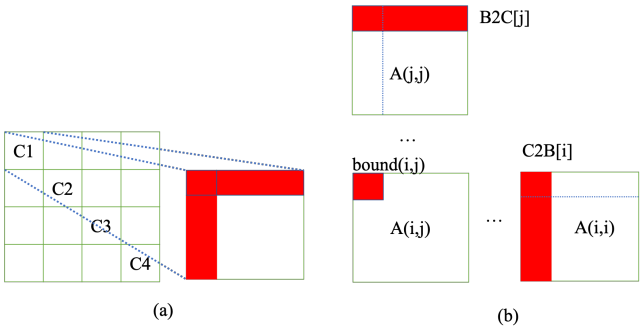


Fig. 1: Out-of-Core Version of the Boundary Algorithm. (a) Shows the Data Structure (the Graphs are Partitioned into Four Components and Each Component Resides in a Diagonal Block). Within Each Block, It Starts with Boundary Nodes, which is Shown in Red Color. (b) Shows the Successive Matrix Multiplications for Computation of Non-Diagonal Block A(i,j)

extract certain matrices as shown in lines 6-11. Here, sub-matrix $C2B[i]$ denotes distances from all nodes of $C_i$ to boundary nodes within the diagonal blocks, which is also the right-bottom red matrix in Figure 1 (b). Similarly, sub-matrix $B2C[i]$ denotes the shortest distances from boundary nodes of $C_i$ to any vertex within the diagonal blocks, which is the left-upper red matrix in Figure 1 (b). According to Equation 1, $A(i, j) = C2B[i] \times bound[i][j] \times B2C[i]$. We perform these two successive multiplications as shown in lines 16-17 and transfer the results back after we process each block.

Here, we use the blocked version of the Floyd-Warshall algorithm described in the previous section to compute $dist_2$ and $dist_3$. This choice is made because first, in our setup, the component sizes are small, specifically, the component size is usually set as around $\sqrt{n}$, which is several hundreds, and second, for graphs with a small separator, the number of boundary nodes is also small, around $\sqrt{kn}$. For these small graphs, the time complexity is less important and the blocked version of the Floyd-Warshall algorithm is more suitable than Johnson's algorithm.

As shown in Algorithm 3, the data transfers from the GPU to the CPU are within a nested loop as shown in line 18. In total, there are $k^2$ data transfers. The value of $k$ is around several hundreds in our experiments. Further, the component sizes are typically small. As a result, it leads to a large number of small data transfers, causing high overheads (which turn out to be between 69.96% to 83.90% of the total execution time in our experiments). To minimize the overhead of data transfers, we first calculate the maximal buffer size allowed and allocate a buffer with that size. Here, we assume that total memory size is $L$, and the memory allocated for diagonal blocks is $S_{dia}$, the memory allocated for the boundary matrix is $S_{bound}$. Then, the remaining data size is

$$S_{rem} = L - S_{dia} - S_{bound}.$$

Suppose the maximal number of vertices in each component is $N_{max}$, the data sizes of each distance value is $W$. Then, we accumulate $N_{row}$ iterations to finish one data transfer, where

$$N_{row} = \frac{S_{rem}}{N_{max} \times n \times W}.$$

Next, we transfer the results of $N_{row}$ iterations back to the CPU until it gets filled. We further improve the performance by overlapping the computations with the data transfers. We create two *streams* and two *buffers* on the device. When we finish the computations for one buffer using the first stream and store the results on GPUs, we can start transferring the data from the first buffer to the CPU's pinned memory and start computations of the next buffer using the second stream. We use the streams and buffers alternatively to complete the overall procedure.

## IV. Performance Characteristics and Algorithm Selection

This section focuses on characterizing different algorithms and developing a methodology to select the best approach for a given matrix.

### A. Summary of Different Implementations

We summarize the key differences between the three algorithms in Table I. The complexity of data movement for both Johnson's algorithm and the boundary algorithm is O($n^2$), while the data movement complexity of the Floyd-Warshall algorithm is O($n_d \times n^2$) since it needs $n_d$ passes. It should be recalled that $n$ is the number of vertices and $n_d$ is the number of blocks a matrix is partitioned into along each dimension.

As for computation complexity, the blocked version of the Floyd-Warshall algorithm incurs a complexity of ($n^3$). Thus, it is only suitable for graphs with a large density. For sparse graphs, the computation complexity of Johnson's algorithm keeps as low as O($mnlog(n)$)[1]. The computation complexity of the boundary algorithm is more complex – it is O($n^2$) if the input graph has a small separator but is close to O($n^3$) otherwise. Thus, for sparse graphs with a small separator, the boundary algorithm delivers a better performance since its data accesses and computation patterns are more regular (dense matrix multiplication), while for other sparse graphs, Johnson's algorithm outperforms the boundary algorithm.

### B. Detailed Cost Models

The execution cost of out-of-core implementations is a combination of data transfer costs and computational costs. In this section, we first model data transfer costs and then introduce the cost models of computations.

*1) Modeling Cost of Data Transfers:* Many previous works [1], [4], [7] proposed cost models of data transfers. Based on these works, we establish data transfers models in our case as follows.

For the Floyd-Warshall algorithm, assuming that the number of vertices in each block is $b$, the number of blocks in each dimension is $n_d$, each distance value occupies $W$ bytes, and the throughput of the data transfer is $TH$ bytes per second, then the cost of data transfer $T_{tr}$ would be:

$$T_{tr} = \frac{n_d \times W \times (b^2 + 2b^2 + n_d^2 b^2)}{TH}.$$

Since $n = n_d \times b$,

$$T_{tr} = \frac{n_d \times W \times (3b^2 + n^2)}{TH}.$$

[1]It assumes that the complexity of our Near-Far implementation is close to the complexity of Dijkstra's algorithm.

