# Scalable All-pairs Shortest Paths for Huge Graphs on Multi-GPU Clusters

Piyush Sao[†], Hao Lu[†], Ramakrishnan Kannan[†], Vijay Thakkar[‡], Richard Vuduc[‡], Thomas Potok[†*]
[†]Oak Ridge National Laboratory, Oak Ridge, TN
[‡] Georgia Institute of Technology, Atlanta, GA
USA

## ABSTRACT

We present an optimized Floyd-Warshall (Floyd-Warshall) algorithm that computes the All-pairs shortest path (Apsp) for GPU accelerated clusters. The Floyd-Warshall algorithm due to its structural similarities to matrix-multiplication is well suited for highly parallel GPU architectures. To achieve high parallel efficiency, we address two key algorithmic challenges: reducing high communication overhead and addressing limited GPU memory. To reduce high communication costs, we redesign the parallel Floyd-Warshall (a) to expose more parallelism, (b) aggressively overlap communication and computation with pipelined and asynchronous scheduling of operations, and (c) tailored MPI-collective. To cope with limited GPU memory, we employ an offload model, where the data resides on the host and is transferred to GPU on-demand. The proposed optimizations are supported with detailed performance models for tuning. Our optimized parallel Floyd-Warshall implementation is up to 5× faster than a strong baseline and achieves 8.1 PetaFLOPS/sec on 256 nodes of the Summit supercomputer at Oak Ridge National Laboratory. This performance represents 70% of the theoretical peak and 80% parallel efficiency. The offload algorithm can handle 2.5× larger graphs with a 20% increase in overall running time.

## CCS CONCEPTS

• **Software and its engineering** → *Massively parallel systems*; • **Theory of computation** → **Shortest paths**; • **Computing methodologies** → *Graphics processors*; Massively parallel and high-performance simulations.

## KEYWORDS

Floyd-Warshall algorithm, all-pair shortest path, semi-ring algebra, GPU computing, distributed algorithm


*Corresponding author*: Piyush Sao, email: saopk@ornl.gov


*HPDC '21, June 21–25, 2021, Virtual Event, Sweden*

ACM ISBN 978-1-4503-8217-5/21/06…$15.00
https://doi.org/10.1145/3431379.3460651

**ACM Reference Format:**
Piyush Sao[†], Hao Lu[†], Ramakrishnan Kannan[†], Vijay Thakkar[‡], Richard Vuduc[‡], Thomas Potok[†]. 2021. Scalable All-pairs Shortest Paths for Huge Graphs on Multi-GPU Clusters. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3431379.3460651

## 1 INTRODUCTION

We consider the problem of computing all-pairs shortest paths (Apsp) for a dense graph in a distributed memory multi-GPU cluster. The Apsp is a fundamental graph problem with many applications in data analytics on knowledge graphs, traffic routing and simulation, and network design, to name a few. For instance, in knowledge graph analytics, the relationship mining problems become computing Apsp in a large and dense graph [22]. This calculation requires massive computing power and aggregate memory capacity only available in a multi-GPU cluster like the Summit supercomputer at Oak Ridge National Laboratory. Floyd and Warshall (Floyd-Warshall) is the algorithm of choice for the Apsp computation on dense graphs. The Floyd-Warshall algorithm, due to its structural similarities with matrix multiplication, is also well suited for a massively parallel GPU architecture. In this paper, we present *a highly scalable distributed memory Apsp for multi-GPU clusters at larger problem sizes.*

Our baseline is a strong one, being the parallel and blocked variant of Floyd-Warshall listed in Algorithm 3 with GPU acceleration using the semi-ring matrix multiplication kernels (SRGEMM) available in cuASR [40]. We identify two key algorithmic challenges in achieving high parallel efficiency in the *baseline*. First, we need to reduce or hide the communication cost to scale to a large number of processors. However, the baseline algorithm has an implied bulk-sequential dependence between the major steps of each iteration, which makes it difficult to overlap communication with computation. Second, the baseline algorithm performs Apsp only using GPU memory. Hence, the largest feasible problem sizes for running Apsp is limited by the available aggregate GPU memory. Beyond these algorithmic challenges, extracting high performance requires a careful mapping of algorithms to hardware and choosing the appropriate algorithmic parameters.

We relax the bulk-sequential dependence by exploiting a finer-grained dependence structure, which in turn reduces communication costs. This tactic allows us to expose more parallelism and effectively overlap communication and computation via pipelined and asynchronous scheduling. Additionally, we further reduce the communication costs using a customized MPI-collective tailored

to Floyd-Warshall-Apsp. Finally, we overcome the memory limitation by using an offload model. In this technique, all the data resides in the host, and the data is transferred for diagonal-block updates and min-outer products to the GPU on an as-needed basis. We carefully design the pipeline scheme to mask host-device data transfers.

We enhance the baseline algorithm with these optimizations to arrive at the following two flavors of ParallelFw , targeting both parallel and problem scalability:

- Communication Optimized ParallelFw (Co-ParallelFw): This flavor uses several algorithmic tools to hide the cost of communication and latency bound operations.
- Memory-Efficient ParallelFw (Me-ParallelFw): This strategy allows us to solve larger problems that do not fit in aggregate GPU memory with negligible performance overhead.

We derive detailed performance models for both variants, which guides our performance tuning for the Summit system. Our performance models are general and apply to other accelerated supercomputer systems as well.

We evaluate the performance of individual optimizations and the proposed algorithmic variants on the Summit supercomputer. The optimized Co-ParallelFw is up to 5× faster than the baseline on 256 nodes (Figure 8). The Co-ParallelFw variant achieves 8.1 PetaFLOPS/sec on 256 nodes, which is close to 70% of the theoretical peak and equates to 80% parallel efficiency. This theoretical efficiency is similar to the HPL benchmark on Summit. With proper parameter tuning, Me-ParallelFw achieves 80% of the Co-ParallelFw. That permits handling of graphs that are 2.5-times larger than the baseline's capability with only a 20% increase in overall running time. In particular, we are able to solve an Apsp on a graph with 1.66 million vertices, which has a roughly 10 TB memory footprint for the output, on 64 GPU-enabled nodes.

## 2 BACKGROUND

**Table 1: Notation**

| Category | Symbol | Description |
|---|---|---|
| Processes | $P$ | Number of MPI Processes |
| | $P_r, P_c$ | Row and Column Processes |
| | $P_r(k)$ | $(k \bmod P_r)$th Process Row |
| | $P_c(k)$ | $(k \bmod P_r)$th Process Column |
| | $K_r, K_c$ | Node grid dimensions |
| | $Q_r, Q_c$ | Intranode process grid |
| Matrices | $A$ | Distance matrix of the Graph |
| | $A(:,k)$ | $A(k:n,k)$ |
| | $A(k,:)$ | $A(k,k+1:n)$ : kth $A$ panels |
| Graph | $G=(V,E)$ | Input Graph |
| | $n, m$ | Number of vertices and edges |

In this section, we describe the baseline sequential and parallel Floyd-Warshall algorithm. The key concepts include the min-plus semi-ring formulation of Apsp and baseline architecture-independent performance models. In Table 1 we summarize the notations.

**Algorithm 1** Floyd-Warshall algorithm for Apsp

```
def FloydWarshall(G = (V,E)):
  n ← dim(V)
  Dist[i,j] = { w_{i,j}   if (i,j) ∈ E
              { ∞         otherwise
  for k in {1, 2..., n}:
    for i in {1, 2..., n}:
      for j in {1, 2..., n}:
        Dist[i,j] = min{Dist[i,j], Dist[i,k]+Dist[k,j]}
```

### 2.1 Graph All Pair Shortest Path (Apsp) Problem

*Notation and terminology.* Let $G = \{V, E\}$ be an directed weighted graph with a vertex set $V$ containing $n = |V|$ vertices or nodes, edge set $E$ with $m = |E|$ edges. The weight $W : E \mapsto \mathbb{R} \cup \infty$ denotes the edge weights. The $W$ is stored as a matrix so that $w_{i,j}$ denotes the the distance between the $i$-th vertex $v_i \in V$ and $j$-th vertex $v_j$ if if $\{i, j\} \in E$; otherwise, $w_{i,j} = \infty$. A *path* between two vertex $v_s$ and $v_d$ is a sequence of edges $e \in E$ starting that starts at vertex $v_s$ and ends at vertex $v_d$. The *length* of path is the sum of edge weights $w_{i,j}$ in a path. A *shortest path* between two vertex $v_s$ and $v_d$ is the path with the minimal length. The length of the shortest path between two vertex $v_s$ and $v_d$ is called *distance* denoted as $\text{Dist}[s, d]$.

