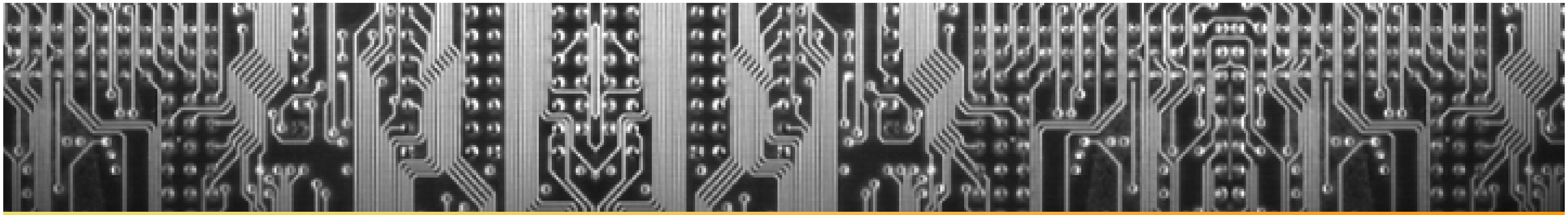


All-Pairs-Shortest-Paths for Large Graphs on the GPU

Gary J Katz[1,2], Joe Kider[1]
[1]University of Pennsylvania
[2]Lockheed Martin IS&GS

What Will We Cover?

- Quick overview of Transitive Closure and All-Pairs Shortest Path
- Uses for Transitive Closure and All-Pairs
- GPUs, What are they and why do we care?
- The GPU problem with performing Transitive Closure and All-Pairs….
- Solution, The Block Processing Method
- Memory formatting in global and shared memory
- Results

Previous Work

- “A Blocked All-Pairs Shortest-Paths Algorithm”

  Venkataraman et al.

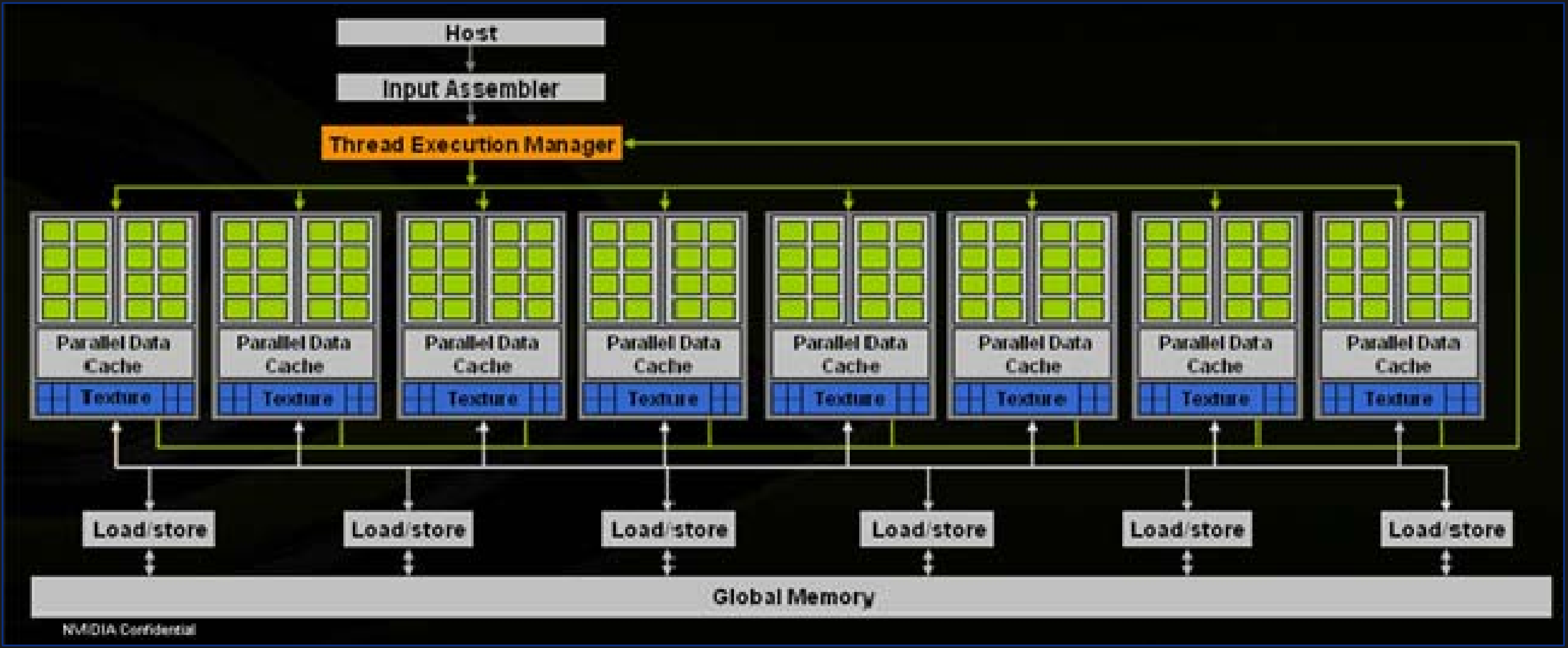
- “Parallel FPGA-based All-Pairs Shortest Path in a Diverted Graph”

  Bondhugula et al.

- “Accelerating large graph algorithms on the GPU using CUDA”

  Harish

NVIDIA GPU Architecture



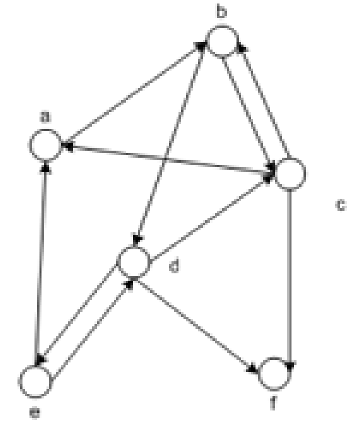
Issues

- No Access to main memory
- Programmer needs to explicitly reference L1 shared cache
- Can not synchronize multiprocessors
- Compute cores are not as smart as CPUs, does not handle if statements well

Background

- Some graph G with vertices V and edges E
- G= (V,E)
- For every pair of vertices u,v in V a shortest path from u to v, where the weight of a path is the sum of he weights of its edges

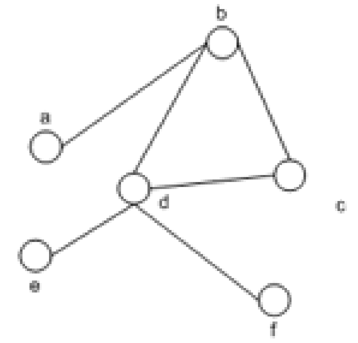
Adjacency Matrix



Directed Graph (a)

	a	b	c	d	e	f
a		✓				
b			✓	✓		
c		✓				✓
d			✓		✓	✓
e	✓			✓		
f						

Adjacency Matrix Representation (a)



Undirected Graph (b)

	a	b	c	d	e	f
a		✓				
b	✓		✓	✓		
c		✓		✓		
d		✓	✓		✓	✓
e				✓		
f				✓		

Adjacency Matrix Representation (b)

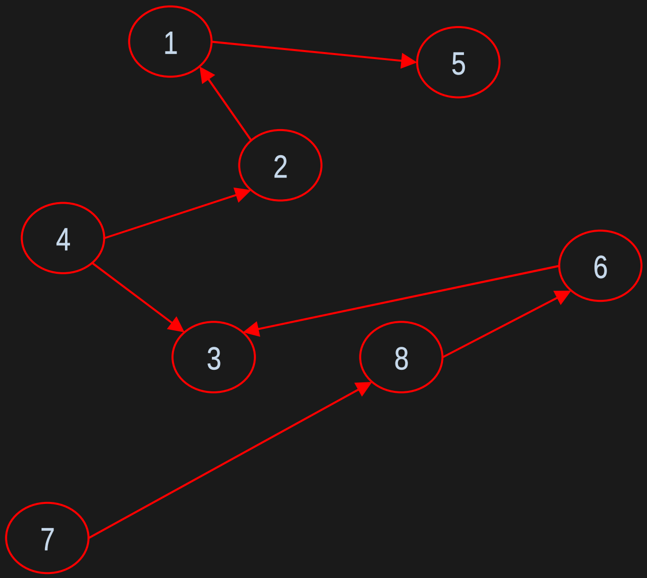
Quick Overview of Transitive Closure

The **Transitive Closure** of G is defined as the graph G* = (V, E*), where
E* = {(i,j) : there is a path from vertex i to vertex j in G}