For the Johnson's algorithm, assuming that the batch size is $bat$, the number of batches is $n_b$.

$$T_{tr} = \frac{n_b \times W \times bat \times n}{TH}.$$

Since $n = n_b \times bat$,

$$T_{tr} = \frac{W \times n^2}{TH}.$$

For the boundary algorithm, the major overhead of transfers are from the step 4. As shown in the previous section, in each data transfer, we accumulate $N_{row}$ iterations. The size for each data transfer would be $S_{rem}$, and there are $\frac{k}{N_{row}}$ such transfers.

Thus, total transfer cost is:

$$T_{tr} = \frac{k}{N_{row}} \times \frac{S_{rem}}{TH}.$$

*2) Cost Models for Computations:* **The Blocked Version of the Floyd-Warshall Algorithm:** Because the computation and memory access patterns for different graphs are same under the Floyd-Warshall algorithm, the costs per operations are similar. Thus, we assume that the cost of computations is linear to the number of floating-point operations (and memory accesses). According to Algorithm 1, the number of computations and memory transactions are both O($n^3$). For a randomly generated graph with $n_0$ vertices, we can observe the computation time (and denote it by $T_0$). Then, for any given graph with $n$ vertices, we estimate the cost of computation to be:

$$T_{comp} = T_0 \times (\frac{n}{n_0})^3$$

.

**Johnson's Algorithm:** For Johnson's algorithm, we observe that the execution times of different batches are similar. To demonstrate this, we compute the standard deviations of execution times of each batch for several graphs, and found that it ranges between 1.67% and 13.4% of the mean execution time. Thus, to estimate the execution time of a graph, we randomly choose $k$ batches [2] to run and obtain the execution time as $T$. Assuming that the number of batches is $n_b$, the cost of computation would be

$$T_{comp} = \frac{n_b}{k} \times T.$$

**The Boundary Algorithm:** For the boundary algorithm, we establish two expressions depending on whether or note the graph has a small separator size. Intuitively, the separator size denotes the number of boundary nodes after partitioning. As the number of boundary nodes depends on the number of components ($k$), we consider the case when the value of $k$ is $\sqrt{n}$, which is when the costs are minimized. For an "ideal graph", which is also called a *planar graph*, the total number of boundary node will be $\sqrt{kn}$. If the number of boundary nodes for a graph is close to this value, we claim that the graph has a small separator.

Now, if a graph has a small separator, when the number of components, $k$ is $\sqrt{n}$, the number of memory operations and the floating-point operations are minimized, and specifically both are O($n^{\frac{9}{4}}$). Readers can refer to previous work [13] for details of the derivation of this complexity. For our

[2]In our experiments we set $k$ to be 5 as that achieved sufficient accuracy.

TABLE I: Comparisons of Different Implementations. Here, $n$ Denotes the Number of Vertices, $m$ Denotes the Number of Edges, $n_d$ Denotes the Number of Blocks in Each Dimension in the Blocked Version of Floyd-Warshall Algorithm.

| **Algorithm** | **Floyd-Warshall** | **Johnson's** | **Boundary** |
|---|---|---|---|
| **Computation Complexity** | O($n^3$) | O($mnlog(n)$)) | O($n^2$) − O($n^3$) |
| **Data access and control flow patterns** | Regular | Irregular | Regular |
| **Data movement complexity** | O($n_d \times n^2$) | O($n^2$) | O($n^2$) |
| **Target Graphs** | Dense Graphs | Sparse Scale-free Graphs | Graphs with a Small Separator |

work, we have empirically shown that the cost per memory operation and floating-point operation stays similar for these graphs. In this case, the cost of computations would be linear to the number of memory operations and the floating-point operations. Specifically, we first run the boundary algorithm for a graph with a small separator. Suppose that the number of vertices and the computation time are $n_0$ and $T_0$, respectively, then, for a graph with $n$ vertices, the cost of computations is:

$$T_{comp} = T_0(\frac{n}{n_0})^{\frac{9}{4}}.$$

However, if a graph does not have a small separator, the number of boundary nodes cannot be estimated and the number of operations will simply not be O($n^{\frac{9}{4}}$). Suppose that the average number of boundary nodes in each partition is $B$. Then, the number of operations during the second, third, and fourth steps are: $n^3/k^2$, $(kB)^3$, and $nkB^2+n^2B$, respectively. In this way, we get the total number of operations ($N_{op}$) as:

$$N_{op} = \frac{n^3}{k^2} + (kB)^3 + nkB^2 + n^2B.$$

Further, for graphs with a large separator, we observed that the time per operation is no longer similar for different graphs. Instead, this cost ($c_{unit}$) increases when graphs become more irregular. We concluded that $c_{unit}$ depends on the total number of boundary nodes, which is denoted as $NB$ in this work. In the ideal planar graph, $NB = \sqrt{kn} = n^{\frac{3}{4}}$. Thus, we classify $NB$ into different ranges like [$n^{\frac{3}{4}}$, $2 \times n^{\frac{3}{4}}$), [$2 \times n^{\frac{3}{4}}$, $4 \times n^{\frac{3}{4}}$] and so on. We assume that $c_{unit}$ is the same for graphs within a range. For each range, we first calculate $c_{unit}$ of graphs of the range using a set of training graphs. Then, for a given graph, we get the value of $c_{unit}$ based on its range and calculate the computation cost as:

