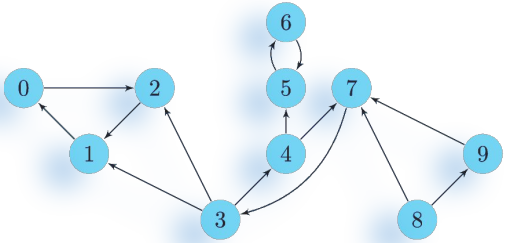
Accelerating large graph algorithms on GPU using CUDA

Archisman Pathak
Kousshik Raj
Koustav Chowdhury
Rounak Patra
Satyam Porwal
Siddhant Agarwal
Sriyash Poddar

Table of Contents

01
Problem and Motivation

Understanding the Problem



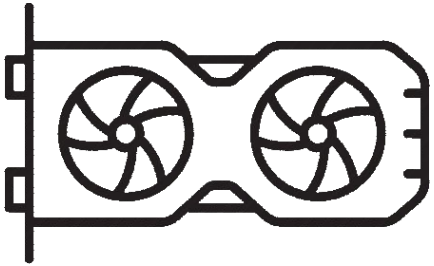
Graph Algorithms

Graph algorithms are used to develop intelligent solutions and enhance various machine learning models.

Impractical Sequential Algorithms

Fast implementations of sequential graph algorithms are fast, but the hardware used in them is very expensive.

Understanding the Motivation



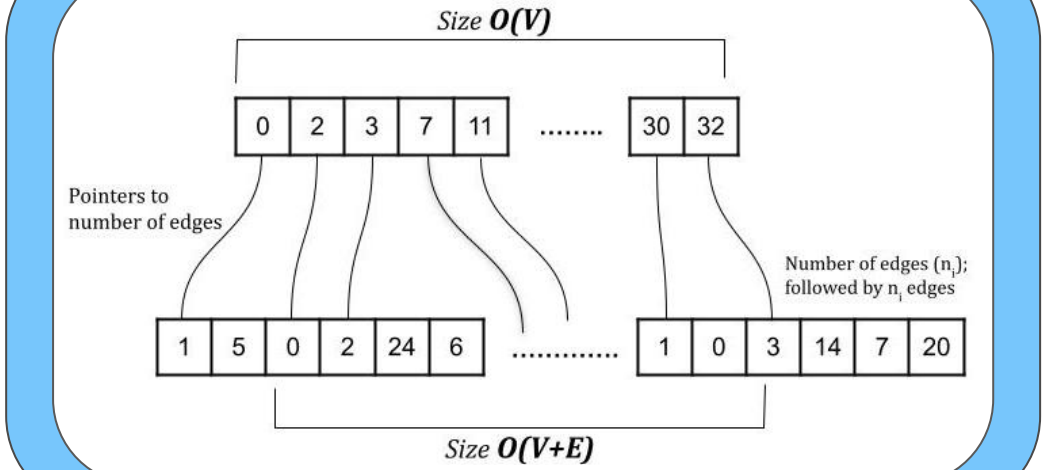
CUDA

Nvidia CUDA provides a development environment for creating high performance GPU-accelerated applications.

Previous Works in Parallel Graph Algorithms

Previous works in parallel graph algorithms achieved practical times on basic graph operations but at high hardware cost.

CUDA Graph Representation



- Each value of posV contains an index of the posE array
- The pointer contains the number of neighbours of the node, say *n*.
- The next *n* elements are the ids of neighbouring vertices.

Graphs used for Experiments

APSP Graphs
(Due to time and memory constraints restricted the size)
*Graphs are numbered in increasing number of vertices

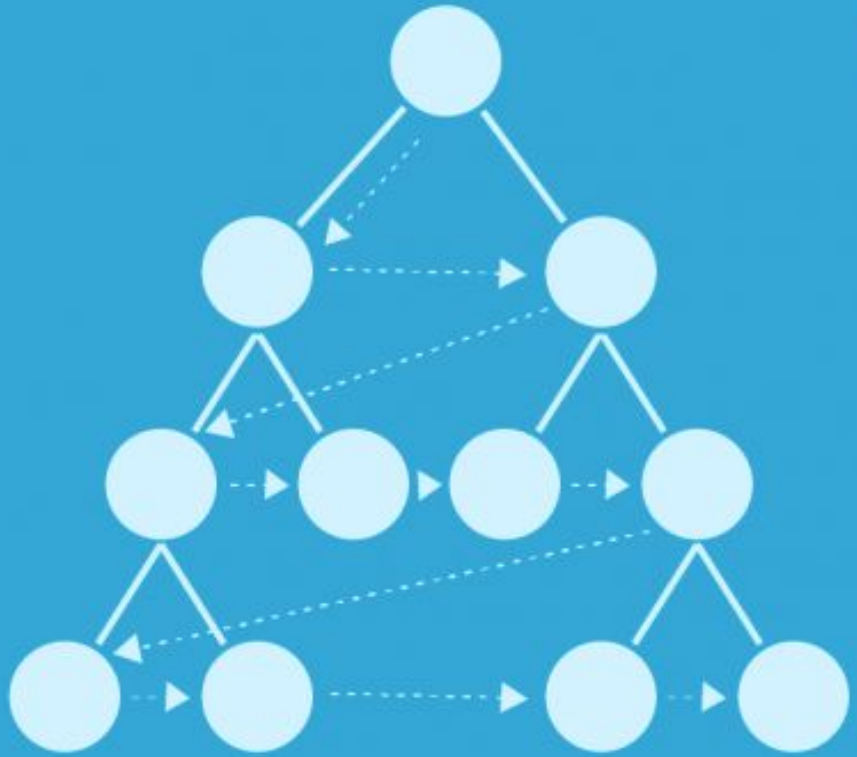
Graph Number	# Nodes	# Edges	Average Degree
1	2,700	1,808,853	1,340
2	4,039	88,234	44
3	4412	108,818	49
4	5,881	21,492	7
5	7,500	837,083	233

Graphs used for Experiments

BFS and SSSP Graphs
(No constraints on the size)
*Graphs are numbered in increasing number of vertices

Graph Number	# Nodes	# Edges	Average Degree
1	2,700	1,808,853	1,340
2	21,853	12,323,648	1,128
3	82,168	504,230	12
4	1,694,616	11,094,209	13
5	16,777,214	132,557,200	16