*Apsp Computation.* Apsp is the simultaneous computation of the shortest paths between all pairs of vertices in a graph. During the computation of Apsp, Floyd-Warshall maintains and updates a 2-D array of distances, Dist. Each entry $\text{Dist}[i, j]$ holds the current shortest distance between $v_i$ and $v_j$ discovered so far, with its value at the termination of the algorithm being the shortest such distance. We will assume for simplicity that the graph $G$ consists of a single connected component, in which case the Dist eventually becomes fully dense; however, our implementation will work when there are multiple connected components.

### 2.2 Sequential Floyd-Warshall algorithm

Floyd-Warshall uses a dynamic programming approach to computing Apsp, as shown in algorithm 1. It initializes Dist with the input weights $W$. Then, in the $k$-th iteration, it checks for all pairs of vertices $v_i$ and $v_j$ if there is a shorter path between them via the intermediate vertex $v_k$. If so, Floyd-Warshall updates $\text{Dist}[i, j]$. Therefore, $\text{Dist}[i, j]$ after $k$ steps, which we denote by $\text{Dist}^k(i, j)$, may be defined recursively as

$$\text{Dist}^k[i,j] \leftarrow \min\left\{\text{Dist}^{k-1}[i,j], \text{Dist}^{k-1}[i,k] + \text{Dist}^{k-1}[k,j]\right\}.$$

In Algorithm 1, this computation is done in place by overwriting $\text{Dist}^k[i, j]$ in place of $\text{Dist}^{k-1}[i, j]$.

At any $k^{th}$ iteration, Floyd-Warshall maintain the invariance that $\text{Dist}[i, j]$ holds the current shortest distance between $v_i$ and $v_j$ with all intermediate vertices $k \in (v_1, v_2, \cdots, v_k)$ so far. This invariance is always satisfied when there are no cycles of negative weight sum.

**Algorithm 2** A *blocked* version of Floyd-Warshall algorithm

```
def BlockedFW (A, b):
# Here A is input distance matrix, n = dim(A) and
# b is the block size and n_b ← n/b
for k in {1, 2, ..., n_b}:
  # Diagonal Update
  A(k,k) ← Floyd-Warshall (A(k,k))
  # Panel Update
  A(k,:) ← A(k,:) ⊕ A(k,k)⊗A(k,:)
  A(:,k) ← A(:,k) ⊕ A(:,k)⊗A(k,k)
  # Min−plus Product
  for i in {1, 2..., n_b}:
    for j in {1, 2..., n_b}:
      A(i,j) ← A(i,j) ⊕ A(i,k)⊗A(k,j)
```

## 2.3 Min-Plus Matrix Multiplication

Apsp may be understood algebraically as computing the matrix closure of the weight matrix, $W$, defined over the tropical semiring [14]. In more basic terms, let $\oplus$ and $\otimes$ denote the two binary scalar operators

$$\begin{aligned} x \oplus y &:= \min(x, y) \\ x \otimes y &:= x + y, \end{aligned}$$

where $x$ and $y$ are real values or $\infty$. Next, consider two matrices $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$. The Min-Plus product $C$ of $A$ and $B$ is

$$C_{ij} \leftarrow \sum_{k}^{\oplus} A_{ik} \otimes B_{kj} = \min_{k} \left(A_{ik} + B_{kj}\right).$$

This product is the analogue of matrix-matrix multiplication over the reals.

## 2.4 Blocked Floyd-Warshall algorithm

Let us divide Dist into $n_b \times n_b$ blocks, each of size $b \times b$ (i.e., $n_b = \frac{n}{b}$). If $A_{ij}$ denote the $(i, j)$ block of $A$, where $1 \le i, j \le n_b$, a blocked version of Floyd-Warshall, called BlockedFw, can carry out the same Apsp computation as Floyd-Warshall in the following three steps as illustrated in Algorithm 2

- **Diagonal Update (DiagUpdate):** Perform the classic Floyd-Warshall algorithm on a diagonal block, $A_{kk}$.
- **Panel Update (PanelUpdate):** Update the $k$-th block row and column. For any block $A(k, j)$, $j{\neq}k$ in the block row, the update is a Min-Plus multiply with $A_{kk}$ from the left, and for block $A(i, k)$ on the $k$-th block column is Min-Plus multiply with $A_{kk}$ from right, i.e.,

$$\begin{aligned} A(k,j) &\leftarrow A(k,j) \oplus A(k,k)\otimes A(k,j) && j \neq k \\ A(i,k) &\leftarrow A(i,k) \oplus A(i,k)\otimes A(k,k) && i \neq k \end{aligned}$$

  Here, $\oplus$ denotes element-wise application of the corresponding scalar operator, and $\otimes$ denotes Min-Plus product.
- **MinPlus Outer Product**: Perform the outer product of $k$-th block row and block column, and update all the remaining blocks of matrix $A$

$$A(i,j) \leftarrow A(i,j) \oplus A(i,k)\otimes A(k,j) \quad i, j \neq k.$$

  This step is analogous to a Schur-complement update in LU or Cholesky factorization.

**Algorithm 3** Parallel Floyd-Warshall algorithm on 2D process grid

```
def ParallelFW(A, P = P_r × P_c):
# A is distributed in block cyclic fashion
# my process Id is p_id
# On each MPI process p_id do in parallel:
for k in {1, 2..., n_b}:
  #Diagonal Update and Broadcast
  if p_id= p_{k,k}:
    A(k,k) ← Floyd-Warshall (A(k,k))
    Broadcast(A(k,k), P_r(k))
    Broadcast(A(k,k), P_c(k))
  #Panel Update and Broadcast
  if p_id ∈ P_r(k):
    Receive(A(k,k), p_{k,k})
    A(k,:) ← A(k,:) ⊕ A(k,k)⊗A(k,:)
    Broadcast(A(k,:), P_c(p_id))
  else :
    Receive(A(k,:))
  if p_id ∈ P_r(c):
    Receive(A(k,k), p_{k,k})
    A(:,k) ← A(:,k) ⊕ A(:,k)⊗A(k,k)
    Broadcast(A(:,k), P_r(p_id))
  else :
    Receive(A(k,:))
  # Min-plus outer product
  for i in {1, 2..., n_b}:
    for j in {1, 2..., n_b}:
      if p_id owns A(i,j):
        A(i,j) ← A(i,j) ⊕ A(i,k)⊗A(k,j)
```

## 2.5 Parallel Floyd-Warshall algorithm on 2D process grid

The Algorithm 3 lists the Message Passing Interface (MPI) based Floyd-Warshall algorithm for expressing the distributed memory parallelism.

*2.5.1 Data structure.* The MPI processes are logically arranged in a two-dimensional (2D) process grid of dimension $p_x \times p_y$. On this 2D process grid, distributes the input matrix distance $A$ in a block cyclic fashion, so the block $A_{i,j}$ resides in process with coordinate $(i\%Pr,\ j\%Pc)$.