$$T_{comp} = N_{op} \times c_{unit}.$$

### C. Filtering Based on Density

According to our discussions in Section III, a key performance indicator for different implementations is the *density* of the graph. While it may not alone suffice as a method to choose the optimal implementation, it can be used to filter out inefficient implementations in most cases. This approach can at least help us reduce the overheads associated with detailed performance modeling. In this work, we define *density* as $density = \frac{m}{n^2}$, where $m$ is the number of edges and $n$ is the number of vertices. Now, based on our experimental observations, we use the following rules. First, when the density of a graph exceeds 1%, we choose between Johnson's algorithm and the blocked version of the Floyd-Warshall algorithm, as the boundary algorithm is likely to have a large number of boundary nodes and thus will be inefficient. Second, when the density is smaller than 0.01%, we choose between Johnson's algorithm and the boundary algorithm, as the cost of the Floyd-Warshall algorithm is too high in such cases. Third, in other cases, i.e, when the density is between 0.01% and 1%, we always select Johnson's algorithm.

TABLE II: Specifications of Nvidia Tesla V100 and Nvidia Tesla K80

| **GPUs** | **Tesla K80** | **Tesla V100** |
|---|---|---|
| **Architecture** | Kepler | Volta |
| **#SM** | 13 | 80 |
| **maximal launch** | 26 | 160 |
| **FP32 CUDA Cores/GPU** | 2496 | 5120 |
| **Memory Interface** | 384-bit GDDR5 | 4096-bit HBM2 |
| **Max Registers / Thread** | 255 | 255 |
| **Shared Memory Size / SM (KB)** | 112KB | Configurable up to 96 KB |
| **Memory Size** | 12 GB | 16 GB |

## V. Evaluation and Performance Study

We evaluate our implementations with the following goals: 1) to demonstrate that out-of-core GPU implementations are an attractive approach for working with large graphs, by comparing these implementations against a state-of-the-art multi-core CPU implementation, 2) to show that our implementations can continue to scale even when the output does not fit into CPU memory, 3) to understand trade-offs between algorithms and how their relative performance matches what our performance model can predict, and 4) to evaluate benefits from optimizations implemented. We also compare the performance of our implementations with the results reported from several other implementations in a recent publication [31] (where a different but faster multi-core machine was used). It should be noted that there is no other GPU implementation of APSP that can work with graph sizes we are handling.

### A. Experimental Environment

We conducted our experiments on an Nvidia Tesla V100 and an Nvidia Tesla K80. The specifications of these two GPUs are shown in Table II. For most experiments, only the results from Nvidia Tesla V100 are reported due to space limitations. However, Figure 7 and Table V show the generality of the work by using the results of an Nvidia Tesla K80. The GPUs are attached to an Intel(R) Xeon(R) CPU E5-2680 (2013 Ivy Bridge) running at 2.4 GHz. The CPU contains 14 physical cores and provides hyper-threading with 2 threads for each core. The size of the host memory is 128GB in our experiments. The host operating system for our experiments is CentOS Linux release 7.4.1708 (Core). Our GPU implementations are based on CUDA 10.1 toolkit and NVCC V10.1.168 is used to compile our programs.

### B. Input Graphs

We select 29 graphs from the SuiteSparse Matrix Collection [12] for detailed study and analysis. Our experiments

TABLE III: Features of Input Graphs where the Output Fits into CPU Memory

| matrix name | small separator? | n (K) | m (K) | $\sqrt{kn}$ | #boundary nodes | density (%) |
|---|---|---|---|---|---|---|
| pkustk14 | No | 152 | 14,988 | 7,695 | 136,798 | 0.0649 |
| SiO2 | No | 155 | 11,439 | 7,824 | 155,319 | 0.0474 |
| bmwcra_1 | No | 149 | 10,793 | 7,575 | 117,156 | 0.0488 |
| gearbox | No | 154 | 9,234 | 7,764 | 88,741 | 0.0391 |
| oilphan | No | 74 | 3,671 | 4,475 | 42,686 | 0.0675 |
| net4-1 | No | 88 | 2,530 | 5,124 | 57,315 | 0.0324 |
| fe_tooth | No | 78 | 905 | 4,673 | 37,186 | 0.0148 |
| onera_dual | No | 86 | 505 | 5,003 | 31,067 | 0.0069 |
| usroads-48 | Yes | 126 | 324 | 6,694 | 8,790 | 0.0020 |
| usroads | Yes | 129 | 331 | 6,813 | 8,758 | 0.0020 |
| luxembourg_osm | Yes | 115 | 239 | 6,229 | 2,543 | 0.0018 |
| wy2010 | Yes | 86 | 428 | 5,031 | 12,665 | 0.0058 |
| nm2010 | Yes | 169 | 831 | 8,321 | 20,498 | 0.0029 |
| me2010 | Yes | 70 | 335 | 4,281 | 10,668 | 0.0069 |
| md2010 | Yes | 145 | 700 | 7,440 | 17,057 | 0.0033 |
| id2010 | Yes | 150 | 728 | 7,616 | 19,040 | 0.0032 |
| nd2010 | Yes | 134 | 626 | 6,995 | 18,262 | 0.0035 |
| nj2010 | Yes | 170 | 830 | 8,357 | 20,188 | 0.0029 |
| wv2010 | Yes | 135 | 663 | 7,051 | 17,734 | 0.0036 |