BREADTH FIRST SEARCH



Our Implementations

Parallel BFS

Queue BFS

Scan BFS

First Approach: Parallel BFS

Host

- Level-synchronous approach
- Runs O(V^2 +E) operations
- *Vertex frontiers* : Nodes that are currently being visited
- *Edge frontiers* : Nodes that will be visited in the next iteration
- *flag* checks for termination

Algorithm 1 parallelBFS_Host

1: **Input:** V_a, E_a, S ▷ The graph $G(V, E)$ and source S
2: Create distance array $Dist_a$, and parent array P_a of size $|V|$
3: Initialise all elements of $Dist_a, P_a$ to ∞
4: $D_a[S] = 0$
5: $level = 0$
6: $flag = True$
7: **while** $flag$ **do**
8: $flag = False$
9: Invoke parallelBFS_kernel($level$, V_a,E_a,$Dist_a$,$flag$).
10: $level = level + 1$

First Approach: Parallel BFS

Kernel

- Threads are launched for each vertex
- Each vertex checks if it is *frontier vertex*
- If yes, it updates the distances of neighbours and populates the *edge frontiers*
- Terminates when no *frontier vertex* updates its neighbour

Algorithm 2 parallelBFS_kernel

```
1:  Input: level, V_a, E_a, Dist_a, flag
2:  tid = getThreadID
3:  f = False
4:  if tid < Va_size and Dist_a[tid] = level then
5:      u = tid
6:      for all v = neighbours of u do
7:          if level + 1 < Dist_a[v] then
8:              Dist_a[v] = level + 1
9:              f = True
10:     if f = True then
11:         flag = True
```

Second Approach: Queue BFS

Host

- Level-synchronous
- Runs O(V + E) operations
- *Vertex frontier* and *edge frontier* is maintained in form of a queue
- Intuition is similar to sequential BFS.
- Terminates when there are no *vertex frontiers*.

Algorithm 3 queueBFS_Host

1: **Input:** V_a, E_a, S ▷ The graph $G(V, E)$ and source S
2: Create cost array $Dist_a$ and parent array P_a of size $|V|$ and initialise all values to ∞
3: Create two array cQ and nQ, and initialise it to S and *null* respectively.
4: $Dist_a[S] = 0$
5: $P_a[S] = -1$
6: $l = 0$ ▷ Start with the source vertex
7: **while** $cQ_{size} > 0$ **do**
8: Invoke queueBFS(l, V_a, E_a, $Dist_a$, P_a, cQ, nQ)
9: $swap(cQ, nQ)$
10: Set nQ to *null*
11: $l = l + 1$

Second Approach: Queue BFS

Kernel

- Threads are launched for each node in *vertex frontier queue*
- For all the neighbours of the node, update the distance if the node can be reached in fewer steps
- Add that node in the *edge frontier queue*
- Involves atomic operations

Algorithm 4 queueBFS_kernel

1: **Input:** $l, V_a, E_a, Dist_a, P_a, cQ, nQ$ ▸ The graph $G(V, E)$ and source S
2: tid = getThreadID
3: **if** $tid < cQ_{size}$ **then**
4: $u = cQ[tid]$
5: **for all** v = neighbours of u **do**
6: **if** $Dist_a[v] = \infty$ and $atomicMin(Dist_a[v], l + 1) = \infty$ **then**
7: $P_a[v] = u$
8: $pos = atomicAdd(nQ_{size}, 1)$
9: $nQ[pos] = v$

Third Approach: Scan BFS

Host

- Level-synchronous approach
- Needs 4 global synchronization
- Perform O(V + E) operations
- Improves on *parallelBFS* by populating vertex and edge frontier queue in linear operations
- Terminates when there are no *vertex frontiers*.

Algorithm 5 ScanBFS_Host

1: **Input:** V_a, E_a, S ▷ The graph $G(V, E)$ and source S
2: Create updating cost array Deg_a, $PreDeg_a$ of size $|V|$ and initialise all values to 0
3: Create cost array $Dist_a$ of size $|V|$ and initialise all values to ∞
4: Create mask array P_a of size $|V|$ and initialise all values to -1
5: Create two array cQ and nQ, and initialise it to S and *null* respectively.
6: $Dist_a[S] = 0$
7: $P_a[S] = -1$
8: $l = 0$ ▷ Start with the source vertex
9: **while** $cQ_{size} > 0$ **do**
10: Invoke nextLayer($l, V_a, E_a, P_a, Dist_a, cQ$)
11: Invoke countDegrees(V_a, E_a, P_a, cQ, Deg_a)
12: Invoke scanDegrees($cQ_{size}, Deg_a, PreDeg_a$)
13: Perform Prefix Sum on Deg_a, and store the results in $PreDeg_a$
14: $nQ = PreDeg_a[cQ_{size}/NUM\_THREADS]$
15: Invoke populateNextQueue($V_a, E_a, P_a, cQ, nQ, Deg_a, PreDeg_a$)
16: $cQ = nQ$
17: $l = l + 1$

Third Approach: Scan BFS

...

Kernels

- Threads are launched for each node in the vertex frontier
- Uses Blelloch's prefix sum on the number of edge frontiers contributed by a vertex.
- Fills up the the edge frontier using the computed prefix sum.

...

Algorithm 6 nextLayer

```
1: Input: l, V_a, E_a, P_a, Dist_a, cQ
2: tid = getThreadId()                          ▷ Get the Id of the thread
3: if tid < cQ_size then
4:     u = cQ[tid]
5:     for all v = neighbours of u do
6:         if Dist_a[v] > l + 1 then
7:             Dist_a[v] = l + 1
8:             P_a[v] = u
```

Algorithm 7 countDegrees

```
1: Input: V_a, E_a, P_a, cQ, Deg_a
2: tid = getThreadId()                          ▷ Get the Id of the thread
3: if tid < cQ_size then
4:     u = cQ[tid]
5:     d = 0
6:     for all v = neighbours of u do
7:         if P_a[v] = E_a.index(v) and v ≠ u then
8:             d = d + 1
9:     Deg_a[tid] = d
```