*2.5.2 Two-D distributed ParallelFw algorithm.* The outer loop of ParallelFw proceeds as the BlockedFw. However, we have additional communication steps namely DiagBcast and PanelBcast so all the processes can perform the PanelUpdate and OuterUpdate. The complete algorithm appears in Algorithm 3. The $k$-th iteration ParallelFw involves following kernels

(1) DiagUpdate($k$): The process $P_{kk}$, which owns block $A_{kk}$, perform the DiagUpdate.
(2) DiagBcast($k$): The process $P_{kk}$ broadcasts $A_{kk}$ across its process row $P_r(k)$ and its process column $P_c(k)$.
(3) PanelUpdate($k$): Each process in the $P_r(k)$ performs left multiplies $A_{k:}$ with $A_{kk}$ and each process in $P_c(k)$ performs right multiplies $A_{:k}$ with $A_{kk}$

(4) PanelBcast($k$): Each process in $P_r(k)$ broadcasts blocks of $A_{k:}$ to its process column, and each process in the $P_c(k)$ broadcasts blocks of $A_{:k}$ to its process row.
(5) OuterUpdate($k$): upon receiving $A_{k:}$ and $A_{:k}$, each process performs OuterUpdate on its local copy.

## 2.6 GPU Parallelization

Matrix multiplication over the tropical semiring is the core compute kernel for As detailed in blocked Floyd-Warshall algorithm Section 2.4. The (min, +) SRGEMM kernel follows similar acceleration opportunities like classical General Matrix Multiplication (GEMM) in terms of arithmetic intensity and data access pattern. The implementation [22], [40] realizes Min-Plus SRGEMM by extending the NVIDIA Cutlass open-source linear algebra framework [24]. According to [22, 40], the SRGEMM implementation achieves 6.81 TF/s at single precision.

## 2.7 The cost of ParallelFw

*2.7.1 Computation Cost.* There are three computational steps in ParallelFw: DiagUpdate, PanelUpdate and OuterUpdate. In the blocked FW algorithm, the total number of floating-point operations is $2n^3$ distributed among $P$ processes. Since the computation is uniform and load-balanced, hence the cost of floating-point operations is $\frac{2n^3}{P}t_f$, where $t_f$ is the cost of unit floating-point operations.

$$T_{comp} = \frac{2n^3}{P}t_f$$

*2.7.2 Communication Cost.* If $b$ is the block-size used for block-cyclic data distribution, then algorithm-2 performs the $\frac{n}{b}$ outer loop iterations. In each of the iterations, each process participates in two broadcasts $\frac{nb}{P_x}$ across process row and $\frac{nb}{P_y}$ across process column. In the ring broadcast, the total cost of the two broadcast is $2t_l + t_w(\frac{nb}{P_x} + \frac{nb}{P_y})$, where $t_l$ is the setup cost of sending a message and $t_w$ is cost of sending a unit float word. Since the outer iteration runs for $\frac{n}{b}$ iterations, hence the total communication cost is $2\frac{n}{b}t_l + t_w(\frac{n^2}{P_x} + \frac{n^2}{P_y})$.

$$T_{comm} = 2\frac{n}{b}t_l + t_w(\frac{n^2}{P_x} + \frac{n^2}{P_y})$$

So the total cost of the ParallelFw is given by

$$T_{fw} = \frac{2n^3}{P}t_f + 2\frac{n}{b}t_l + t_w(\frac{n^2}{P_x} + \frac{n^2}{P_y}) \quad (1)$$

Having described the baseline parallel Floyd-Warshall in Algorithm 3, in the next two sections, we will describe, communication and single node optimizations respectively. From the Algorithm 3, we can witness that there is a broadcast after panel and diagonal updates. These collectives come with an inherent barrier that inhibits the scalability to the large number of nodes. To overcome this problem, in Section 3, we describe techniques to hide or reduce inter-node communication and unnecessary synchronization. In Section 4, we describe optimization to improve the performance of single node computation and hide the intra-node communication between host and GPU.

**Algorithm 4** Pipelined parallel Floyd-Warshall algorithm on 2D process grid

```
def pipelinedFW(A, P = P_r × P_c):
# On each MPI process p_id do in parallel:
# First start the pipeline
diagUpdate(0);  diagBcast(0);
panelUpdate(0); panelBcast(0);
for k in {1, 2..., n_b}:
  if k ≠ n_b:
    if p_id= p_{k+1,k+1}:
      A(k+1,k+1) ← A(k+1,k+1) ⊕ A(k+1,k) ⊗ A(k,k+1)
      A(k+1,k+1) ← FW (A(k+1,k+1))
      # Broadcast A(k+1,k+1) to P_r(k+1) and P_c(k+1)
      diagBcast(k+1)
    if p_id ∈ P_r(k+1):
      #Look ahead update
      A(k+1,:) ← A(k+1,:) ⊕ A(k+1,k)⊗A(k,:)
      #Receives A(k+1,k+1) from p_{k+1,k+1}
      diagBcast(k+1)
      A(k+1,:) ← A(k+1,k+1) ⊗ A(k+1,:)
      panelUpdate(k+1)
    if p_id ∈ P_c(k+1):
      #Look ahead update
      A(:,k+1) ← A(:,k+1) ⊕ A(:,k)⊗A_{k,k+1}
      #Receives A(k+1,k+1) from p_{k+1,k+1}
      diagBcast(k+1)
      A(:,k+1) ← A(k+1,k+1) ⊗ A(:,k+1)
      panelUpdate(k+1)
  # Asynchronously launch SrGemm on GPU
  OuterProduct(k)
  #Waits for the next panels broadcast to finish
  if k ≠ n_b:
    panelBcast(k+1)
```

# 3 OPTIMIZING COMMUNICATION IN PARALLEL FLOYD-WARSHALL

In this section, we describe the design of Co-ParallelFw, which improves the performance of Algorithm 3 by optimizing the communication by pipelined scheduling, aggressively overlapping communication with computation and increasing the asynchrony among processes.

## 3.1 Data dependencies in ParallelFw :

Note that in Algorithm 3, all the steps within an iteration must be performed sequentially. In each iteration the complete distance matrix is updated, hence there is a bulk synchronous dependency between the iterations. The ParallelFw formulation in Algorithm 3 does not expose enough parallelism that allows overlapping communication with computation. So it is required to break the abstractions in Algorithm 3 to expose more parallelism.

To do so, consider the dependency between iteration $k$ and $k$+1. The OuterUpdate will update the complete matrix in the $k$-th iteration. However, the DiagUpdate and PanelUpdate of the $k$+1-th iteration only require $k$+1 panels, not the whole matrix, hence we can start DiagUpdate and PanelUpdate of the $k$+1-th iteration as

soon as the OuterUpdate update is performed on the $k$+1 panels. Thus, we can break the data dependency between the $k$ and the $k$+1 iteration by prioritizing the OuterUpdate update on the $k$+1 panels.

## 3.2 Pipelined Scheduling

Recall that the most expensive communication operation is PanelBcast and the most expensive compute operation is OuterUpdate in any iteration of Algorithm 3. The goal of the pipelined scheduling is to overlap the two operations. We use the idea of breaking the dependency between $k$ and $k$+1 iteration to achieve the overlap as follows.

Consider the execution of the $k$-th iteration of Algorithm 3. Let's assume that we have performed the PanelBcast($k$). So each process has received $k$-th horizontal and vertical panels and is ready to perform the OuterUpdate($k$). Then any process in $P_r(k{+}1)$ and $P_c(k{+}1)$ except $P_{k+1,k+1}$, will first perform the OuterUpdate($k$) on the $k$+1-th panels and wait for the DiagBcast($k$+1). Then the process $P_{k+1,k+1}$ will perform OuterUpdate($k$) on the block $A_{k+1,k+1}$, followed by DiagUpdate($k$+1) and DiagBcast($k$+1). Following that we perform the OuterUpdate($k$) on the $k$+1-th panels.

After DiagBcast($k$+1), processes in $P_r(k{+}1)$ and $P_c(k{+}1)$ can perform the PanelUpdate($k$+1), and initiate PanelBcast($k$+1). Subsequently, all the processes on $P_r(k{+}1)$ and $P_c(k{+}1)$ performs the OuterUpdate($k$) on the remaining matrix. Meanwhile, all other processes initiate OuterUpdate($k$) on the GPU and the host CPU waits for the PanelBcast($k$+1). At the end of this step, all processes have finished PanelBcast($k$+1) and OuterUpdate($k$). Likewise we perform PanelBcast($k$+2) and OuterUpdate($k$+1) concurrently in the next iteration.