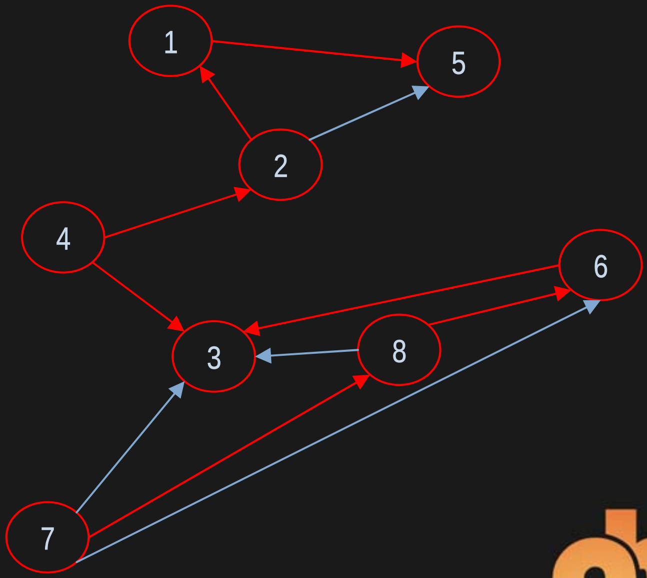
-Introduction to Algorithms, T. Cormen

Simply Stated: The Transitive Closure of a graph is the list of edges for any vertices that can reach each other

Edges
1, 5
2, 1
4, 2
4, 3
6, 3
8, 6



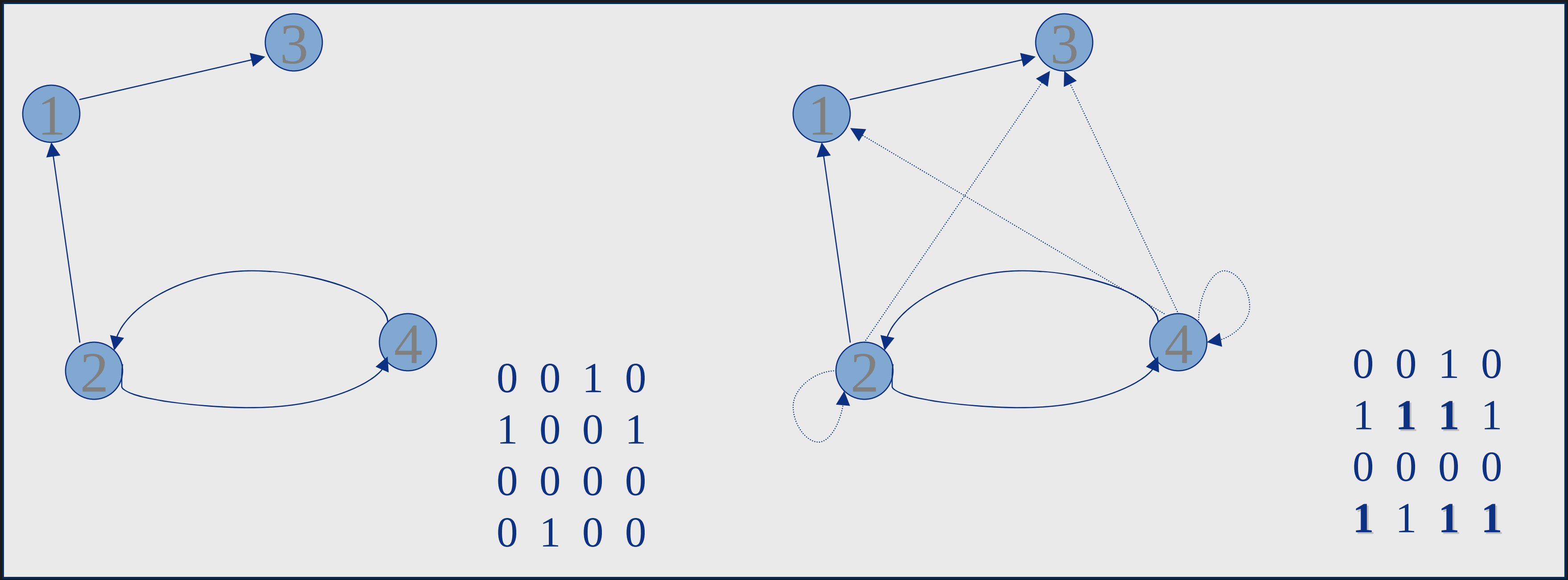
Edges
1, 5
2, 1
4, 2
4, 3
6, 3
8, 6
2, 5
8, 3
7, 6
7, 3



Warshall's algorithm: transitive closure

- Computes the transitive closure of a relation
- (Alternatively: all paths in a directed graph)

- Example of transitive closure:1



$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

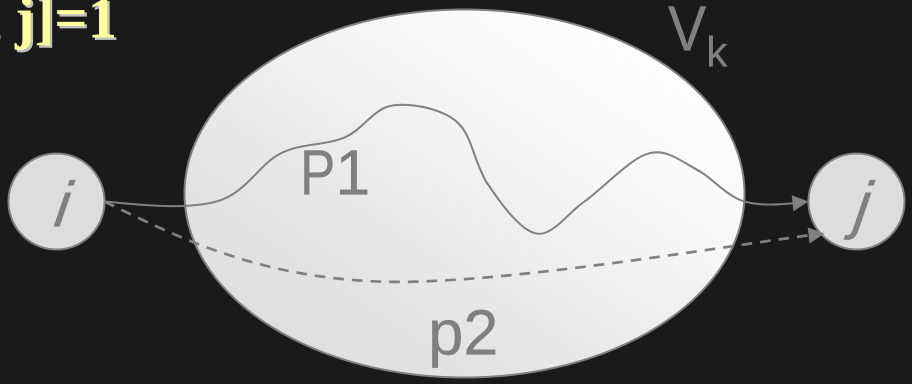
$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

Warshall's algorithm

- **Main idea: a path exists between two vertices i, j, iff**
  - **there is an edge from i to j; or**
  - **there is a path from i to j going through vertex 1; or**
  - **there is a path from i to j going through vertex 1 and/or 2; or**
  - **…**
  - **there is a path from i to j going through vertex 1, 2, … and/or k; or**
  - **...**
  - **there is a path from i to j going through any of the other vertices**