Third Approach: Scan BFS

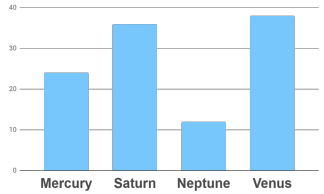
Kernels

Algorithm 9 populateNextQueue

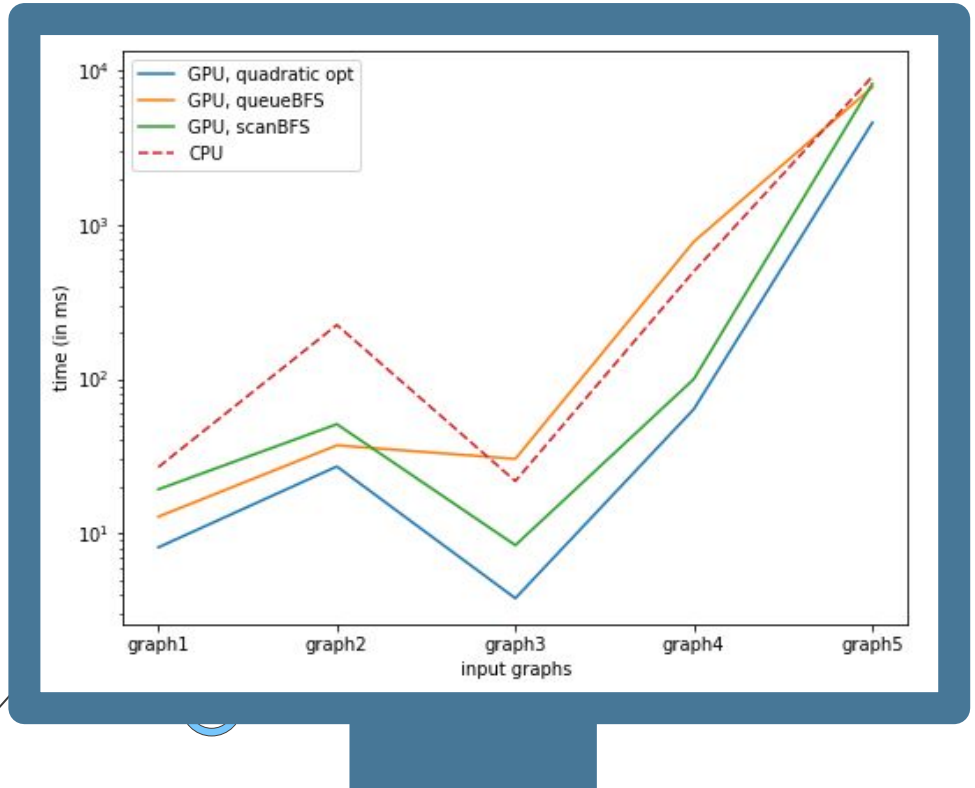
```
1:  Input: V_a, E_a, P_a, cQ, nQ, Deg_a, PreDeg_a
2:  tid = getThreadId()                          ▷ Get the Id of the thread
3:  if tid < cQ_size then
4:      Initialise a shared variable i
5:      if threadId.x = 0 then
6:          i = PreDeg_a[NUM_THREADS]
7:      sync_threads
8:      s = 0
9:      if threadId.x ≠ 0 then
10:         s = Deg_a[tid − 1]
11:     u = cQ[tid]
12:     c = 0
13:     for all v = neighbours of u do
14:         if P_a[v] = E_a.index(v) and v ≠ u then
15:             nQ[i + s + c] = v
16:             c = c + 1
```

Algorithm 8 scanDegrees

```
1:  Input: cQ_size, Deg_a, PreDeg_a
2:  tid = getThreadId()                          ▷ Get the Id of the thread
3:  if tid < cQ_size then
4:      Create a shared array preSum of size NUM_THREADS
5:      m = threadId.x
6:      preSum[m] = Deg_a[tid]
7:      sync_threads
8:      n = 2
9:      while n ≤ NUM_THREADS do
10:         if bitwiseAnd(m, n − 1) = 0 and tid + (2 ∗ n) < cQ_size
    then
11:             preSum[m]+ = preSum[tid + (2 ∗ n)]
12:         sync_threads
13:         n = 2 ∗ n
14:     if m = 0 then
15:         PreDeg_a[tid/NUM_THREADS + 1] = preSum[m]
16:     n = NUM_THREADS
17:     while n > 1 do
18:         if bitwiseAnd(m, n − 1) = 0 and tid + (n/2) < cQ_size
    then
19:             temp = preSum[m]
20:             preSum[m]+ = preSum[tid + (n/2)]
21:             preSum[tid + (n/2)] = temp
22:         sync_threads
23:         n = n/2
24:     Deg_a[tid] = preSum[m]
```



Results and Analysis



Various Observations:

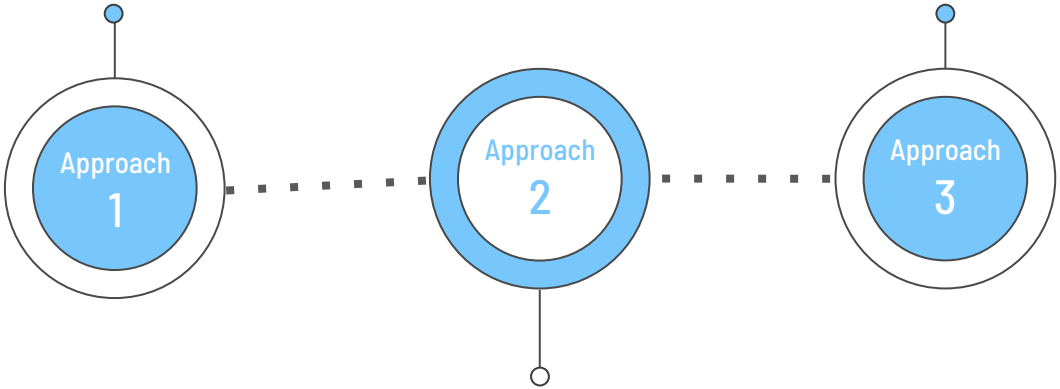
- *parallelBFS* gives the best results for all possible graphs
- *queueBFS* performs well on **dense** graphs
- *scanBFS* performs well on **sparse** graphs

Single Source Shortest Path

Our Approach

Bugged Parallel Dijkstra

Coarsening



Corrected Version

First Approach: Bugged Parallel Dijkstra

- Algorithm proposed by Harish et. al.
- Uses an updating cost array $\mathbf{U_a}$ as intermediate to update the actual cost array $\mathbf{C_a}$. Prevents RAW and WAR data hazards.
- Operates in two sequential phases over multiple iterations.
- The boolean array $\mathbf{M_a}$ and variable ***flag*** determines the termination of the algorithm.