In Algorithm 4 we show the pseudocode for the pipelined execution. To initialize the pipeline, we first perform the first DiagUpdate. DiagBcast, PanelUpdate and PanelBcast outside the main loop and all subsequent iterations $k$, PanelBcast($k$+1) and OuterUpdate($k$) concurrently.

## 3.3 Asynchronous execution using ring broadcast

To further improve the scheduling and asynchrony among the process, we use a variant of ring-broadcast in Co-ParallelFw. The standard library broadcast has two limitations. First, it uses a tree-based broadcast which is optimized for latency rather than bandwidth. And second, it is synchronizing in nature, i.e. acts as a barrier at the end of any iteration of Algorithm 4, so in the cases where some network links are slower due to network contention or if there are straggler processes then its impact propagates to all the processes.

We overcome these limitations by using a ring-based broadcast for PanelBcast instead of the library broadcast. In ring-broadcast, any participating process $p_i$ relays the message to its neighboring process $p_{i+1}$, and this procedure continues until all the processes have received the message. The ring-broadcast has a large latency ($\alpha$) term, i.e. any broadcast requires $p-1$ sequential steps to finish, but it is optimal in terms of bandwidth since any process needs to receive and sends only one message.

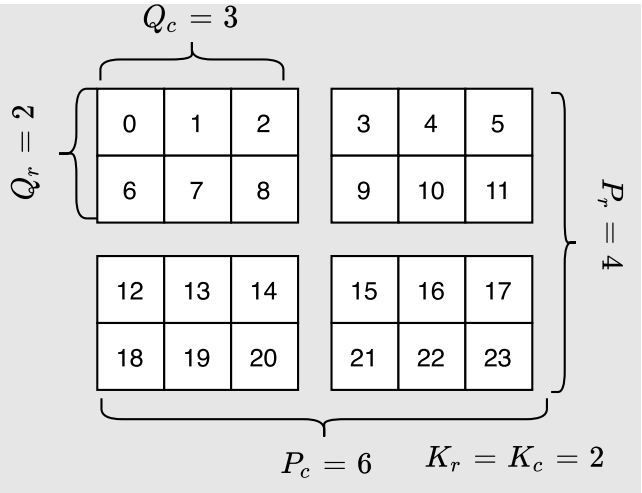


**Figure 1: Optimal rank placement for $K = 4$ and $Q = 6$. This represent the minimal inter-node communication placement for 24 MPI-processes on 4 node with 6 MPI-processes per node.**

We use the library broadcast for DiagBcast to minimize the latency since message size here is small and it is on the critical path of the computation. On the other hand, we optimize the bandwidth bounded PanelBcast using the ring-based broadcast algorithm.

Besides improving the bandwidth, the ring-broadcast shortens the critical path of ParallelFw . This is because in the ring based PanelBcast($k$), $P_r(k{+}1)$, and $P_c(k{+}1)$ receives the panels first, so they can perform the pipelined update before PanelBcast($k$) is finished. The ring-based PanelBcast introduces asynchrony between the processes across iterations, since PanelBcast($k$+1) need not wait until the completion of PanelBcast($k$) or even PanelBcast($k{-}1$). In contrast, the Algorithm 4 can overlap at most two consecutive iterations only.

## 3.4 Optimal Rank Ordering

*3.4.1 Improved models for communication cost.* We go back to the model of communication cost in Eq. (1). This model has two key limitations when used in practice.

First, the cost of sending a data $t_w$ depends on how many MPI processes are spawned in a node. Since all the processes share a single network interface (NIC), if multiple MPI processes are spawned, per process bandwidth will be decreased and $t_w$ will increase. Second, this model does not capture intranode data movement. For instance, if all processes in a communicator are within a node, then for a collective operation the effective $t_w$ will be much smaller compared to the instance where the communicator spawns multiple nodes.

A more accurate model of communication costs by considering total data sent outside of NIC. Let assume that we have MPI grid of dimension $P = P_r \times P_c$ and we have $Q$ processes per-node draw from MPI grid in dimension $Q_r \times Q_c$. Typically, while creating MPI 2D grid, every node was assigned with continuous MPI-process ranks hence the typical configurations are $1 \times Q$ or $Q \times 1$.

We define the logical *node-grid* of dimension $K_r \times K_c$ where $K_r = P_r/Q_r$ and $K_c = P_c/Q_c$. Suppose we had one MPI process per node, so $Q_r = Q_c = 1$ and $P_r = K_r$ and $P_c = K_c$, then we can use Eq. (1). and obtain the communication cost $T_{comm}$ (neglecting the latency term) as

$$T_{comm} = t_w(\frac{n^2}{K_r} + \frac{n^2}{K_c}).$$

For any choice of the number of MPI process per node $Q = Q_r \times Q_c$ and for a given $K$ number of nodes in $K_r \times K_c$, the above equation represents the lower bound on the amount of data transferred via from a node in ParallelFw. Hence for a given MPI process grid $P = P_r \times P_c$ with $Q = Q_r \times Q_c$ MPI processes sharing a node's NIC, the lower bound on communication is

$$T_{comm} = t_w(\frac{n^2 Q_r}{P_r} + \frac{n^2 Q_c}{P_c}).$$

where $t_w$ = Size in byte for unit data/NIC-bandwidth.

*3.4.2 Optimal Rank Placement.* To minimize the communication volume per Section 3.4.1, we should have

$$K_r \approx K_c \tag{2}$$

The latency term only depends on $P_r$ and $P_c$. To minimize the latency cost we should have

$$P_r \approx P_c. \tag{3}$$

To satisfy both Eqs. (2) and (3), we should also have $Q_r \approx Q_c$. We place ranks within a node to achieve optimal $Q_r$ and $Q_c$ as shown in Figure 1. Such a placement can be specified using MPICH_RANK_ORDER or by using an explicit resource file in the case of the Summit supercomputer.

# 4 SINGLE NODE OPTIMIZATION

## 4.1 Semiring Matrix multiplication on GPU

We implemented semiring matrix multiplication on GPU SrGemm using Nvidia's Cutlass template library for constructing efficient GEMM type of operation on Nvidia GPUs[1]. While Cutlass does not directly support semi-ring algebra as yet, we modified it suitably to do so. Our SrGemm kernel achieves single-precision 6.8TFlop/sec on Nvidia V100 GPU. The theoretical single-precision peak of V100 is 15.7 TFlop/sec, SrGemm can not use the fused multiply and add (FMA) units available on V100, hence theoretical peak for SrGemm is 7.8TFlop/sec. The SrGemm kernel is central to high performance for Floyd-Warshall, we consider SrGemm as a black-box unit in this work. We plan to opensource the SrGemm code and provide a detailed performance analysis of SrGemm beyond min-plus semiring in the near future.

## 4.2 DiagUpdate on GPU

The cost of DiagUpdate in the BlockedFw computation is $2nb^2t_f$, which we often ignore as typically $b \ll n$. However, at extreme strong scaling case when $P = O\left(\frac{n^2}{b^2}\right)$, the cost of DiagUpdate can not be ignored. To achieve good strong-scaling we must also perform the DiagUpdate on the GPU. The DiagUpdate is the semi-ring equivalent of matrix-inversion. To express DiagUpdate using SrGemm, we use the following relation for computing transitive closure which is semi-ring equivalent of Neuman series for matrix inversion:

$$\textsc{DiagUpdate}(A) = \bigoplus_{\dim(A)} A^i. \tag{4}$$

Note that Eq. (4) can be computed with $\log_2(\dim(A))$ matrix-matrix multiplications. This, however, increases the asymptotic cost of DiagUpdate to $O(nb^2 \log b)$, in practice it significantly speeds up DiagUpdate due to relatively higher GPU performance.