Warshall’s algorithm

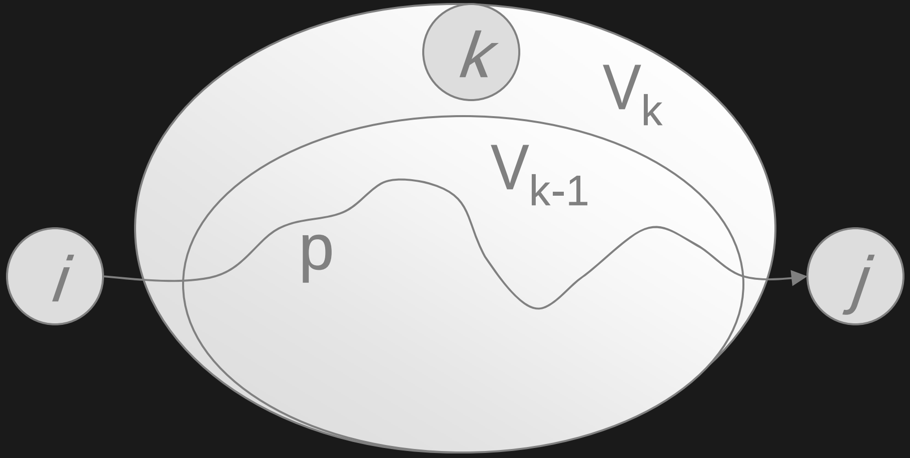
- **Idea: dynamic programming**
  - **Let V={1, …, n} and for $k \leqslant n$, V_k={1, …, k}**
  - **For any pair of vertices $i, j \in V$, identify all paths from i to j whose intermediate vertices are all drawn from V_k: P_{ij}^k={p1, p2, …}, if $P_{ij}^k \neq \emptyset$ then $R^k[i, j]=1$**



  - **For any pair of vertices i, j: $R^n[i, j]$, that is R^n**
  - **Starting with $R^0=A$, the adjacency matrix, how to get $R^1 \Rightarrow \dots \Rightarrow R^{k-1} \Rightarrow R^k \Rightarrow \dots \Rightarrow R^n$**

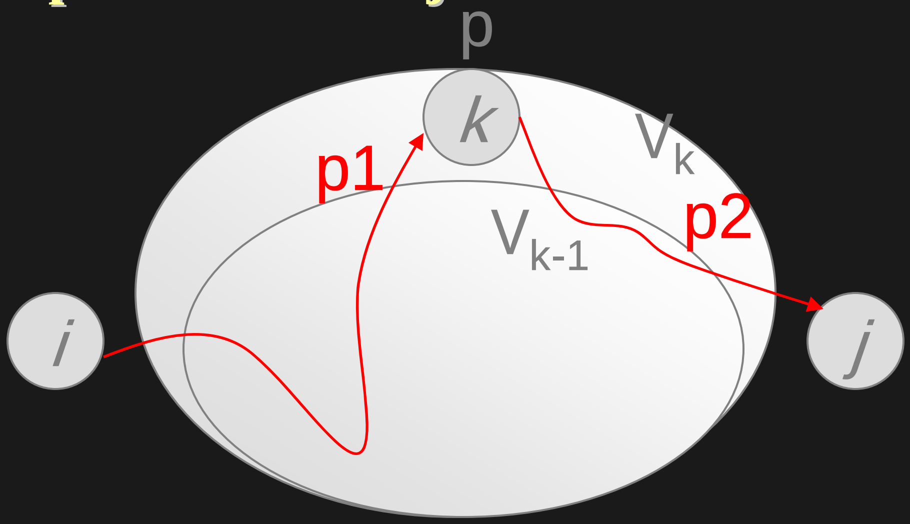
Warshall's algorithm

- **Idea: dynamic programming**
  - **$p \in P_{ij}^{k}$: p is a path from i to j with all intermediate vertices in V_k**
  - **If k is not on p, then p is also a path from i to j with all intermediate vertices in V_{k-1}: $p \in P_{ij}^{k-1}$**



Warshall's algorithm

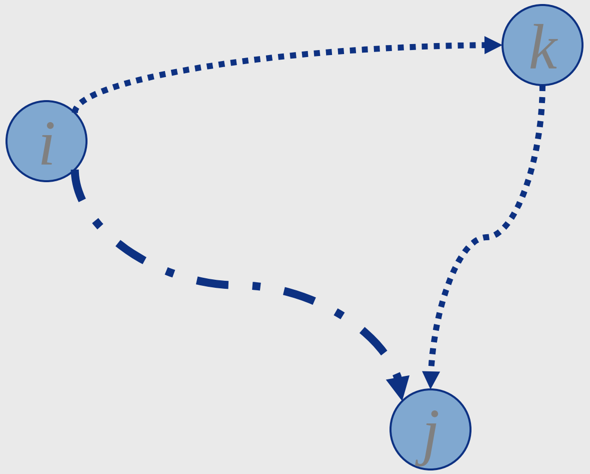
- **Idea: dynamic programming**
  - **$p \in P_{ij}^{k}$: p is a path from i to j with all intermediate vertices in V_k**
  - **If k is on p, then we break down p into p_1 and p_2 where**
    - **p_1 is a path from i to k with all intermediate vertices in V_{k-1}**
    - **p_2 is a path from k to j with all intermediate vertices in V_{k-1}**



Warshall's algorithm

- **In the k^{th} stage determine if a path exists between two vertices i, j using just vertices among 1, …, k**

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1, \ldots, k\text{-1)} \\ \text{or} & \\ (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \\ & \text{and from } k \text{ to } j \\ & \text{using just } 1, \ldots, k\text{-1)} \end{cases}$$



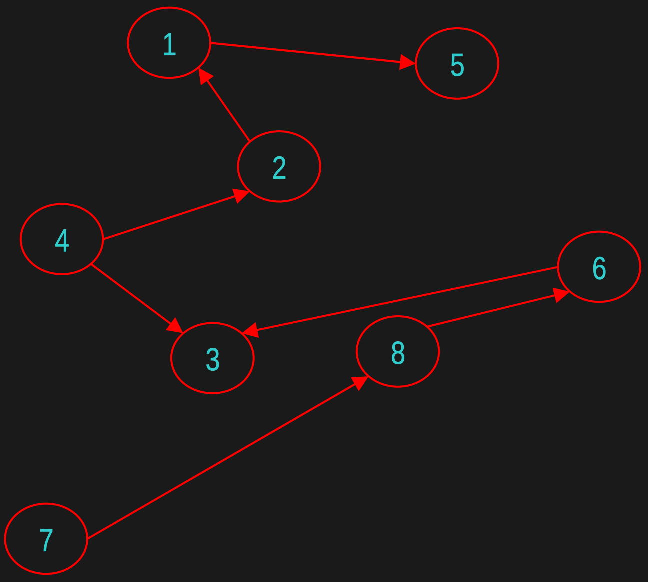
k^{th} stage

Quick Overview All-Pairs-Shortest-Path

The **All-Pairs Shortest-Path** of G is defined for every pair of vertices u,v E V as the shortest (least weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges.

-Introduction to Algorithms, T. Cormen

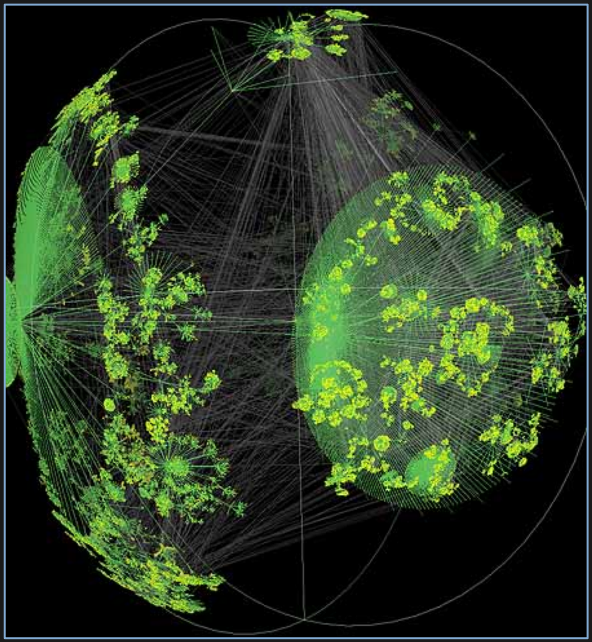
Simply Stated: The All-Pairs-Shortest-Path of a graph is the most optimal list of vertices connecting any two vertices that can reach each other