Algorithm 10 SSSP_Host

1: **Input:** V_a, E_a, W_a, S ▷ The graph $G(V, E, W)$ and source S
2: Create updating cost array U_a of size $|V|$ and initalise all values to ∞
3: Create cost array C_a of size $|V|$ and initalise all values to ∞
4: Create mask array M_a of size $|V|$ and initialise all values to $false$
5: $U_a[S] = 0$
6: $C_a[S] = 0$
7: $M_a[S] = flag = true$ ▷ Start with the source vertex
8: **while** $flag$ **do**
9: $flag = false$
10: **for all** $v \in V$ in parallel **do**
11: Invoke SSSP_Phase1($V_a, E_a, W_a, C_a, U_a, M_a$)
12: Invoke SSSP_Phase2($C_a, U_a, M_a, flag$)

First Approach: Bugged Parallel Dijkstra

- In Phase 1, the vertices in $\mathbf{M_a}$ are treated as potential intermediaries for a shortest path.
- The distance to neighbours of such vertices are updated (Line 5 - 7).
- In Phase 2, $\mathbf{C_a}$ is updated using $\mathbf{U_a}$, and corresponding bit is set in $\mathbf{M_a}$.
- If no such update, the algorithm is terminated through the **flag** variable.

...

Algorithm 11 SSSP_Phase1

```
1: Input: V_a, E_a, W_a, C_a, U_a, M_a
2: tid = getThreadId()                          ▷ Get the Id of the thread
3: if M_a[tid] = true then
4:     M_a[tid] = false
5:     for all neighbours nid of tid do ▷ Line 6, 7 must be atomic
6:         if U_a[nid] > C_a[tid] + W_a[nid] then
7:             U_a[nid] = C_a[tid] + W_a[nid]
```

Algorithm 12 SSSP_Phase2

```
1: Input: C_a, U_a, M_a, flag
2: tid = getThreadId()                          ▷ Get the Id of the thread
3: if C_a[tid] > U_a[tid] then
4:     M_a[tid] = flag = true
5:     C_a[tid] = U_a[tid]
```

...

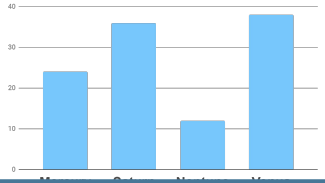
...

Second Approach: Corrected Parallel Dijkstra

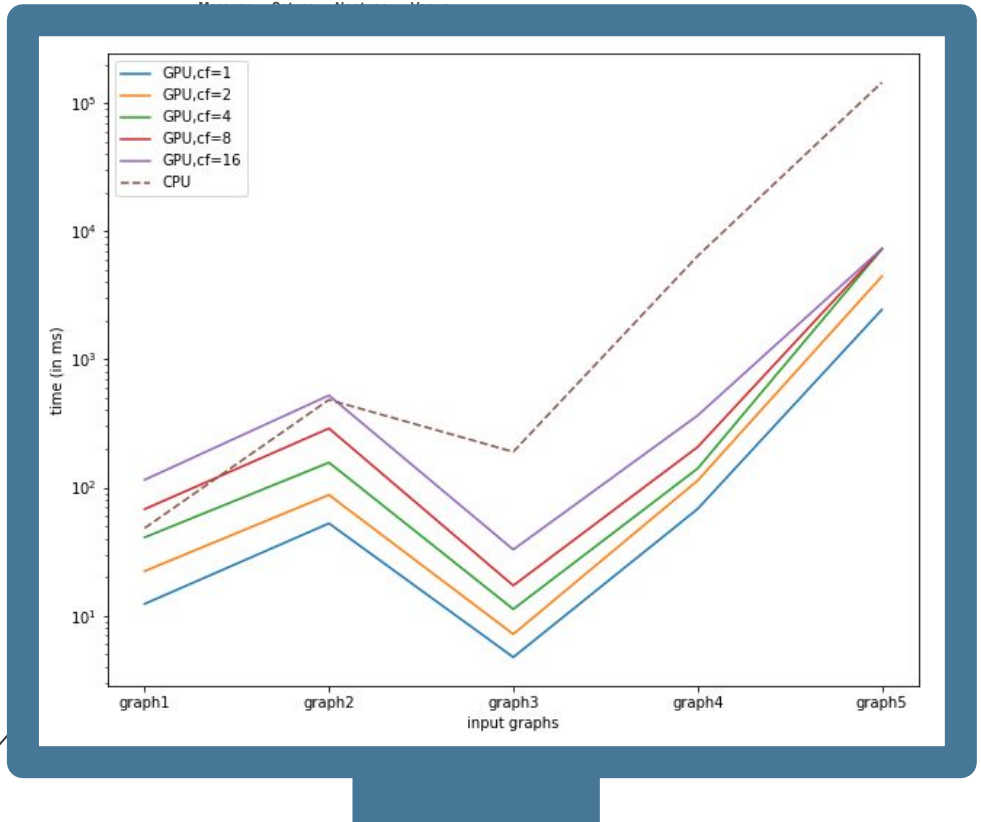
- Above presented algorithm is bugged, pointed out by Martin et. al.
- Using $\mathbf{U_a}$ prevents RAW and WAR in $\mathbf{C_a}$, but does nothing for **WAW** dependencies in $\mathbf{U_a}$ (Line 6 and 7).
- During simultaneous update of $\mathbf{U_a[nid]}$ in Line 7, we need the smallest value to be retained.
- Solve it indirectly, by executing Line 6 and 7 **atomically.**

Algorithm 11 SSSP_Phase1

```
1: Input: V_a, E_a, W_a, C_a, U_a, M_a
2: tid = getThreadId()                          ▷ Get the Id of the thread
3: if M_a[tid] = true then
4:     M_a[tid] = false
5:     for all neighbours nid of tid do ▷ Line 6, 7 must be atomic
6:         if U_a[nid] > C_a[tid] + W_a[nid] then
7:             U_a[nid] = C_a[tid] + W_a[nid]
```