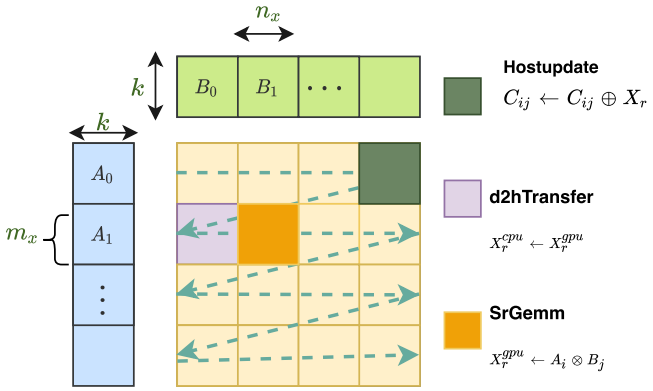


**Figure 2: Execution order and pipelines scheme for ooGSrGemm. Steps of SrGemm, d2hXfer and hostUpdate execute in parallel to mask the memory transfer cost.**

## 4.3 Out-of-GPU semi-ring matrix multiplication

So far we have assumed that complete local distance matrix fits in the GPU memory, and many of the optimizations we have used in Co-ParallelFw require that to work. However, when the local matrix does not fit in the GPU memory then we can not compute Apsp. This poses a significant limitation on the size of the problem that our algorithm can handle. In the section we describe a memory-efficient flavor of algorithm 3 Me-ParallelFw that doesn't have such limitation. However, such improvement comes at cost of increased data transfer between the host and the GPU typically via NVLink or PCIe. We show that by choosing the parameters correctly we can significantly reduce this penalty. In the heart of Me-ParallelFw, lies *out-of-GPU* semi-ring matrix multiplication kernel (ooGSrGemm). In this section, we describe the design and analysis of ooGSrGemm.We focus on OuterUpdate since its the most computationally demanding substep in ParallelFw .

Consider the semiring matrix multiplication $C \leftarrow C \oplus A \otimes B$ of two panels $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$ which is accumulated on $C \in \mathbb{R}^{m \times n}$. We are interested in the case where $m, n \gg k$ and $m \times n$ is so large that it does not fit in the GPU memory. The idea of ooGSrGemm is to divide the SrGemm into smaller chunks of size $m_x \times n_x$ so it can easily fit in GPU memory.

We assume that $A$ and $B$ are already in GPU memory. We divide $A$ by rows into $A_0, A_1 \ldots A_{m_b-1}$ where $m_b = m/m_x$ and each $A_i$ has a dimension $m_x \times k$; and $B$ by columns into $B_0, B_1 \ldots B_{n_b-1}$ where $n_b = n/n_x$ and each $B_j$ has a dimension $k \times n_x$. Using GPU buffer $X$ of size $m_x \times n_x$, we perform the following steps:

- SrGemm to compute $X \leftarrow A_i \otimes B_j$;
- d2hXfer: transfer the compute X from the GPU to host; and
- hostUpdate: $C_{ij} \leftarrow C_{ij} \oplus X$.

Since each of the three steps involves different hardware, we can potentially overlap the three. We use cudaStream API to do so. In a single cudaStream all the tasks will be performed sequentially but cudaStreams are asynchronous to each other. So when one cudaStream is computing SrGemm, the other stream performs d2hXfer.

## 4.4 Hiding GPU to host transfer cost

To use $s$ cudaStreams, we initialize $s$ buffers $X_0, X_1 \ldots, X_{s-1}$ each of size $m_x \times n_x$. The SrGemm and d2hXfer of any block $C_{ij}$ are performed in a single stream. The blocks are assigned to streams in a round-robin fashion.

Any cudaStream $r$ computes $X_r \leftarrow Ai \times B_j$ and sends it to the host. The host waits for cudaStream in the order they were initiated. So when the host receives $X_r$, it performs the hostUpdate $C_{ij} \leftarrow C_{ij} \oplus X_r$. Now the host initializes SrGemm and d2hXfer for another block on $r$-th cudaStream, if there are any blocks left to be updated. The host now waits for the $r$+1-th stream to finish the data transfer.

We can also hide the cost of transferring $A$ and $B$ by pipelining it. To do so, instead of sending the complete $A$ and $B$ matrices we send $A_i$ when we are performing update of $C_{i,0}$, and send $B_j$ when we are performing update of $C_{0,j}$. $A_i$ and $B_j$ needs to sent only once, and we reuse $A_i$ and $B_j$ when performing update on any blocks $C_{i,j}$

## 4.5 Cost of SrGemm in offload-model

The total number of flops performed in ooGSrGemm is $2mnk$, so the total cost of SrGemm ($t_0$) is $t_0 = 2mnkt_f$, where $t_f$ is time to perform a floating point operation. The total data sent from host to device and device to host is $(m+n)k$ and $mn$ respectively. So the cost of the data transfer between host and device ($t_1$) is $(mn+nk+mk)t_{hd}$, where $t_{hd}$ is the cost of sending unit data between host and device. And finally, the performance of hostUpdate is limited by the CPU-DRAM memory bandwidth. Since hostUpdate performs $2mn$ reads ($C$ and $X$) and $mn$ writes ($C$) between CPU and DRAM, the cost of the hostUpdate ($t_2$) is $3mnt_m$ where $t_m$ is the cost of unit DRAM to CPU transfer.

If we use a single cudaStream then we will not be able to overlap either of the three steps so the cost will be $t_0 + t_1 + t_2$. With two streams we can only overlap one substep with another two so the total cost will be $\min\left\{\max_{i,j,k}\left\{t_i, t_j + t_k\right\}\right\}$ where $i, j, k \in \{0, 1, 2\}$. And with three or more streams, all three substeps can be overlapped so the cost of ooGSrGemm will be $\max\{t_0, t_1, t_2\}$

To achieve the peak flop-rate we should have cost of either d2hXfer or hostUpdate be lower than the cost of SrGemm. So

$$t_0 \geq \max\{t_1, t_2\}$$

In our case, $m, n \gg k$ so the cost of the data transfer between host and device can be approximated as $t_1 \approx mnt_{hd}$. So we can simplify above equation:

$$k \geq \max\left\{\frac{t_{hd}}{2t_f}, 3\frac{t_m}{2t_f}\right\} \tag{5}$$

# 5 EXPERIMENTS AND EMPIRICAL RESULTS

We perform experiments to understand the individual impact of each optimization as well as how they work in cohesion.

## 5.1 Setup

We describe experimental details such as the testbed, data, metrics, and programming environment that we used to explain the observations. In all cases, we experimentally confirmed that the output of our revised implementations match outputs (results) of the sequential Floyd-Warshall baseline.

*5.1.1 Testbed.* The Summit system consists of 4,608 nodes. Each node has two 22-core IBM POWER9 processors and six NVIDIA Volta V100 GPUs, connected by NVLINK-2, which has a peak performance bidirectional bandwidth of 100 GB/s. V100 GPU has 5,120 cores operating at 1.53 GHz, which translates to theoretical peak of 7.85 TF/s and 15.7 TF/s in single precision with and without FMA instructions. The peak memory bandwidth of each V100 GPU is 900 GB/s. Each node contains 512 GB main memory, while each GPU contains 16 GB HBM2 memory. The nodes are connected with a Mellanox Infiniband fat-tree interconnect which has an effective bandwidth of 25 GB/s per node.

*5.1.2 Legends.* In this section, we discuss the different legends Baseline, Pipelined, +Reordering, +Async, and offload that appear in the plots.

- **Baseline**: represents the implementation of Algorithm 3.
- **Pipelined**: Algorithm 4 which overlaps communication with computation.
- **+Reordering**: Pipelined with optimal rank reordering discussed in Section 3.4.
- **+Async**: +Reordering with the asynchronous ring broadcast discussed in Section 3.3.
- **offload**: the memory-efficient flavor of Algorithm 3 outlined in Section 4.