TABLE IV: Features of Input Graphs where Output does not Fit into the CPU's Memory

| matrix | n (K) | m (K) | density (%) |
|---|---|---|---|
| af_shell1 | 505 | 18,094 | 0.0071 |
| cage13 | 445 | 7,479 | 0.0038 |
| kim2 | 457 | 11,330 | 0.0054 |
| Lin | 256 | 2,022 | 0.0031 |
| pwtk | 218 | 11,852 | 0.0250 |
| Stanford | 282 | 2,312 | 0.0029 |
| stomach | 213 | 3,022 | 0.0066 |
| troll | 213 | 12,199 | 0.0268 |
| boyd2 | 466 | 1,780 | 0.0008 |
| CO | 221 | 7,887 | 0.0161 |

use *int* as the data type for distance value, which allowed the use of *atomicMin* operations for the out-of-core version of Johnson's algorithm. We verified that the output distance matrices of any of these graphs cannot fit into NVIDIA Tesla V100 GPU's device memory. The first 19 graphs' output fits in the CPU main memory, while the last 10 graphs' output does not. Table III shows the main features of the graphs where the output can fit into CPU memory. The number of vertices of these graphs is in the range 69,518 - 1695,88. Here, we classify graphs into two kinds based on the number of boundary nodes after partitioning graphs using a k-way partitioning method from the METIS library [19]. We show the number of the boundary nodes in the table – as this value depends on the number of components ($k$), we obtain this value with $k$ to be $\sqrt{n}$ , where $n$ is the number of vertices. If the actual number of boundary nodes is close to the value of $\sqrt{kn}$, it is a graph with a *small separator*. In this work, we classify 11 graphs as graphs with a small separator, as shown in the table. We also show the densities of these graphs in the last column. As expected, graphs with a small separator are much sparser than other sparse graphs. TableIV shows the features of 10 large graphs where the output can not fit into CPU memory. Finally, some of our experiments use synthetic matrices generated by the R-MAT generator [8]

### C. Comparison with Multi-Core CPU Implementations

We first compare the performance of our out-of-core GPU implementations with multi-core implementations. In our experiments, the data transfer times between GPU and CPU are included. The baseline we execute and perform most comparisons with is what we refer to as the `BGL-Plus` implementation. This implementation uses OpenMP to parallelize among different SSSP instances, which are themselves using Dijkstra's algorithm implementation from the popular Boost Graph Library (BGL) [33]. Figure 2 compares the execution times of BGL-plus with our out-of-core GPU implementation for graphs with a small separator. In this case, our out-of-core implementation chooses the boundary algorithm. As shown in the figure, our out-of-core implementation achieves speedups in the range 8.22-12.40x. Figure 3 shows the comparison results for other sparse graphs we had listed earlier in Table III. In this experiment, our out-of-core implementation is the one based on Johnson's algorithm. We note that the speedups are in the range of 2.23-2.79x, which again shows that out-of-core GPU implementations are effective. These speedups are relatively lower because with the Johnson's method, as the number of edges increases, the batch size gets smaller (with output limitations). This, in turn, implies that there is less work to be parallelized within the GPU.

Two other candidates for comparison include a recent implementation of the Floyd-Warshall algorithm [31] and the implementation from the Galois Graph library [28] (which uses the parallel delta-stepping variant of Dijkstra's algorithm). We denote them as `SuperFW` and `Galois` implementations, respectively. Since we either did not have the codes or could not execute these, we simply examine the reported execution times for the same graphs in the 2020 publication [31]. Their experiments are performed in a shared memory system that contains 32 cores as a dual-socket 16-core Intel E5-2698 v3 "Haswell" processors. Since each core can support two hyperthreads, they use 64 threads in total. This is overall a more powerful system than the one where our multi-core executions took place. The comparison results are shown in Figure 4. As can be seen from the table, our out-of-core implementation is more efficient with speedups are in the

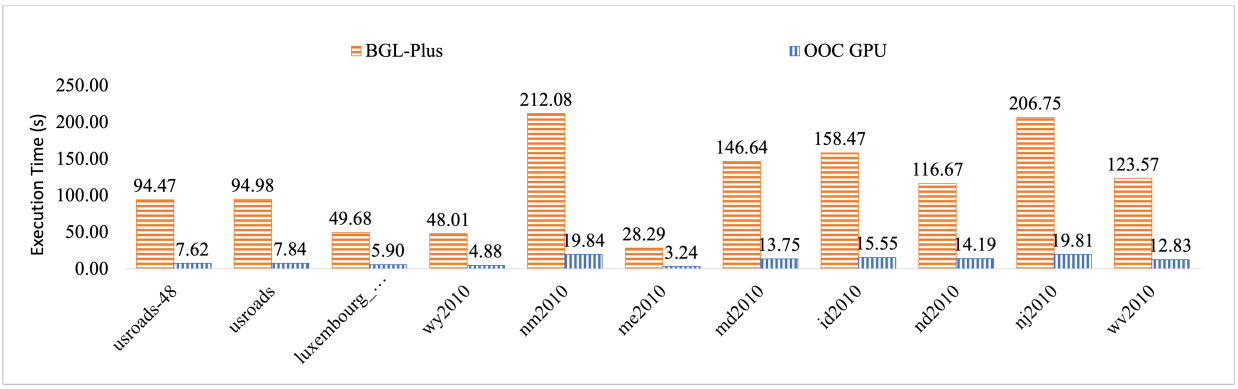


Fig. 2: Comparing the Performance of our Out-of-Core Implementation of the Boundary Algorithm with `BGL-Plus` Multi-Core Implementation for Graphs with a Small Separator