Results and Analysis

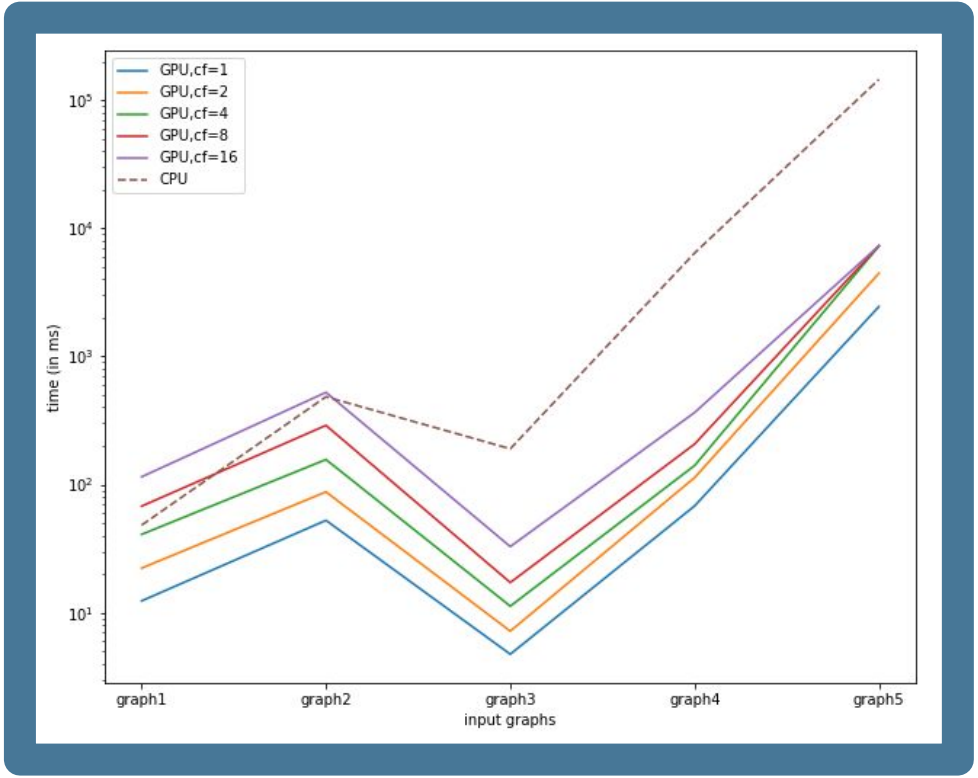


Observations

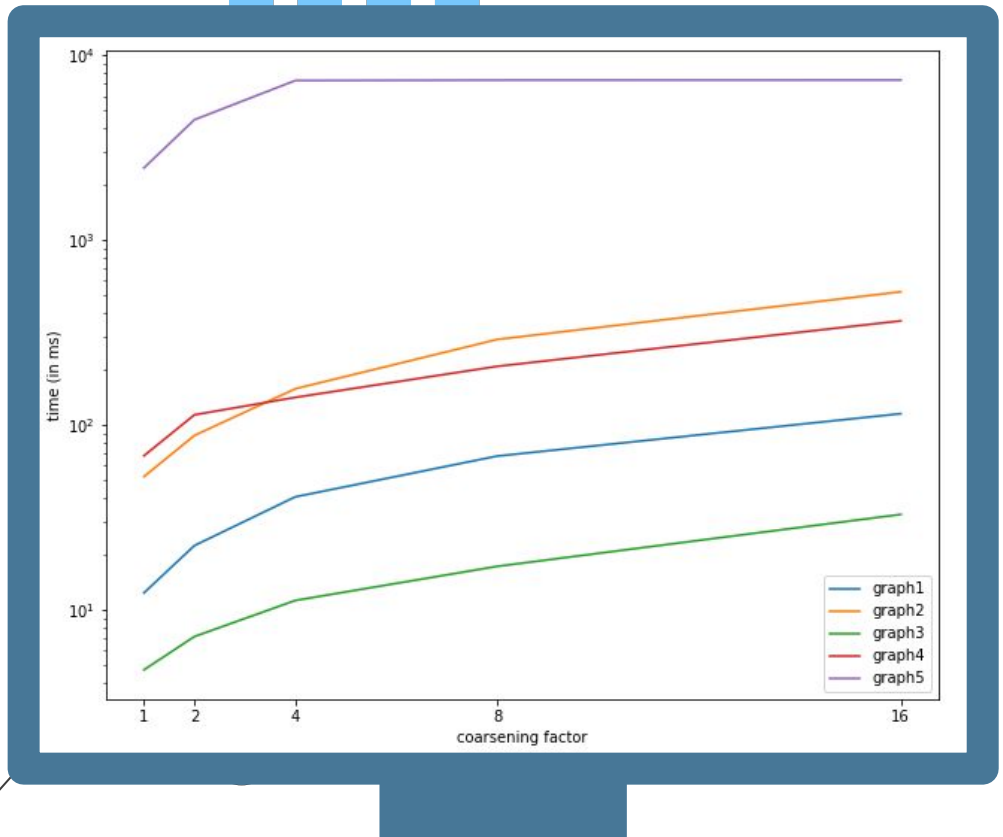
- Heavily dependent on number of **edges** (Graph 2 vs Graph 3 & 4)
- Speedup of GPU compared to CPU is dependent on the **density of graph**.
- Lower the density, higher the speedup offered.
- **Density** [G1 (0.5) >> G5 (10^{-6})], **SpeedUp** [G1 (4x) << G5 (95x)]

Third Approach: Thread Coarsening

- Smaller number of more Coarse-grained threads are being executed
- Instructions executed by a number of different threads are merged into a single thread.
- For finding the optimal thread coarsening factor, we used the manual approach of merging.



Results and Analysis



Various Observations:

- A consistent fall of performance with increase in the thread coarsening factor (**c.f**).
- Reason can be reduction in parallelism with increasing **c.f**
- Another reason can the subsequent increase in pressure on the kernel.

All Pair Shortest Path

First Approach: Using SSSP

What if we call SSSP on all the vertices?

- Works well on sparse graphs.

- Serial time complexity: $O(V^2 \log V + EV)$

...