*5.1.3 Metrics.* For reporting absolute performance, we use the normalized metric Flops/sec depending upon the scale of the problem. In the case of a single GPU, we use *GigaFlops/Sec* and for multiple GPUs, we always report *PetaFlops/sec*. We use *effective bandwidth per node* in GB/Sec when reporting performance of individual communication optimizations. The effective bandwidth per node is computed as $\frac{W_{min}}{t_{FW}}$, here $W_{\min}$ is the *theoretical* minimum per-node communication volume among all the configurations for given problem size and the number of nodes, and $t_{FW}$ is the total time spent in ParallelFw.

*5.1.4 Test Graphs.* The entire experimentation was conducted on a dense uniform random matrix.

*5.1.5 Programming Environment.* The software versions used are GCC 6.4.0, IBM Spectrum MPI 10.2.0.0, and CUDA 10.1.243. Summit's `jsrun` tool is used for application launch. No other proprietary software was used in the execution.

## 5.2 Impact of Communication Optimizations

*5.2.1 Optimal Rank Placement.* In this section, we evaluate different rank placement schemes discussed in Section 3.4.1. Recall the $K$, $P$ and $Q$ parameters from Section 3.4.1: $P_r$, $P_c$ are the dimensions of MPI grid; $Q_r$, $Q_c$ are the dimensions node local MPI grid; and $K_r$, $K_c$ are the dimensions of node grid; and $K_r = P_r/Q_r$ and $K_c = P_c/Q_c$. Given a node count, we explore the different combinations of $P_r, P_c, K_r, K_c, Q_r, Q_c$ and measure the effective per node bandwidth GB/Sec.

The Figure 3 shows the effect of rank reordering by sweeping the $P_r, P_c, K_r, K_c$ on every node count for $n = 196,608$ vertices. We observe that for a given number of nodes, the maximum effective bandwidth is always achieved when $K_r \approx K_c$. For instance when node=4, best performance occurs at $K_r = K_c = 2$. Similarly, the worst performance occurs when $K_r$ and $K_c$ are far-off. Note that in the single-node case, the best effective bandwidth is higher than the theoretical limit of 25 GB/s since all the communications are within a single node.

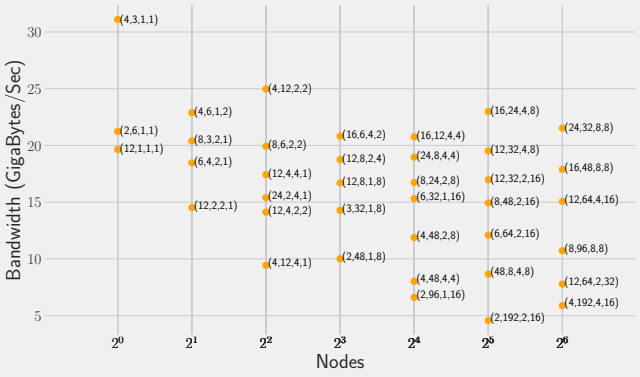


**Figure 3: Effect of Rank Reordering. The plot shows obtained bandwidth for different combination of $P_r, P_c, K_r, K_c$ on every node count for 196,608 vertices.**

*5.2.2 Evaluation of Different Communication Strategies.* We evaluate the impact of different communication optimizations discussed in Section 3.3 and Section 3.2. The Figure 4 show the effective bandwidth achieved for Pipelined, +Reordering, +Async and the Baseline variants. We vary the problem size from 26k to 524k on 64 nodes of Summit. Note that our effective bandwidth calculations are meaningful when the execution time is dominated by inter-node communication time. Hence when the problem size is small, execution time is dominated by bandwidth cost. Whereas for large problem size the execution time is dominated by compute time. On 64 nodes, 120k is the theoretical estimate of the smallest problem size when Floyd-Warshall becomes compute-bound.

When the execution time is dominated by communication time, we observe that Pipelined achieves better effective bandwidth compared to the Baseline as it hides the cost of computation. Whereas +Reordering reduce the communication cost in addition to hiding the computation cost. Furthermore +Async reduce the synchronization cost on top of all former optimizations. In the best case, our

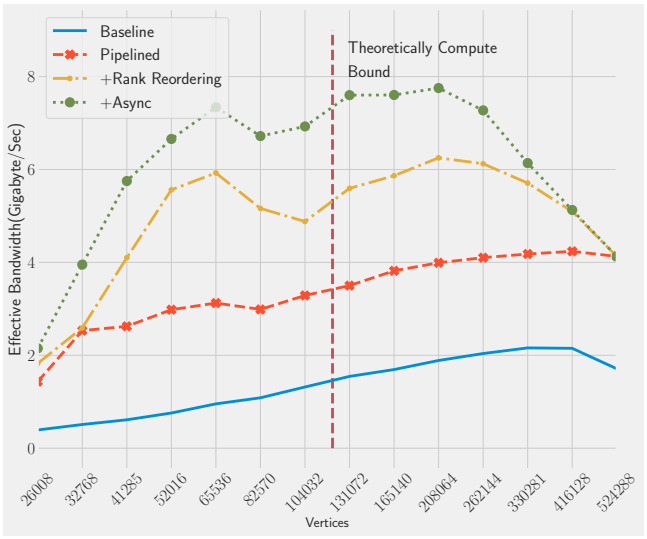


**Figure 4: The effect of optimizing communication with pipelined, asynchronous and rank reordering on ParallelFw for various vertices in 64 nodes**

implementation that encompasses all the optimizations is achieves four times higher effective bandwidth.

## 5.3 Impact of Single Node Optimizations

*5.3.1 Performance of Offload Model SRGEMM on Single GPU.* We built the ooGSrGemm micro benchmark to understand its performance on a single GPU. The performance of ooGSrGemm is heavily dependent on (a) Input size $m, n$ (b) Block Size $b$ and (c) GPU Max Rows $m_x, n_x$. For simplicity, we are assuming $m = n$ and $m_x = n_x$.

First, we find the minimum block size for the ooGSrGemm. So for different $m_x \in \{512, 1k, 2k, 4k\}$, we vary the block size in Section 5.3.1, and observe the performance. We observe that for block size > 768 ooGSrGemm performs very close to the peak for all $m_x$. Per Eq. (5), we estimate minimum block size of 624 assuming NVlink's bandwidth= 50 GB/s and peak flop rate =7.8TFlops. So our model's prediction is very close to the observed block size.

Recall that $m_x$ and $n_x$ are the dimensions for buffer. We assume $m_x = n_x$ and vary $m_x$ and the input size $n$ = #vertices block size $b = 768$. From Figure 6, we can observe that, in a single GPU, the ooGSrGemm performance is close to peak even for buffers of dimension $2k \times 2k$ if $n$ is sufficiently large.

We evaluate the performance of the end-to-end ParallelFw with the different optimizations. All the experiments in Sections 5.4 and 5.5, utilized the entire 6 GPUs in a node and used 2 MPI ranks per GPU.

## 5.4 Performance on 64 nodes

We compare all the optimizations to Baseline by varying the input size from 16e3 to 1.6e6 vertices on 64 nodes shown in Figure 7. When the vertex size is less than 208k, ParallelFw is network bandwidth bound so Co-ParallelFw has higher relative performance. As the number of vertices increases, ParallelFw becomes compute-bound so in such cases, we do not see the advantage of