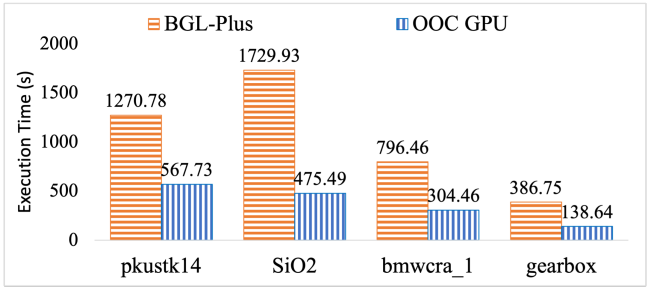


Fig. 3: Comparing Performance of our Out-of-Core Johnson's GPU Implementation with `BGL-Plus` Multi-Core Implementation for Other Sparse Graphs

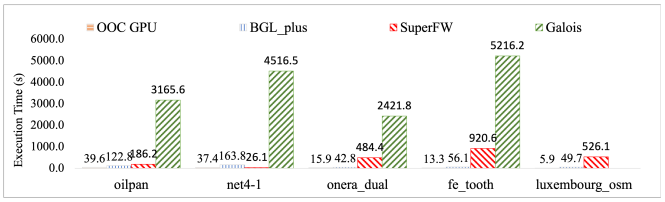


Fig. 4: Execution Times of Our Out-of-Core Implementation, BGL-Plus, SuperFW Implementation, and Galois Implementation (some of the results as reported by Sao *et al.*[31])

range 4.70-69.21 over SuperFW and 79.93-152.62 over Galois for all graphs except *net4-1*.

### *D. Performance on Large Matrices*

In this subsection, we demonstrate that our implementation is still efficient when the graphs are large and their output cannot fit into the CPU's main memory. To achieve this, we first showed the execution times of the larger graphs previously summarized in Table IV. Figure 5 summarizes these results. While none of the other implementations previously considered could process these graphs, our implementation could.

The performance of our implementations is impacted by a large number of factors, including the distribution of in-degree of vertices. Thus, to better understand the performance implications of working with very large graphs, we performed experiments with graphs generated using the R-MAT generator [8]. The distribution of in-degrees does not change as the

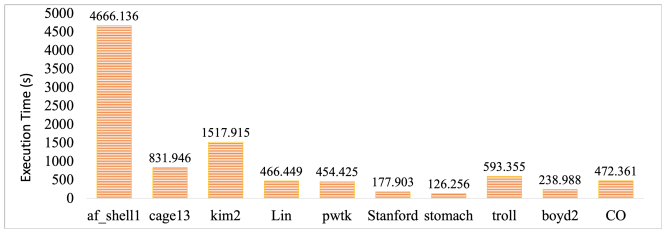


Fig. 5: Execution Times of Out-of-Core Implementation on Large Matrices (from Table IV)

TABLE V: Computational Efficiency as Graph Sizes are Scaled (Synthetic Graphs from the R-MAT)

| n | m | (n*m)/s | execution time on V100 (s) | execution time on K80 (s) |
|---|---|---|---|---|
| 20000 | 128000 | 1.03E+10 | 0.249444 | 2.13395 |
| 40000 | 256000 | 8.67E+09 | 1.18131 | 8.15059 |
| 80000 | 512000 | 7.33E+09 | 5.58834 | 34.3958 |
| 160000 | 1024000 | 6.40E+09 | 25.6043 | 146.029 |
| 320000 | 2048000 | 6.14E+09 | 106.821 | 605.126 |
| 20000 | 256000 | 1.04E+10 | 0.493702 | 3.86824 |
| 40000 | 512000 | 7.77E+09 | 2.63682 | 17.6307 |
| 80000 | 1024000 | 6.84E+09 | 11.9776 | 69.4629 |
| 160000 | 2048000 | 5.05E+09 | 64.9454 | 301.055 |
| 320000 | 4096000 | 3.10E+09 | 423.322 | 1506.42 |
| 20000 | 512000 | 1.08E+10 | 0.944806 | 6.50237 |
| 40000 | 1024000 | 5.37E+09 | 7.62732 | 29.4108 |
| 80000 | 2048000 | 3.85E+09 | 42.5313 | 132.674 |
| 160000 | 4096000 | 3.26E+09 | 201.193 | 633.463 |
| 320000 | 8192000 | 2.09E+09 | 1255.77 | 3459.07 |

graph size is varied. The graphs, as summarized in Table V, range from those where the output fits into GPU memory (up to 40,000 nodes) and where the output does not even fit into CPU memory (320,000 nodes). The execution times of these matrices on both Nvidia V100 and Nvidia K80 are also shown in the table. For these graphs, the optimal implementation is always Johnson's algorithm. As shown in section III, the computational workload of Johnson's algorithm is $O(n \times m)$. Thus, we use $n \times m/s$ ($s$ is the time in seconds) to denote the computational efficiency of our implementation. As can be seen from the table, the values of $n \times m/s$ are relatively stable when we increase the number of edges for these graphs. This shows that data movement costs do not dominate the overall performance as sizes are increased.