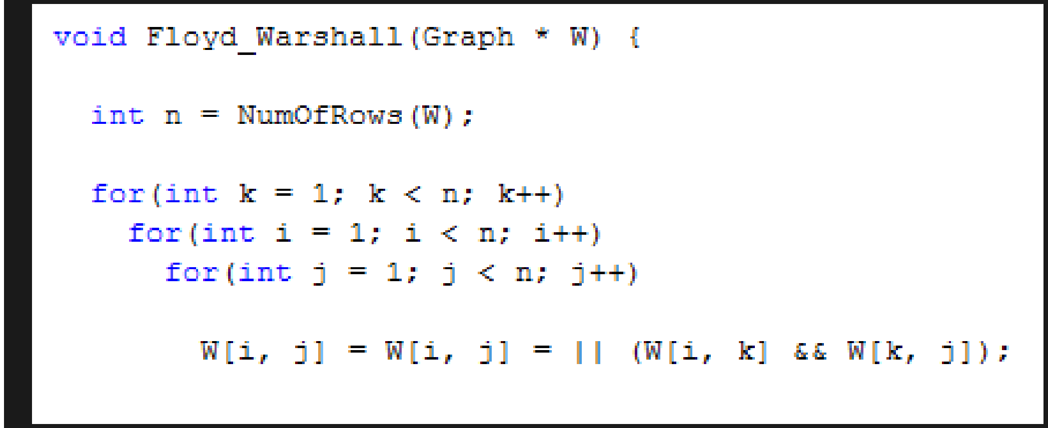
Paths

1 → 5
2 → 1
4 → 2
4 → 3
6 → 3
8 → 6
2 → 1 → 5
8 → 6 → 3
7 → 8 → 6
7 → 8 → 6 → 3

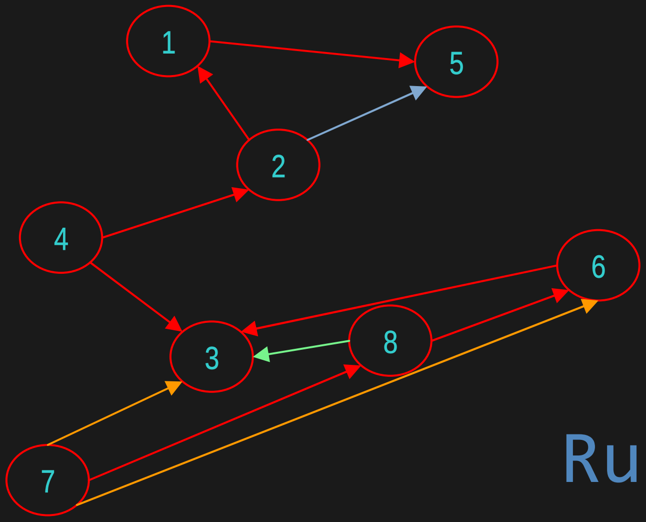
Uses for Transitive Closure and All-Pairs



Floyd-Warshall Algorithm



```
void Floyd_Warshall(Graph * W) {

  int n = NumOfRows(W);

  for(int k = 1; k < n; k++)
    for(int i = 1; i < n; i++)
      for(int j = 1; j < n; j++)

        W[i, j] = W[i, j] = || (W[i, k] && W[k, j]);
```



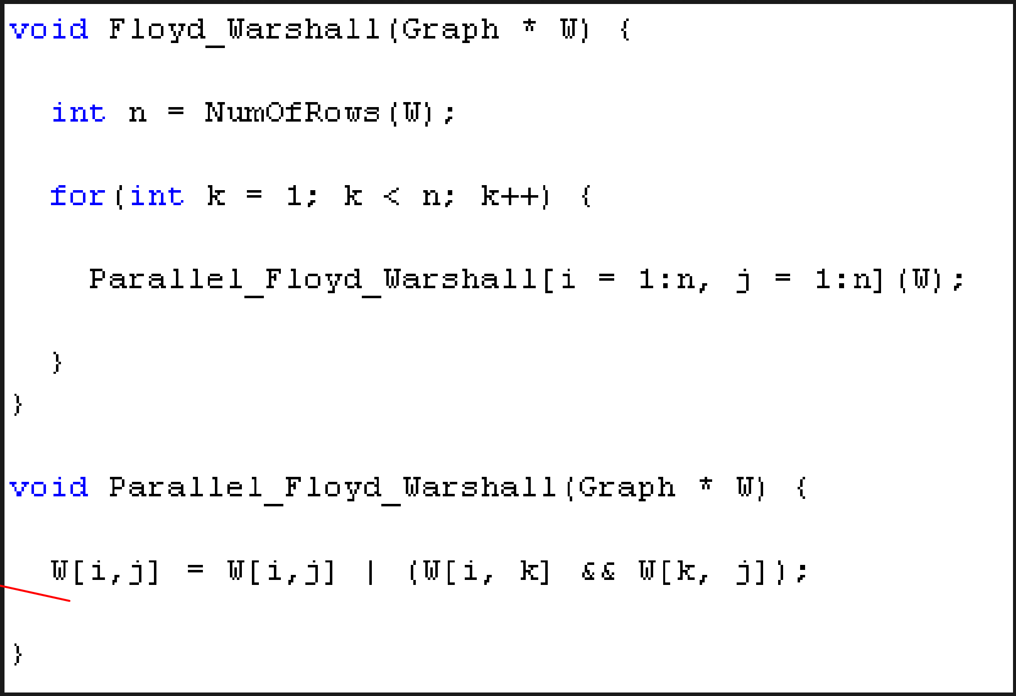
	1	2	3	4	5	6	7	8
1	1	1						
2		1		1				
3			1	1		1	1	1
4				1				
5	1	1			1			
6						1	1	1
7							1	
8							1	1

Pass 8: Finds all connections that are connected through 8

Running Time = $O(V^3)$

Parallel Floyd-Warshall

Each Processing Element needs global access to memory



```
void Floyd_Warshall(Graph * W) {

  int n = NumOfRows(W);

  for(int k = 1; k < n; k++) {

    Parallel_Floyd_Warshall[i = 1:n, j = 1:n](W);

  }
}

void Parallel_Floyd_Warshall(Graph * W) {

  W[i,j] = W[i,j] | (W[i, k] && W[k, j]);

}
```

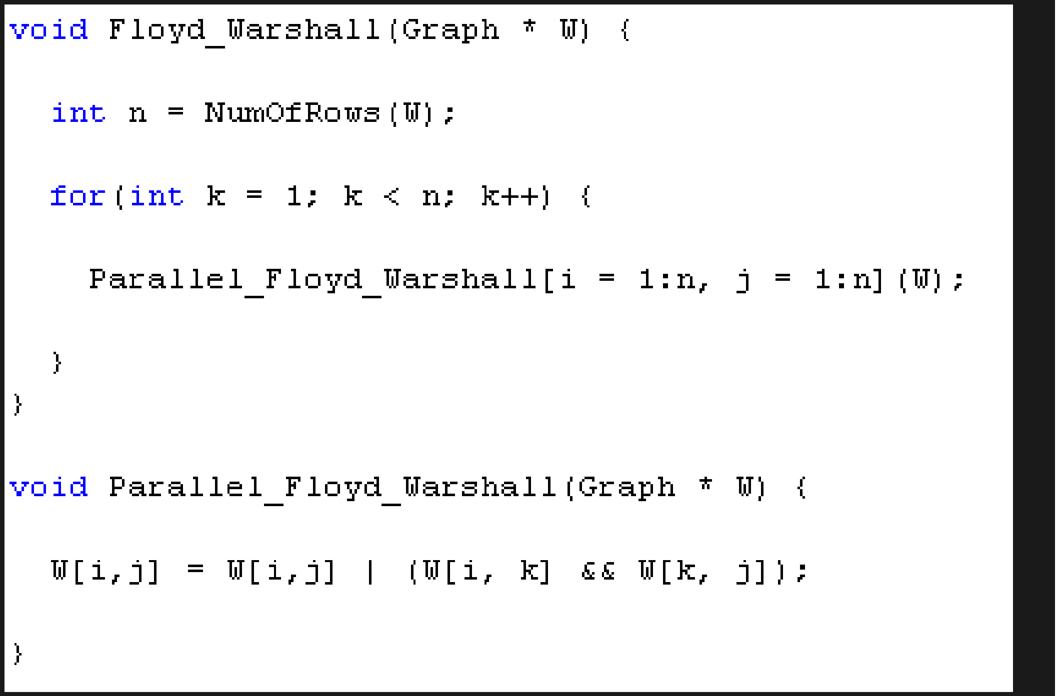
This can be an issue for GPUs

There’s a short coming to this algorithm though…

The Question

How do we calculate the transitive closure on the GPU to:

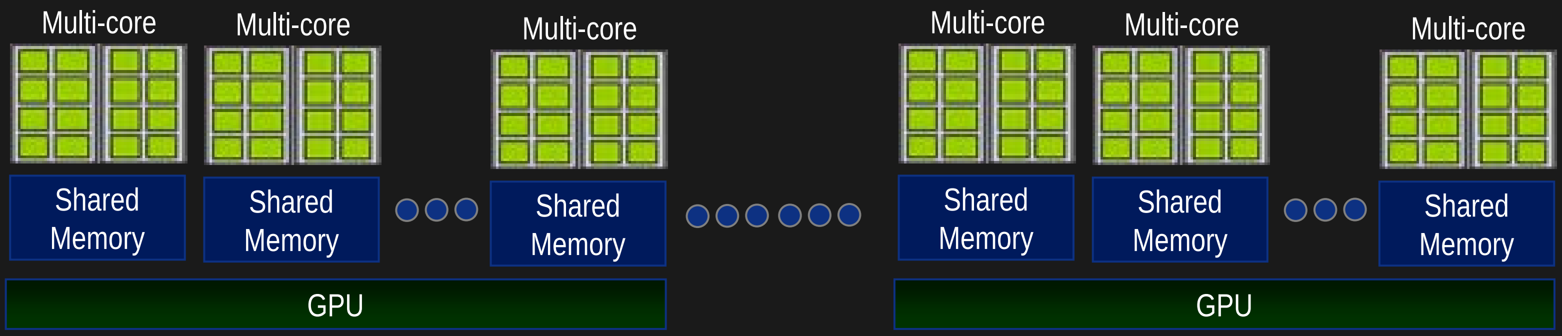
1. Take advantage of shared memory
2. Accommodate data sizes that do not fit in memory

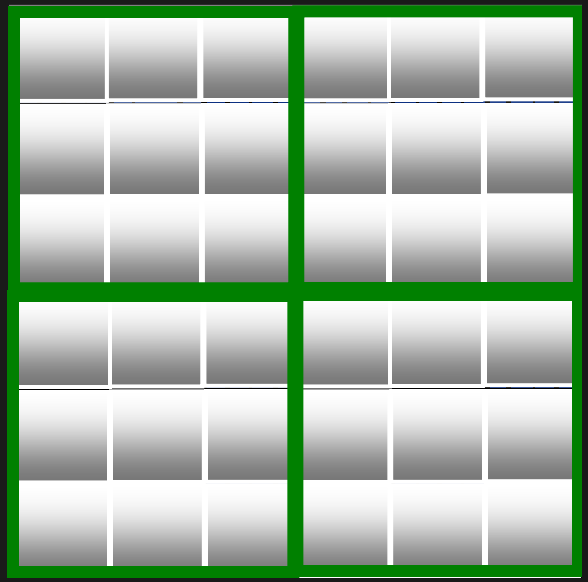


```
void Floyd_Warshall(Graph * W) {
  int n = NumOfRows(W);
  for(int k = 1; k < n; k++) {
    Parallel_Floyd_Warshall[i = 1:n, j = 1:n](W);
  }
}

void Parallel_Floyd_Warshall(Graph * W) {
  W[i,j] = W[i,j] | (W[i, k] && W[k, j]);
}
```

Can we perform partial processing of the data?

Block Processing of Floyd-Warshall

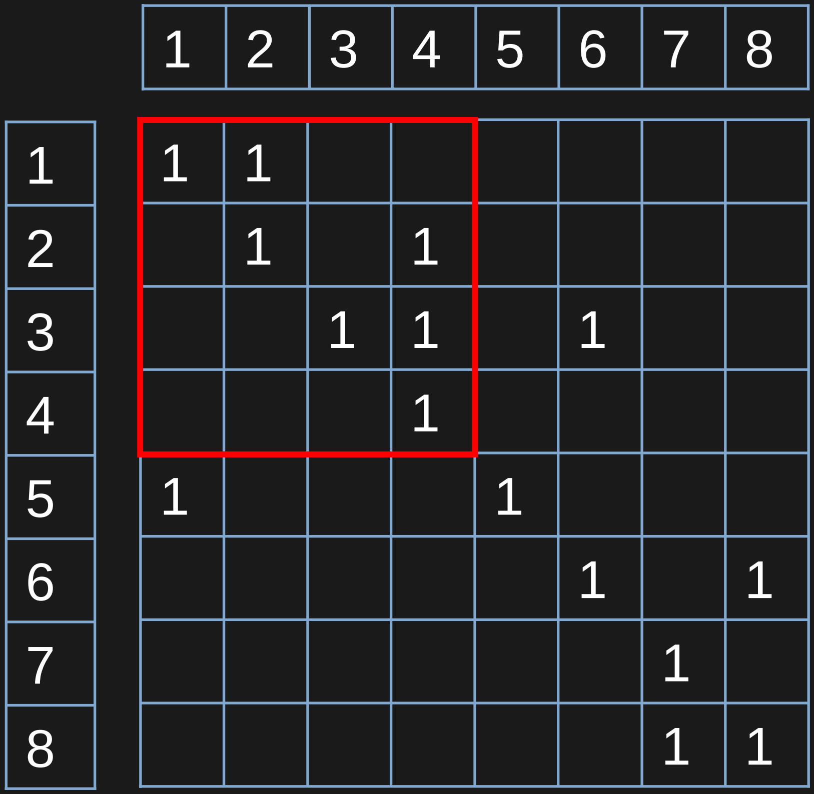




Data Matrix

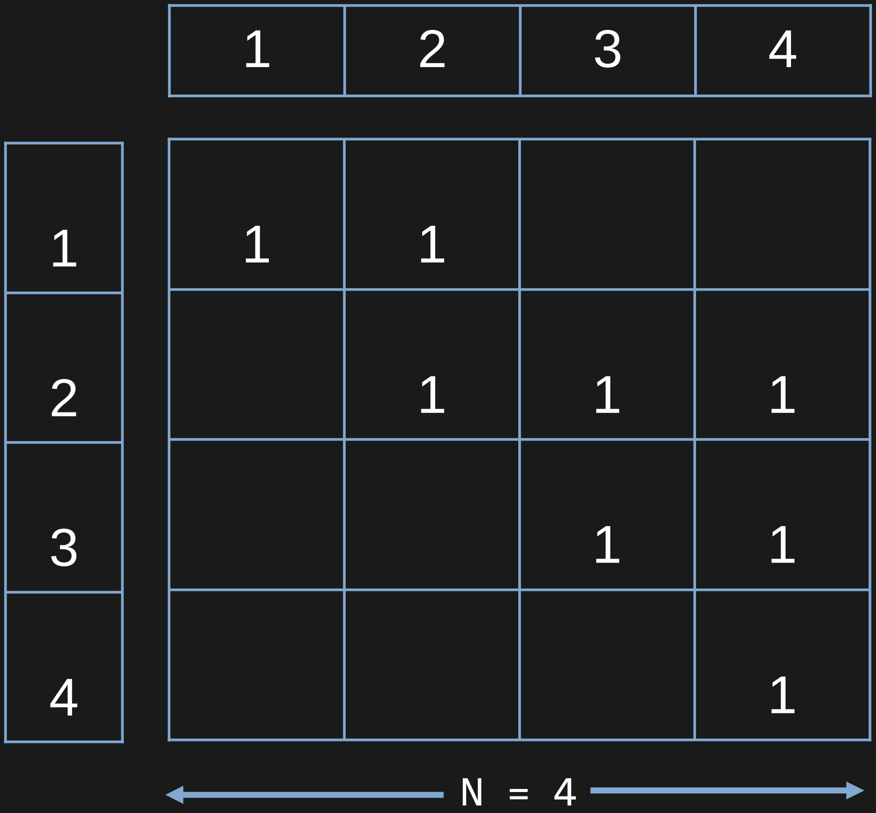
Organizational structure for block processing?