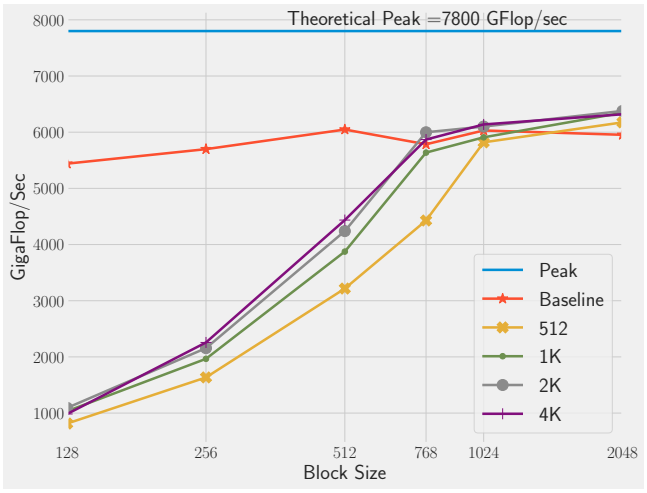


**Figure 5: Offload out-of-gpu Srgemm performance with respect to block size**

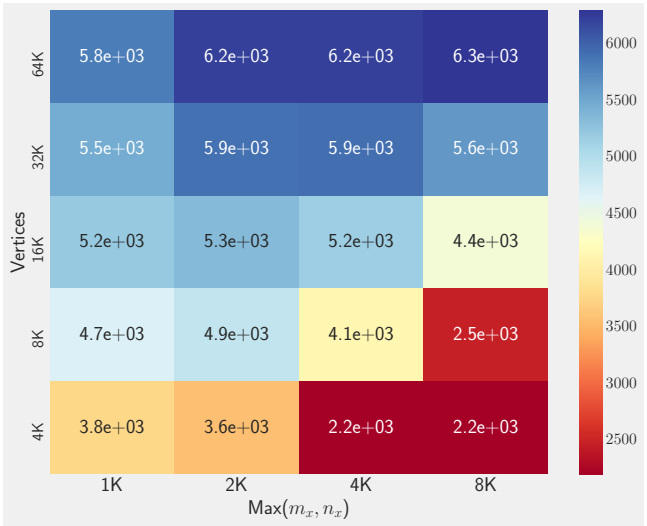


**Figure 6: out-of-gpu Srgemm performance in Gigaflops/sec for different $m$ (the operand size) and $m_x$ GPU buffer's dimension.**

communication optimizations. Also note that all other implementations except offload can run only up to 524k vertices whereas, with Me-ParallelFw, we can push it to 1.6 Million vertices while achieving 50% of peak theoretical throughput.

## 5.5 Strong and Weak Scaling

*5.5.1 Strong Scaling.* We evaluated performance on 16 to 256 nodes for $n = 300,000$ vertices and present the performance in Figure 8. The Co-ParallelFw implementation achieves 45% of parallel efficiency. Note that, in 16 nodes Co-ParallelFw is 1.6x faster over the baseline and it is 4.6x faster on 256 nodes. This is expected since at

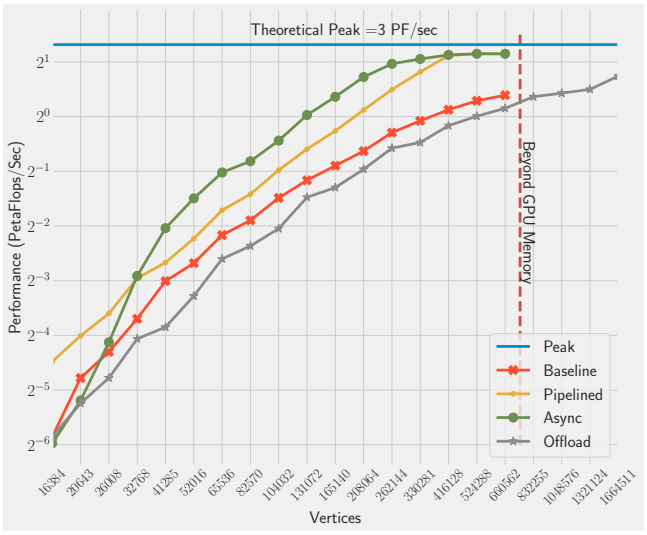


**Figure 7: ParallelFw Performance with different optimization on 64 nodes. The performance was measured by sweeping the number of vertices.**

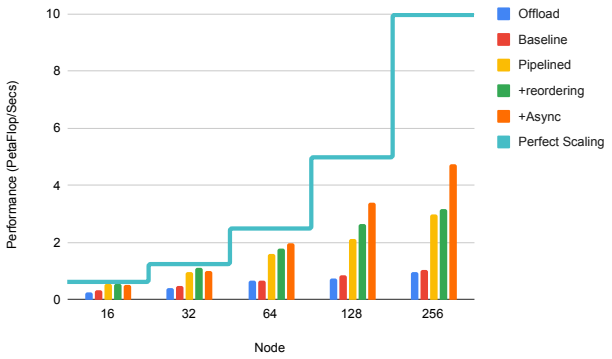


**Figure 8: Strong Scaling of ParallelFw on 300,000 vertices.**

the higher node count effect of communication cost is pronounced, that Co-ParallelFw handles gracefully.

*5.5.2 Weak Scaling.* For weak scaling experiment, we keep the workload $O(\frac{n^3}{p})$ constant. The experiments was conducted with 300,000 vertices on 16 nodes and scale accordingly to 256 nodes. From the Figure 9, Co-ParallelFw shows perfect weak scaling, whereas for offload and baseline do not scale well. This is attributed to fact that baselines and offload do not actively hide the communication.

## 6 RELATED WORK

Besides Floyd-Warshall, Johnson's algorithm[21] which computes single-source shortest path (Sssp) from all the vertices is another popular method used in HPC. It can achieve the lowest asymptotic complexity of $O(mn + n^2 \log n)$ if Dijkstra's algorithm[11] with the Fibonacci heap[16] is used for Sssp. When the input graph

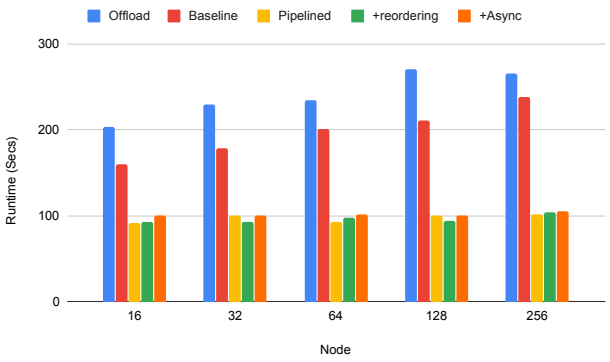


**Figure 9: Weak Scaling of ParallelFw begin with 300,000 vertices on 16 nodes. The workload $O(\frac{n^3}{p})$ per node is kept constant**

is sparse i.e. $m = O(n)$, it becomes an attractive alternative to Floyd-Warshall. Dijkstra's algorithm for Sssp uses a priority queue data structure which is difficult to parallelize for massively threaded architecture. Another choice for Sssp inside Johnson's algorithm is Bellmen-Ford[5, 15], which is an embarrassingly parallel computation but may not be work optimal. The Delta-stepping algorithm [28] of Scott and Meyers is a hybrid of Dijkstra and Bellmen-Ford that provides more parallelism than Dijkstra's algorithm and performs fewer operations than Bellman-Ford. For undirected graphs with positive integer weights, Thorup algorithm[41] is a theoretically optimal algorithm. On graphs with multiple components one may use graph connected-components algorithm[30], and perform Apsp on each connected component of the graph. Out-of-memory GPU computation of these graph kernels has been explored by Gera et al. [17]. Optimized implementations of these Sssp routines available in many popular graph packages such as Boost Graph Library (BGL) [36], Galois [29], and Graphmat [39] for CPUs, and cuGraph [2] and Gunrock [44] for GPUs. Single-node performance comparison of these approaches with Floyd-Warshall may be found elsewhere [10, 22, 31].

Based on the original Floyd-Warshall, the first 2D distributed-memory algorithm for the APSP without blocking using $n$ global synchronization is attributed to Jenq and Sahni [20]. Kumar & Singh [25] analyzed the scalability of different Apsp algorithms and showed overlapping communication with computation in the non-blocked case. Solomonik et al. proposed a communication avoiding parallel Apsp which uses the divide and conquer approach on 2.5D process grid [37]. But in absolute performance, it achieved only about 10 to 25% of peak, with maximum tested problem sizes of n = 65,536. A distributed GPU Apsp showed good performance for smaller clusters [12]. Also, their centralized communication scheme limits their scalability beyond 64 GPUs.

The Floyd-Warshall based Apsp shares structural similarity with High Performance Linpack (HPL) and therefore is more amenable to some of the parallelization techniques such as look ahead and customized broadcast explored in HPL library [3, 7, 13, 19, 38, 43]. It's no surprise that our Apsp implementation Co-ParallelFw achieves similar efficiency as HPL benchmark on Summit. For sparse matrices, similar optimizations have been explored in [32–34]. Such optimizations can be combined with sparse Floyd-Warshall approaches as [31].