### *E. Comparison between Different Implementations and the Effectiveness of Our Selector*

To examine the relative performance of different implementations and the effectiveness of our performance prediction methodology, we experimented with all graphs from SuiteS-

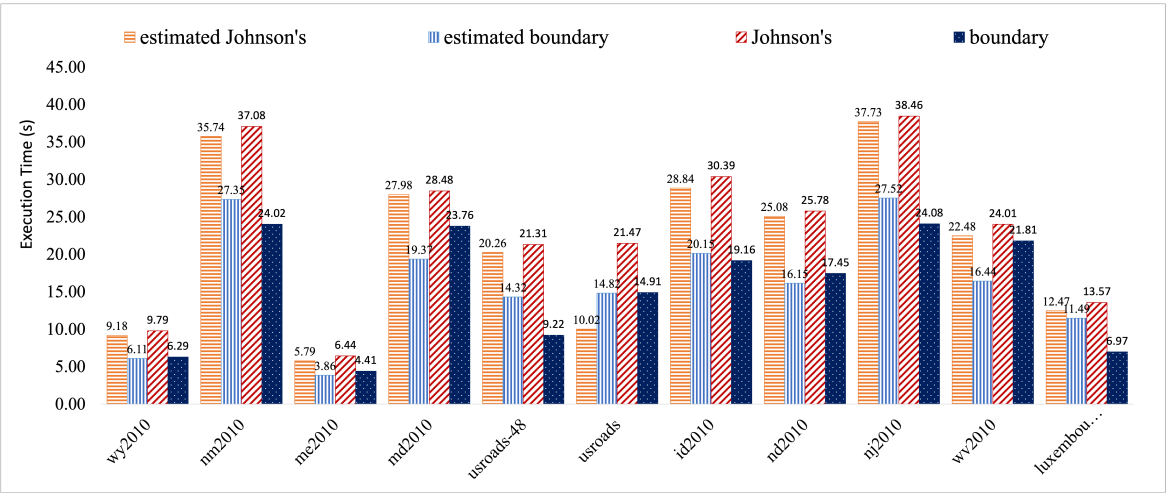


Fig. 6: Estimated and Actual Execution Times of the Boundary Algorithm and Johnson's Algorithm for Graphs with a Small Separator on Nvidia V100

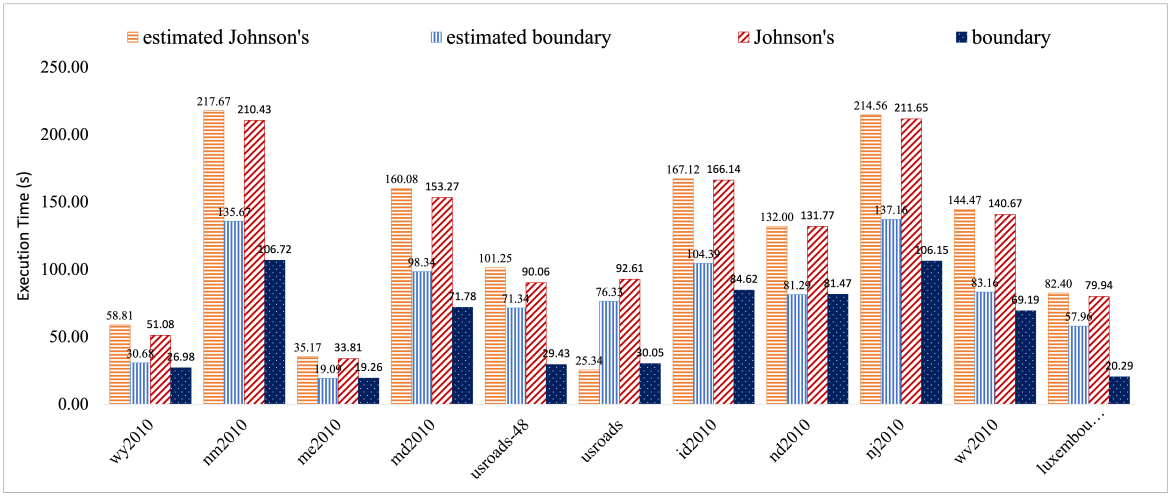


Fig. 7: Estimated and Actual Execution Times of the Boundary Algorithm and Johnson's Algorithm for Graphs with a Small Separator on Nvidia K80

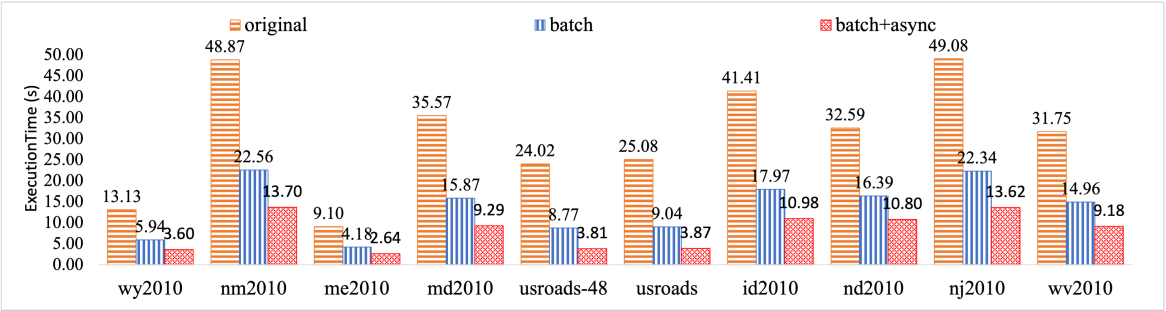


Fig. 8: Benefits from Optimizations in Out-of-Core Boundary Algorithm

TABLE VI: Selection between Johnson's Algorithm and the Blocked Version of the Floyd-Warshall (FW) Algorithm (all times in seconds)