Block Processing of Floyd-Warshall



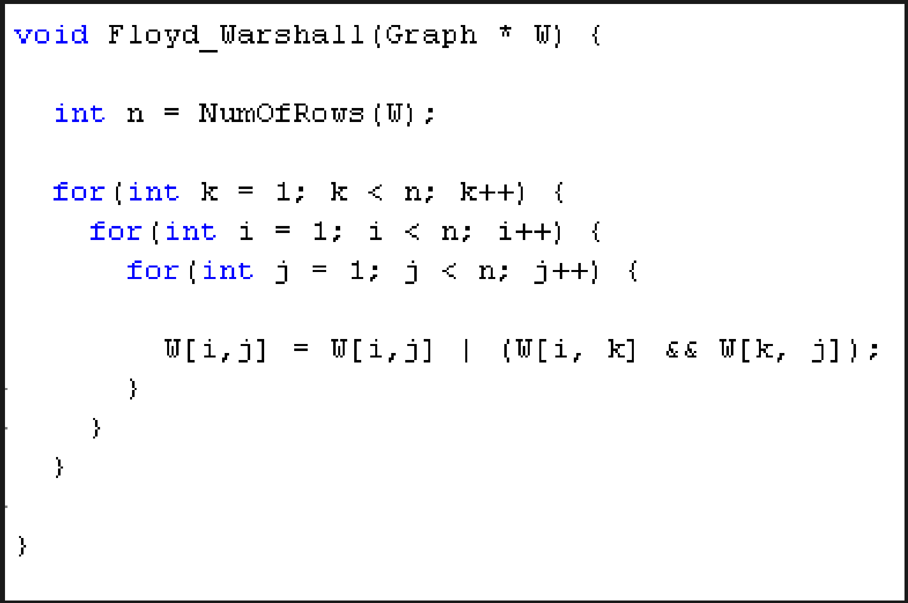
	1	2	3	4	5	6	7	8
1	1	1						
2		1		1				
3			1	1		1		
4				1				
5	1				1			
6						1		1
7							1	
8							1	1

Block Processing of Floyd-Warshall



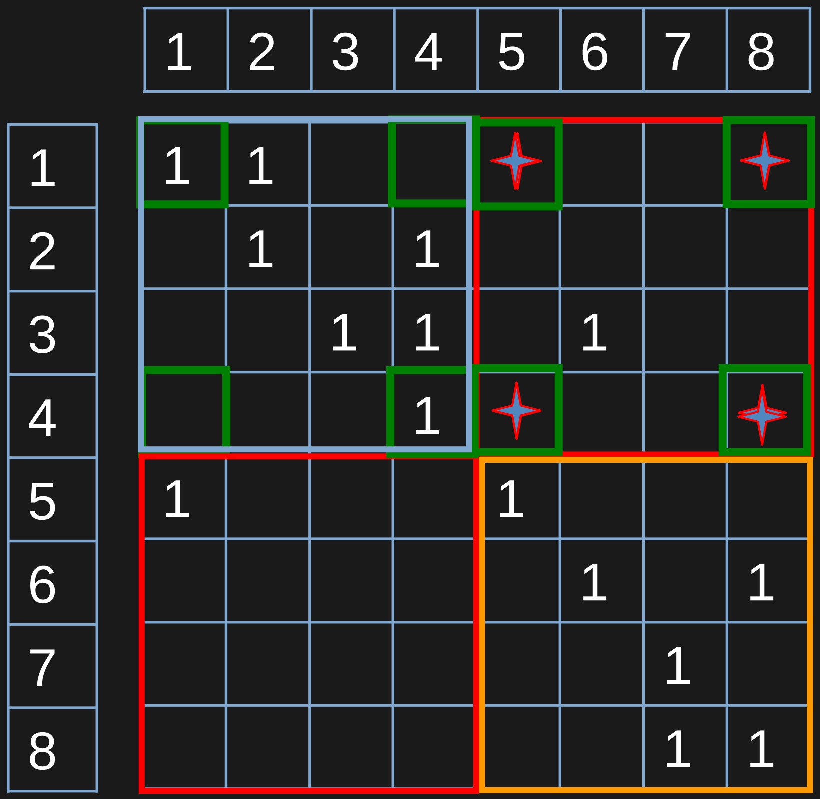
	1	2	3	4
1	1	1		
2		1	1	1
3			1	1
4				1

N = 4



```
void Floyd_Warshall(Graph * W) {

  int n = NumOfRows(W);

  for(int k = 1; k < n; k++) {
    for(int i = 1; i < n; i++) {
      for(int j = 1; j < n; j++) {

        W[i,j] = W[i,j] | (W[i, k] && W[k, j]);
      }
    }
  }

}
```

Block Processing of Floyd-Warshall



	1	2	3	4	5	6	7	8
1	1	1			✦			✦
2		1		1				
3			1	1		1		
4				1	✦			✦
5	1				1			
6						1		1
7							1	
8							1	1

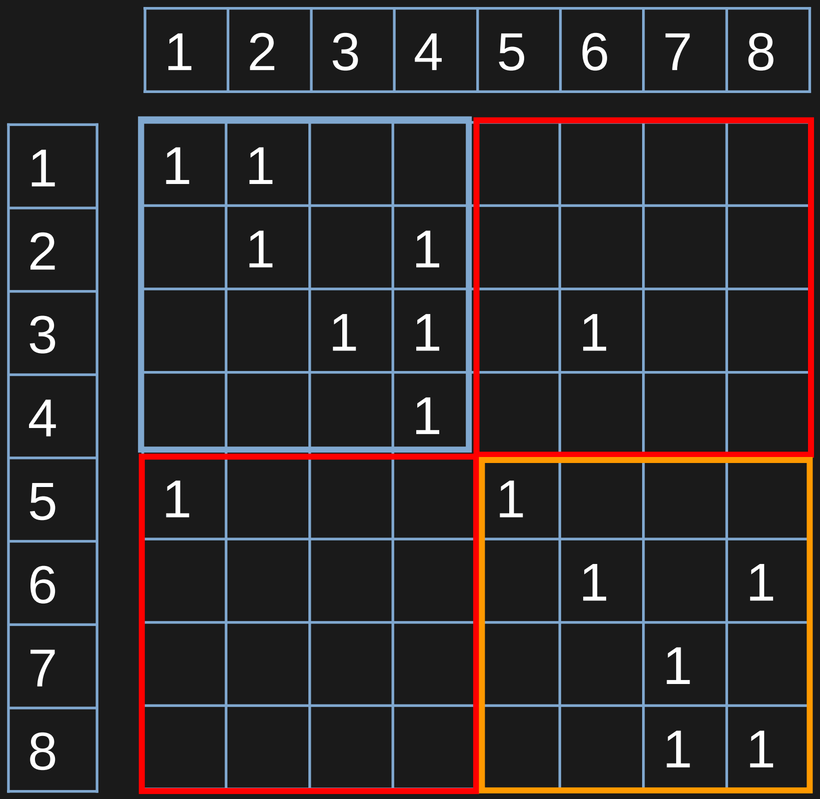
```
K = 1

   [i,j]     [i,k]    [k,j]
   (5,1) -> (5,1) & (1,1)
   (8,1) -> (8,1) & (1,1)
   (5,4) -> (5,1) & (1,4)
   (8,4) -> (8,1) & (1,4)
```

```
K = 4

   [i,j]     [i,k]    [k,j]
   (5,1) -> (5,4) & (4,1)
   (8,1) -> (8,4) & (4,1)
   (5,4) -> (5,4) & (4,4)
   (8,4) -> (8,4) & (4,4)
```

```
W[i,j] = W[i,j] | (W[i,k] && W[k,j])
```