Algorithm 13 APSP_Using_SSSP

1: **Input:** V_a, E_a, W_a ▷ The graph $G(V, E, W)$
2: Create updating cost array U_a of size $|V|$
3: Create cost array C_a of size $|V|$
4: Create mask array M_a of size $|V|$ and initialise all values to $false$
5: Create a 2d output array O_a of size $|V|$ x $|V|$
6: **for all** $S \in V$ **do**
7: Assign ∞ to all values of U_a and C_a
8: $U_a[S] = C_a[S] = 0$
9: $M_a[S] = flag = true$ ▷ Start with the source vertex
10: **while** $flag$ **do**
11: $flag = false$
12: **for all** $v \in V$ in parallel **do**
13: Invoke SSSP_Phase1($V_a, E_a, W_a, C_a, U_a, M_a$)
14: Invoke SSSP_Phase2($C_a, U_a, M_a, flag$)
15: Copy the distances in C_a to $O_a[S]$

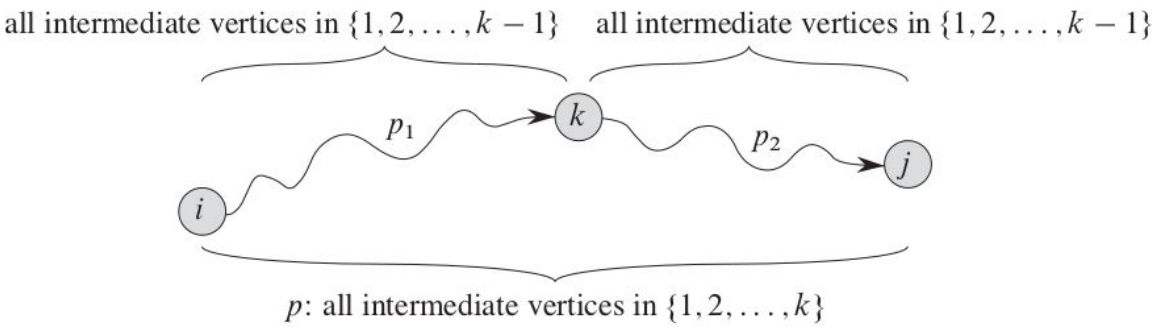
...

...

Second Approach: Floyd Warshall

...

Floyd Warshall in Brief



$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases}$$

...

...

Second Approach: Floyd Warshall

...

Implementing FW in GPU

|V| iterations

One thread per element

Algorithm 14 APSP_Naive_FW

```
1: Input: E        ▷ G(V, E, W) as adjacency matrix of size |V|x|V|
2: Initalise the non-edges in E as ∞
3: for all k ∈ V do
4:     for all (u, v) ∈ V × V in parallel do
5:         E[u, v] = min(E[u, v], E[u, k] + E[k, v])
```

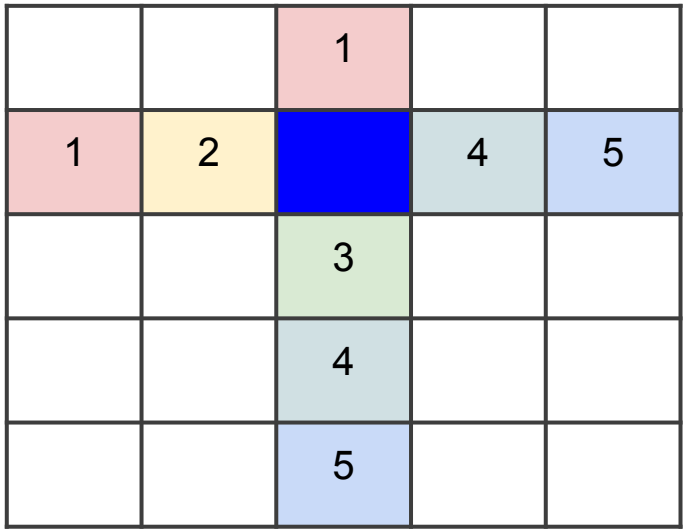
- Uses the adjacency matrix representation rather than the adjacency list representation discussed earlier.
- $O(|V|^2)$ threads and $O(|V|)$ iterations.

... ...

Third Approach: Blocked Floyd Warshall

...

A deeper look into FW



Solution: Use Tiling

...

...