| setups | n | m | FW | estimated FW | Johnson's | estimated Johnson's |
|---|---|---|---|---|---|---|
| setup1 | 80,000 | 51,200,000 | 7,245.81 | 9,521.67 | 4,224.15 | 4,889.07 |
| setup1 | 80,000 | 102,400,000 | 7,280.00 | 9,521.67 | 21,646.46 | 24,186 |
| setup1 | 80,000 | 204,800,000 | 7,510.16 | 9,521.67 | 46,318.53 | 57,112.9 |
| setup2 | 80,000 | 51,200,000 | 7,301.32 | 9,521.67 | 2,624.38 | 3,480.62 |
| setup2 | 80,000 | 102,400,000 | 7,842.77 | 9,521.67 | 16,702.08 | 19,930.9 |
| setup2 | 80,000 | 204,800,000 | 8,104.72 | 9,521.67 | 64,683.64 | 72,239.9 |
| setup3 | 80,000 | 51,200,000 | 7,364.78 | 9,521.67 | 4,836.88 | 6,677.07 |
| setup3 | 80,000 | 102,400,000 | 7,519.44 | 9,521.67 | 24,363.89 | 29,567.8 |
| setup3 | 80,000 | 204,800,000 | 7,316.42 | 9,521.67 | 64,624.34 | 84,808.8 |

parse matrix collection, in which the number of vertices is in the range $[80,000, 100,000]$. We first verified the effectiveness of the filtering method for graphs whose densities are between between 0.01% and 1%. For other graphs, we select the optimal implementation based on the cost models.

In our experiments, the estimated overheads of data transfers are also included. To get the throughput of data transfers on GPUs, we use *nvprof* to measure the execution time of a test program, which transfers 1M integers from the device to the host. Based on our experiments, the data transfer throughput on Nvidia K80 and Nvidia V100 are 7.23GB/s and 11.75 GB/s, respectively.

*Selection between Johnson's Algorithm and the Boundary Algorithm:* For the graphs where the densities are smaller than 0.01%, we consider Johnson's algorithm and the boundary algorithm as the Floyd-Warshall algorithm is not competitive. For this experiment, we randomly select 5 batches to run and estimate the overall computation time of Johnson's algorithm. For the boundary algorithm, we use different cost models for different kinds of graphs as mentioned in Section IV.

For graphs with a small separator, we show results on Nvidia V100 and Nvidia K80 in Figure 6 and Figure 7, respectively. As can be seen from the figures, our cost model can quite accurately predict the real execution times and is always able to choose the correct implementation different graphs.

*Selection between Johnson's Algorithm and the Blocked Version of the Floyd-Warshall Algorithm:* As also stated in Section IV-C, when graph density is greater than 1%, the two competitive algorithms are Johnson's algorithm and the blocked version of the Floyd-Warshall (FW) algorithm. Unfortunately, these density levels are uncommon in real graphs. Thus, we carried out an evaluation using synthetic scale-free graphs, which are generated using the R-MAT generator. To estimate the execution times of the blocked version of the Floyd-Warshall algorithm, we first get the execution time of a graph with 70000 vertices. In this experiment, we fix the number of vertices to be 80000 and double the number of edges each time. As indicated in Table VI, the execution times of the blocked version of the Floyd-Warshall algorithm depend on the number of vertices, while the execution times of Johnson's algorithm increase with the number of edges when the number of vertices is fixed. More importantly, our selector can always select the optimal implementation.

### F. Effectiveness of Optimizations

For the boundary algorithm implementation, we performed experiments for graphs with a small separator because the boundary algorithm is more efficient for this kind of graphs. We set the number of components to be $\sqrt{n}/4$ since we found it achieves the best performance in most cases. The comparison results are shown in Figure 8. We observe that it achieves speedups of 1.988-5.706 with batching because it improves the data transfer efficiency. At the same time, overlapping data transfers with computations achieves performance improvement in the range of 12.7%-29.1%.

## VI. Related Work

In this section, we first briefly introduce research efforts on out-of-core implementation for graph algorithms. Then, we introduce existing work on GPU acceleration of APSP. Finally, we discuss major research works on SSSP optimizations on GPU.

Most research efforts on out-of-core implementation are based on the assumption that the input graph is too large to fit in GPU device memory. Typically, the input graph is first partitioned into smaller chunks, then explicitly loaded to the GPU memory in each iteration of the graph processing. The major challenge of this approach is that the overhead of data transfer is typically much larger than the computation costs. To reduce the data transfer overhead, Sengupta *et al.* [32] proposed to detect and skip inactive partitions. Han *et al.* [15] further improve the approach, with an adoption of X-Stream style graph processing and a couple of renaming techniques to reduce the cost of explicit GPU memory management. Recently, Sabet *et al.* [30] proposed efficient GPU-accelerated subgraph generation techniques to further reduce the data transfer overhead. Besides, they adopt asynchronous execution to reduce the needs for subgraph generations and reloading.

In the area of GPU acceleration of APSP, the initial work by Harish and Narayanan [16] proposed implementations of both the Dijkstra and Floyd-Warshall algorithms and compared them to parallel CPU implementations. They reported that execution times are long for both approaches. Based on earlier work by Gayathri *et al.* [35] on the blocked Floyd-Warshall algorithm, a GPU implementation of blocked Floyd-Warshall algorithm was presented [20] that exploited shared memory. All of this work only considered small graphs and cannot handle graphs of the sizes we have considered.