For each pass, k, the cells retrieved must be processed to at least k-1

Block Processing of Floyd-Warshall



	1	2	3	4	5	6	7	8
1	1	1						
2		1		1				
3			1	1		1		
4				1				
5	1				1			
6						1		1
7							1	
8							1	1

Putting it all Together
Processing K = [1-4]

Pass 1:
i = [1-4], j = [1-4]

Pass 2:
i = [5-8], j = [1-4]
i = [1-4], j = [5-8]

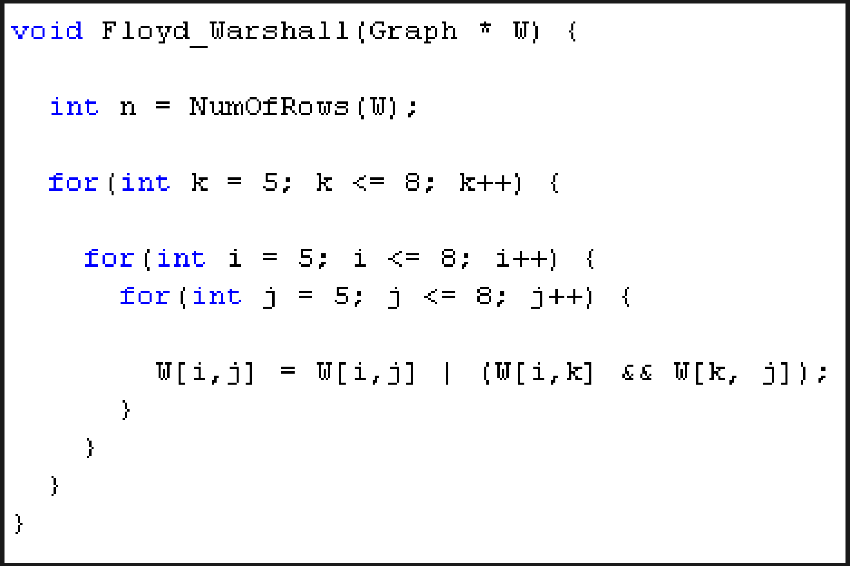
Pass 3:
i = [5-8], j = [5-8]

`W[i,j] = W[i,j] | (W[i,k] && W[k,j])`

Block Processing of Floyd-Warshall

	5	6	7	8
5	1	1		
6		1		1
7			1	
8			1	1

N = 8



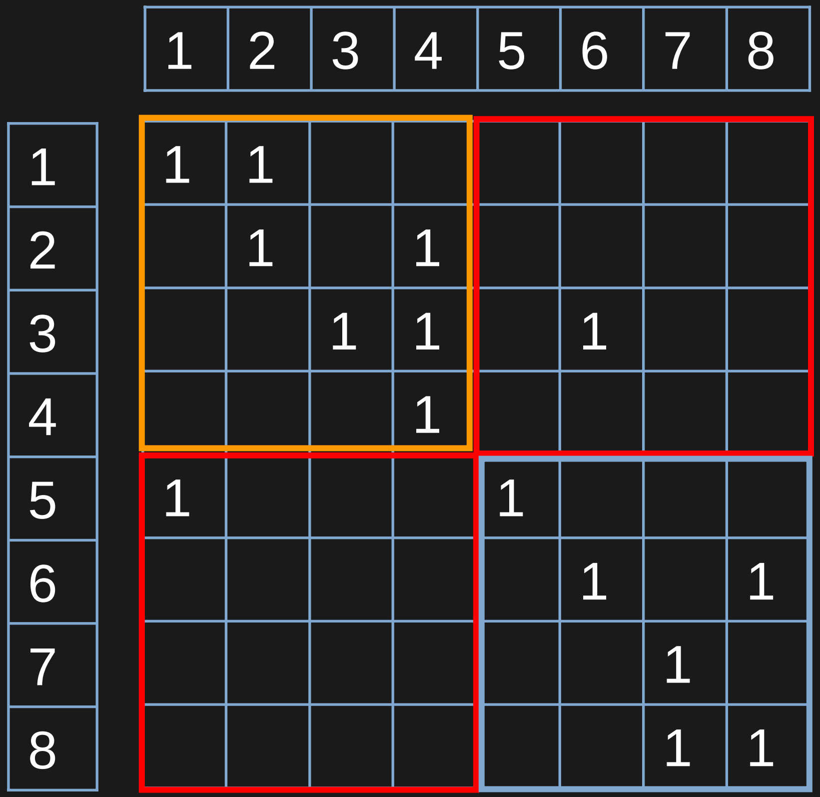
```
void Floyd_Warshall(Graph * W) {

  int n = NumOfRows(W);

  for(int k = 5; k <= 8; k++) {

    for(int i = 5; i <= 8; i++) {
      for(int j = 5; j <= 8; j++) {

        W[i,j] = W[i,j] | (W[i,k] && W[k, j]);
      }
    }
  }
}
```

Range:

i = [5,8]
j = [5,8]
k = [5,8]

Computing k = [5-8]

Block Processing of Floyd-Warshall



	1	2	3	4	5	6	7	8
1	1	1						
2		1		1				
3			1	1		1		
4				1				
5	1				1			
6						1		1
7							1	
8							1	1

Putting it all Together
Processing K = [5-8]

Pass 1:
i = [5-8], j = [5-8]

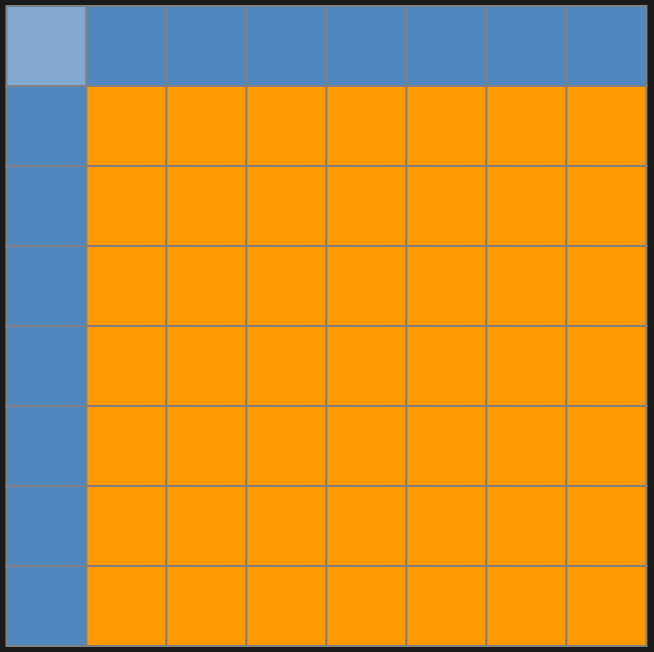
Pass 2:
i = [5-8], j = [1-4]
i = [1-4], j = [5-8]

Pass 3:
i = [1-4], j = [1-4]

Transitive Closure
Is complete for k = [1-8]

```
W[i,j] = W[i,j] | (W[i,k] && W[k,j])
```