Third Approach: Using Blocked Floyd Warshall

First Phase

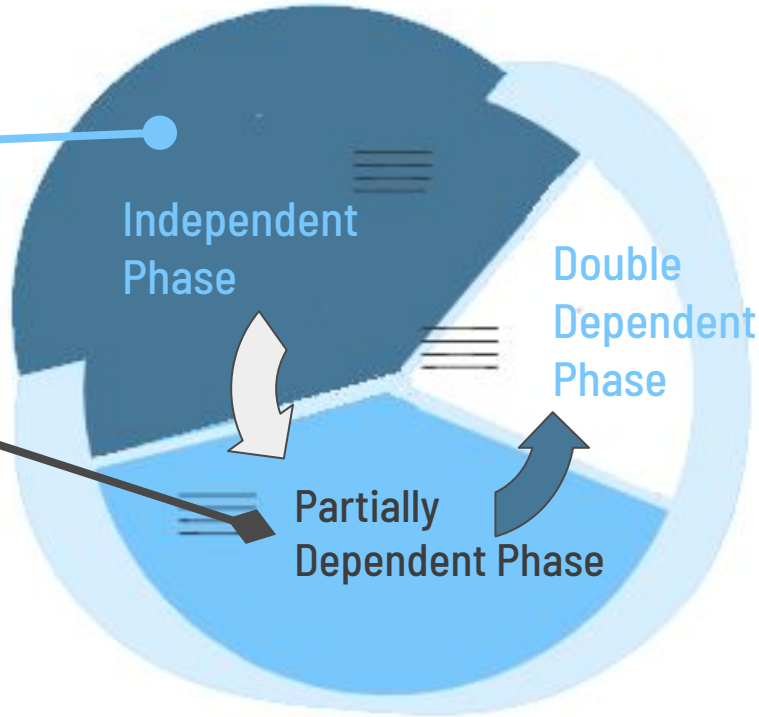
Diagonal Blocks

Second Phase

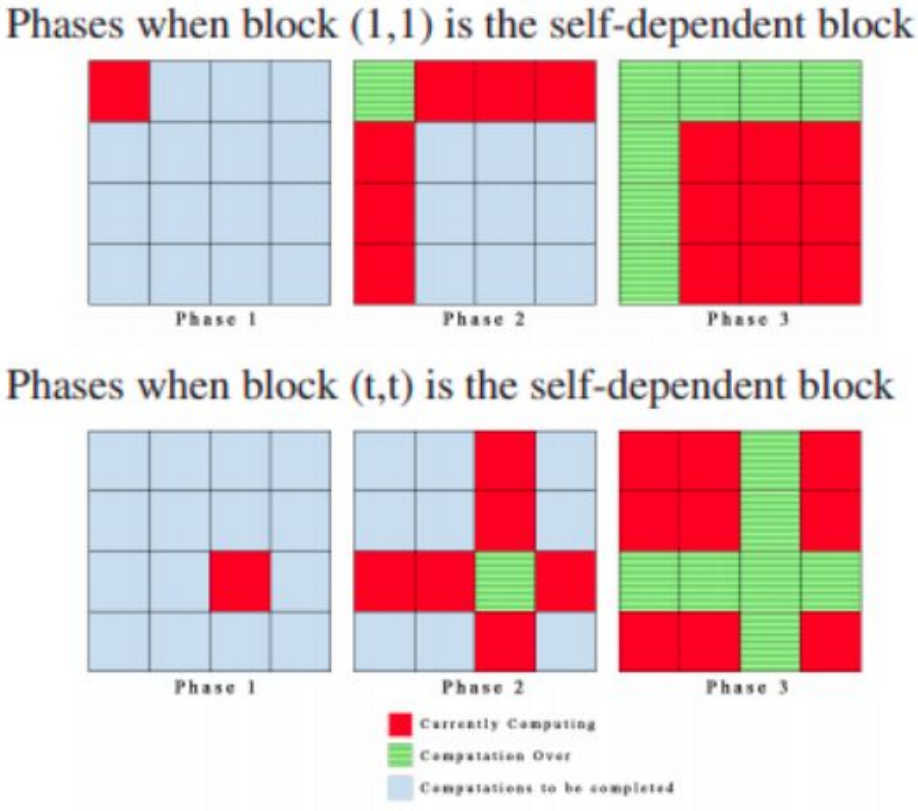
Blocks row and column aligned to diagonal