Many GPU implementations of APSP are based on the Bellman-Ford algorithm [5], [6], [16], [34]. An important issue with these efforts is that processing of vertices in arbitrary order leads to redundant work. An ideal implementation should provide high degree of parallelism like the Bellman-Ford's and the work efficiently like Dijkstra's algorithms. Meyer *et al.* [24] proposed delta-stepping algorithm to explore the space between these two endpoints. However, since the overhead of organization in the delta-stepping algorithm is expensive, many efforts [36], [11], [37], [3] utilized the *Near-Far* algorithm, which simplifies the delta-stepping algorithm.

## VII. Conclusions

All Pair Shortest Path (APSP) is a classical graph problem that offers many challenges in developing an out-of-core version. On one hand, the size of the output can be orders of magnitude larger than the size of the input. At the same time, existing algorithms involve complex trade-offs and the execution can be compute-bound or I/O-bound depending upon the graph involved. This paper has thoroughly explored the design space for such out-of-core versions. We start with three algorithms developed in the past for other contexts,

and created novel out-of-core implementations, carrying out several optimizations in the process. Next, we focused on the problem of choosing the best approach for a given graph, and developed a combination of simple heuristics (based on density) and detailed cost models. Our evaluation shows that for graphs with a small separator, the boundary algorithm usually achieves the best performance because of regular memory accesses and computation pattern. For other sparse graphs, the maximal number of components allowed in the boundary algorithm is small and thus Johnson's algorithm achieves better performance. However, as density of the graphs increase, the workload of Johnson's algorithm increases, and when graphs are sufficiently dense, the blocked version of the Floyd-Warshall algorithm delivers the best performance. Our evaluation also shows that our out-of-core implementations outperform an efficient implementation of multi-core implementation, i.e, it achieves speedups ranging from 8.22 to 12.40 for graphs with a small separator, and speedups ranging from 2.23 to 2.79 for other sparse graphs. We also show significant performance improvements from our optimizations.

*Acknowledgements:* This work was partially supported by NSF awards 2131509, 2034850, and 2007793.

## REFERENCES

[1] Alok Aggarwal and S Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] Jaume Barceló, Esteve Codina, Jordi Casas, Jaume L Ferrer, and David García. Microscopic traffic simulation: A tool for the design, analysis and evaluation of intelligent transport systems. *Journal of intelligent and robotic systems*, 41(2):173–203, 2005.

[3] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. *ACM SIGPLAN Notices*, 52(8):235–248, 2017.

[4] Michael Boyer, Jiayuan Meng, and Kalyan Kumaran. Improving gpu performance prediction with data transfer modeling. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1097–1106. IEEE, 2013.

[5] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE IISWC*, pages 141–151. IEEE, 2012.

[6] Federico Busato and Nicola Bombieri. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2222–2233, 2015.

[7] Thomas C Carroll and Prudence WH Wong. An improved abstract gpu model with data transfer. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 113–120. IEEE, 2017.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SDM*, pages 442–446. SIAM, 2004.

[9] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. The exact distance to destination in undirected world. *The VLDB Journal*, 21(6):869–888, 2012.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[11] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Proceedings of IPDPS*, pages 349–359. IEEE, 2014.

[12] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[13] Hristo Djidjev, Sunil Thulasidasan, Guillaume Chapuis, Rumen Andonov, and Dominique Lavenier. Efficient multi-gpu computation of all-pairs shortest paths. In *Proceedings of IPDPS*, pages 360–369. IEEE, 2014.

[14] Design Guide. Cuda c programming guide. *NVIDIA, July*, 2013.

[15] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th PACT Conference*, pages 233–245. IEEE, 2017.

[16] Pawan Harish and Petter J Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

[17] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. *Acm Sigplan Notices*, 46(8):267–276, 2011.

[18] Florian Huc, Aubin Jarry, Pierre Leone, and Jose Rolim. Brief announcement: routing with obstacle avoidance mechanism with constant approximation ratio. In *Proceedings of PODC*, pages 116–117, 2010.

[19] George Karypis and Vipin Kumar. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering*, 38, 1998.

[20] Gary J Katz and Joseph T Kider. All-pairs shortest-paths for large graphs on the gpu. 2008.

[21] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[22] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd HPDC Conference*, pages 239–252, 2014.

[23] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 SIGMOD*, pages 447–461, 2016.

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

[25] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of SOSP*, pages 456–471, 2013.

[26] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices*, 53(2):622–636, 2018.

[27] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN OOPSLA*, pages 1–19, 2016.

[28] Keshav Pingali. High-speed graph analytics with the galois system. In *Workshop on Parallel Programming for Analytics Applications*, pages 41–42, 2014.

[29] Gábor Rétvári, József J Bíró, and Tibor Cinkler. On shortest path representation. *IEEE/ACM Transactions on Networking*, 15(6):1293–1306, 2007.

[30] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: minimizing data transfer during out-of-gpu-memory graph processing. In *European Conference on Computer Systems*, pages 1–16, 2020.

[31] Piyush Sao, Ramakrishnan Kannan, Prasun Gera, and Richard Vuduc. A supernodal all-pairs shortest path algorithm. In *Proceedings of the 25th ACM PPoPP*, pages 250–261, 2020.

[32] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *SC'15 Proceedings*, pages 1–12. IEEE, 2015.

[33] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

[34] Ganesh G Surve and Medha A Shah. Parallel implementation of bellman-ford algorithm using cuda architecture. In *2017 ICECA*, volume 2, pages 16–22. IEEE, 2017.

[35] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics (JEA)*, 8:2–2, 2003.

[36] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu. In *Proceedings of PPoPP*, pages 38–52, 2019.

[37] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of PPoPP*, pages 1–12, 2016.