The GraphBLAS Forum [23] is an open effort to define standard building blocks for graph algorithms in the language of linear algebra. This decouples the realization of graph algorithms independent of the distributed performance and scalability. The first distributed realization of GraphBLAS on MPI runtime using C++ was CombBLAS [6]. Recently, LAGraph [27], provided a distributed Scala API on Spark runtime based on GraphBLAS. However, the APSP algorithm implemented in LAGraph [27] and uses an outer product formulation equivalent to SGER of level-II BLAS, which will not be as efficient as BlockedFw on GPUs.

Apsp is theoretically an important problem as a number of other problems are equivalent to Apsp e.g. metricity, minimum-weight triangle, second shortest path etc.[45, 46]. While Apsp is the semiring-equivalent of matrix inversion, no truly sub-cubic (Strassen-like) algorithm for Apsp is known. Seidel[35] showed a way to use fast matrix multiplication algorithms, such as Strassen's algorithm, for the solution of the APSP problem by embedding the semiring into a ring. The best known complexity of Apsp for the dense case is $O\left(n^{3-o(1)}\right)$ [45] and $O\left(\frac{mn}{\log n}\right)$ for sparse graphs [9]. For the parallel case, the complexity is $O(\log n)$ due to Tishkin [42]. The seminal work of Carre and others establishes the equivalence between finding shortest paths and solving a system of linear equations [4, 8]. There are several modern treatments of this subject as well [18, 26].

## 7 CONCLUSIONS AND FUTURE WORK

The performance-improvement methods we explored are inspired by techniques from parallel dense linear algebra. Their application to dense Floyd-Warshall significantly improves its inherent strong and weak scalability and, critically, overcomes the memory capacity limits imposed by existing GPU designs, which apportion relatively low per-device memory capacities compared to their host nodes. Furthermore, our scaling results on Summit should extend to other systems, and the performance models we derived can guide their tuning when porting ParallelFw to any accelerated architecture.

For extremely large and sparse graphs, alternatives such as Johnson's algorithm will be competitive to Floyd-Warshall but cannot exploit GPUs. Hence, even for sparser graphs, we expect the performance gap between Johnson's and Floyd-Warshall will continue to shrink [31]. Our optimizations for ParallelFw are also applicable to other Floyd-Warshall-based approaches that exploit the structure and sparsity of the graph [12, 31].

Currently, we plan to extend this work to support distributed shortest path generation and incremental Floyd-Warshall, which are critical in applications, and add support of structured sparse graphs, where exploiting sparsity becomes paramount [31].

## 8 ACKNOWLEDGEMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Robinson Pino, program manager, under contract number DE-AC05-00OR22725, as well as by the National Science Foundation under Grant Nos. 1533768 and 1710371. This research used resources of the Oak Ridge Leadership Computing Facility,

which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

## REFERENCES


[1] Nvidia/cutlass: Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass. (Accessed on 01/24/2021).

[2] rapidsai/cugraph: cugraph - rapids graph analytics library. https://github.com/rapidsai/cugraph. (Accessed on 01/24/2021).

[3] Matthias Bach, Matthias Kretz, Volker Lindenstruth, and David Rohr. Optimized HPL for AMD GPU and multi-core CPU usage. *Computer Science-R&D*, 26(3-4):153–164, 2011.

[4] Roland C Backhouse and Bernard A Carré. Regular algebra applied to path-finding problems. *IMA Journal of Applied Mathematics*, 15(2):161–186, 1975.

[5] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[6] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[7] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The impact of multicore on math software. In *International Workshop on Applied Parallel Computing*, pages 1–10. Springer, 2006.

[8] Bernard A Carré. An algebra for network routing problems. *IMA Journal of Applied Mathematics*, 7(3):273–294, 1971.

[9] Timothy M Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 514–523. Society for Industrial and Applied Mathematics, 2006.

[10] Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. Phast: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[11] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

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

[13] Massimiliano Fatica. Accelerating Linpack with CUDA on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 46–51. ACM, 2009.

[14] Jeremy T. Fineman and Eric Robinson. Fundamental graph algorithms. In Jeremy Kepner and John Gilbert, editors, *Graph Algorithms in the Language of Linear Algebra*, chapter 5, pages 45–58. Society of Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011.

[15] Lester R Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

[16] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[17] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. Traversing large graphs on gpus with unified memory. *Proceedings of the VLDB Endowment*, 13(7):1119–1133, 2020.

[18] Michel Gondran and Michel Minoux. *Graphs, dioids and semirings: new models and algorithms*, volume 41. Springer Science & Business Media, 2008.

[19] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, George Chrysos, and Pradeep Dubey. Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel® Xeon Phi coprocessor. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 126–137. IEEE, 2013.

[20] Jing Fu Jenq and Sartaj Sahni. All pairs shortest paths on a hypercube multiprocessor. In *Proc Int Conf Parallel Process 1987*, pages 713–716. Pennsylvania State Univ Press, 1987.

[21] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[22] R. Kannan, P. Sao, H. Lu, D. Herrmannova, V. Thakkar, R. Patton, R. Vuduc, and T. Potok. Scalable knowledge graph analytics at 136 petaflop/s. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–13, Los Alamitos, CA, USA, nov 2020. IEEE Computer Society.

[23] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[24] Andrew Kerr. Cutlass: Cuda templates for linear algebra subroutines, November 2019.

[25] Vipin Kumar and Vineet Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, 1991.

[26] Grigory L Litvinov. Idempotent/tropical analysis, the hamilton–jacobi and bellman equations. In *Hamilton-Jacobi equations: approximations, numerical analysis and applications*, pages 251–301. Springer, 2013.

[27] Tim Mattson, Timothy A Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. Lagraph: A community effort to collect graph algorithms built on top of the graphblas. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 276–284. IEEE, 2019.

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

[29] Keshav Pingali. High-speed graph analytics with the galois system. In *Proceedings of the first workshop on Parallel programming for analytics applications*, pages 41–42. ACM, 2014.

[30] Piyush Sao, Oded Green, Chirag Jain, and Richard Vuduc. A self-correcting connected components algorithm. In *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, pages 9–16. ACM, 2016.

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

[32] Piyush Sao, Xiaoye S Li, and Richard Vuduc. A communication-avoiding 3d algorithm for sparse lu factorization on heterogeneous systems. *Journal of Parallel and Distributed Computing*, 131:218–234, 2019.

[33] Piyush Sao, Xing Liu, Richard Vuduc, and Xiaoye Li. A sparse direct solver for distributed memory Xeon Phi-accelerated systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 71–81. IEEE, 2015.

[34] Piyush Sao, Richard Vuduc, and Xiaoye Sherry Li. A distributed CPU-GPU sparse direct solver. In *European Conference on Parallel Processing*, pages 487–498. Springer, 2014.

[35] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences*, 51(3):400–403, 1995.

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

[37] Edgar Solomonik, Aydın Buluç, and James Demmel. Minimizing communication in all-pairs shortest paths. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, MA, USA, 5 2013.

[38] Peter Strazdins et al. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. 1998.

[39] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241*, 2015.

[40] V. Thakkar, R. Kannan, P. Sao, H. Lu, D. Herrmannova, , R. Patton, R. Vuduc, and T. Potok. Dense semiring linear algebra on modern cuda hardware. In *SIAM Computational Sciences and Engineering*. SIAM, 2021.

[41] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[42] Alexandre Tiskin. All-pairs shortest paths computation in the bsp model. In *International Colloquium on Automata, Languages, and Programming*, pages 178–189. Springer, 2001.

[43] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5):232–240, 2010.

[44] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):1–49, 2017.

[45] R Ryan Williams. Faster all-pairs shortest paths via circuit complexity. *SIAM Journal on Computing*, 47(5):1965–1985, 2018.

[46] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 645–654. IEEE, 2010.