Third Phase

All other blocks

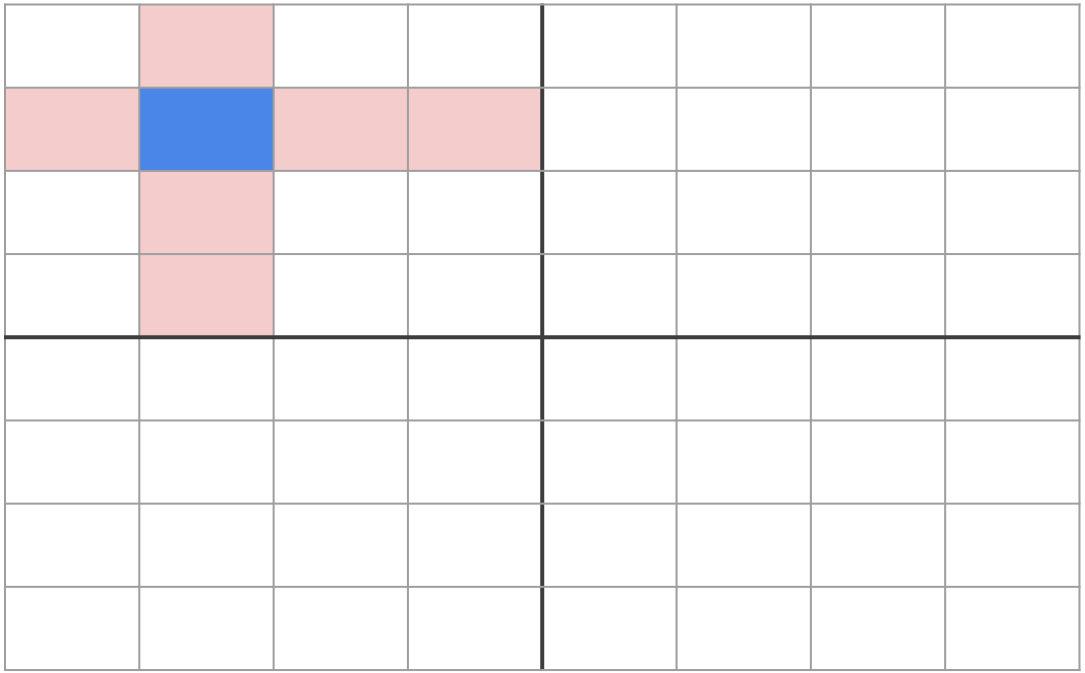


Third Approach: Using Blocked Floyd Warshall



Third Approach: Using Blocked Floyd Warshall

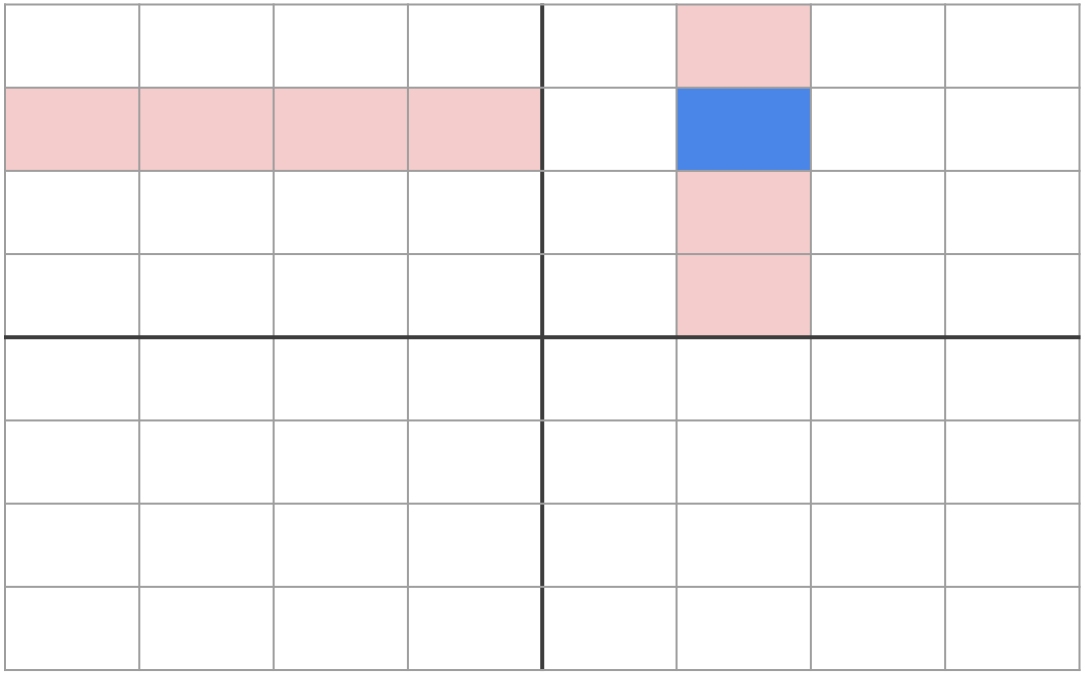
Independent Phase



K = 1 to 4

Third Approach: Using Blocked Floyd Warshall

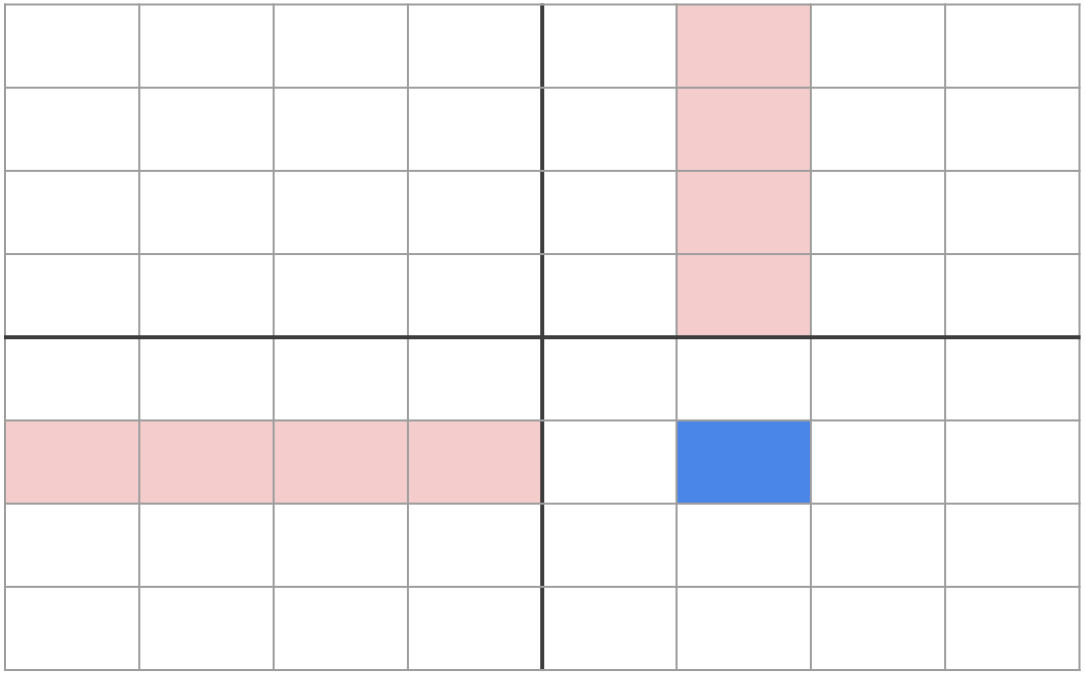
Partially dependent Phase



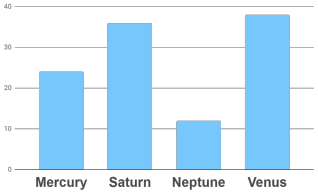
K = 1 to 4

Third Approach: Using Blocked Floyd Warshall

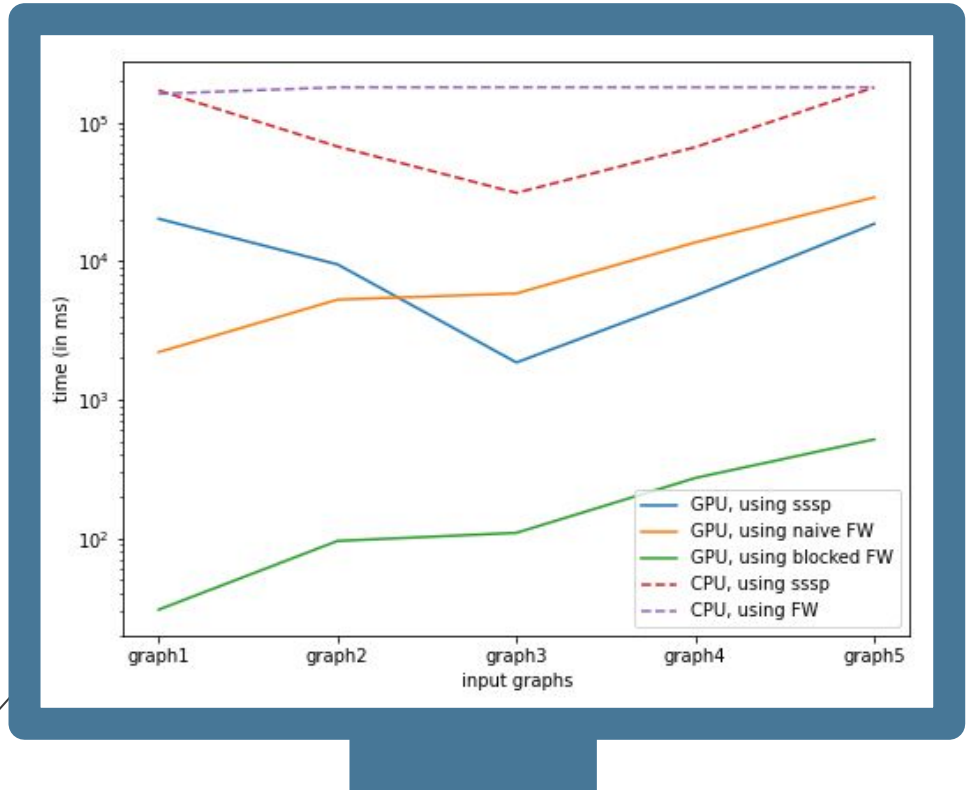
Double dependent Phase



K = 1 to 4



Results and Analysis



Various Observations:

- Naive GPU FW performs better than using SSSP for dense graphs.
- **Blocked FW has a massive improvement in performance.**
- **FW does not depend on sparsity** of the graph.

Thanks!