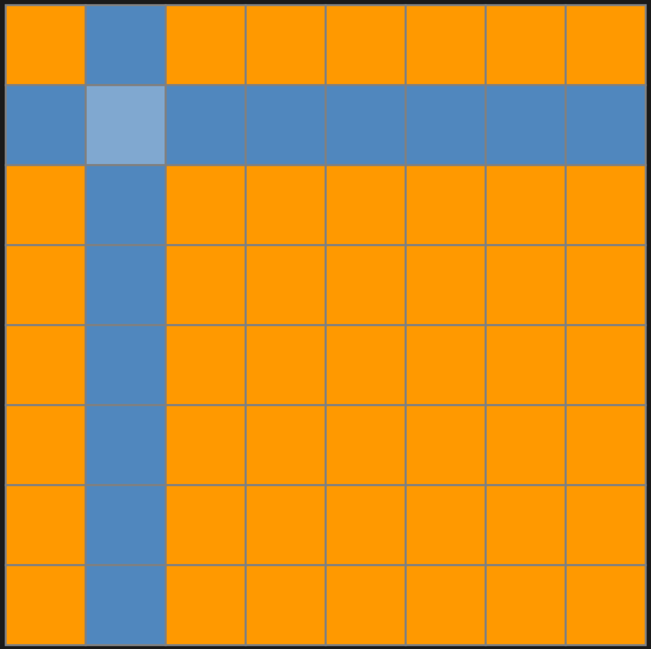
Increasing the Number of Blocks



Pass 1

- Primary blocks are along the diagonal
- Secondary blocks are the rows and columns of the primary block
- Tertiary blocks are all remaining blocks

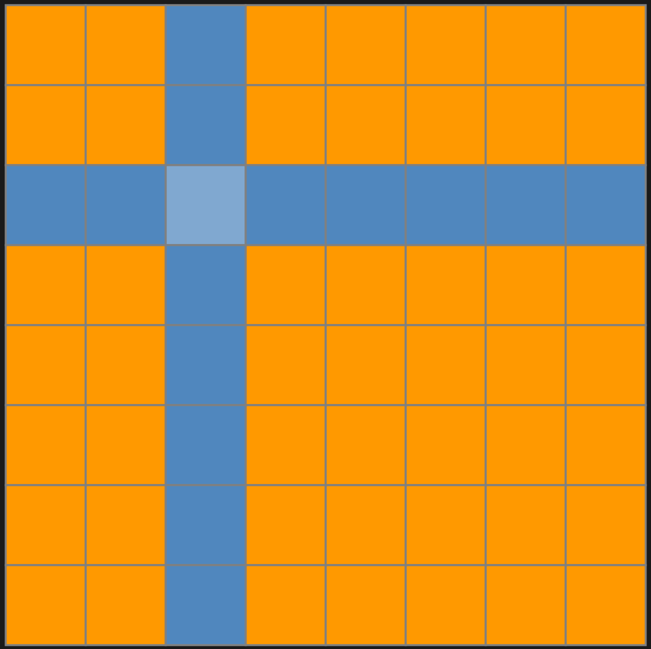
Increasing the Number of Blocks



- Primary blocks are along the diagonal
- Secondary blocks are the rows and columns of the primary block
- Tertiary blocks are all remaining blocks

Pass 2

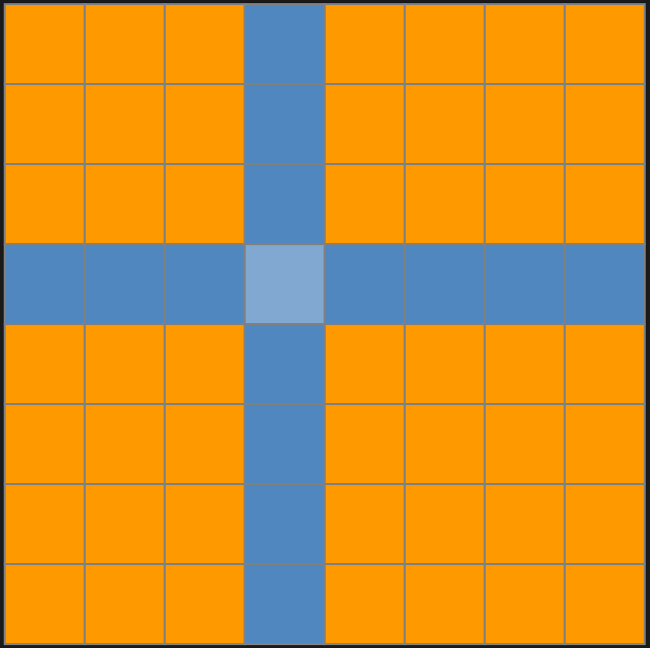
Increasing the Number of Blocks



Pass 3

- Primary blocks are along the diagonal
- Secondary blocks are the rows and columns of the primary block
- Tertiary blocks are all remaining blocks

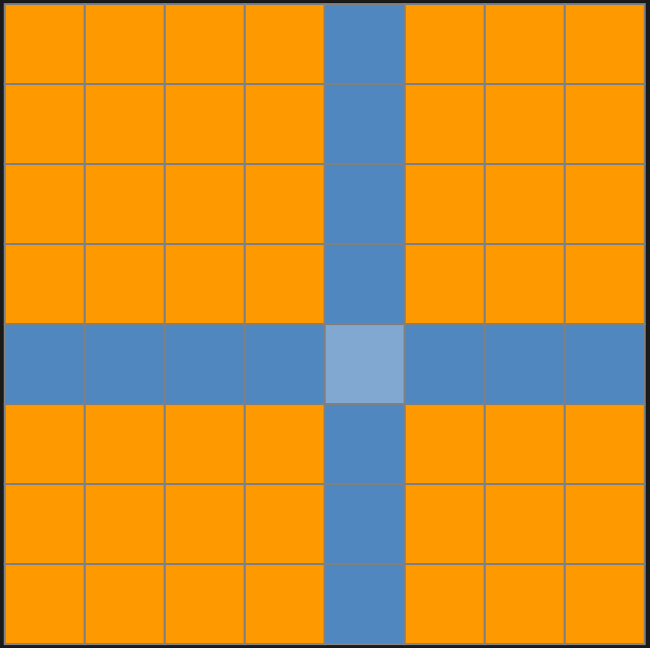
Increasing the Number of Blocks



Pass 4

- Primary blocks are along the diagonal
- Secondary blocks are the rows and columns of the primary block
- Tertiary blocks are all remaining blocks

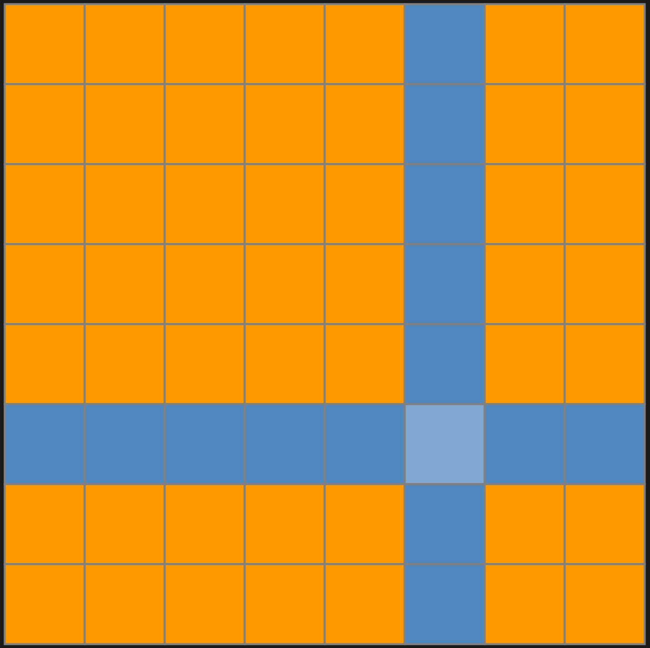
Increasing the Number of Blocks



Pass 5

- Primary blocks are along the diagonal
- Secondary blocks are the rows and columns of the primary block
- Tertiary blocks are all remaining blocks

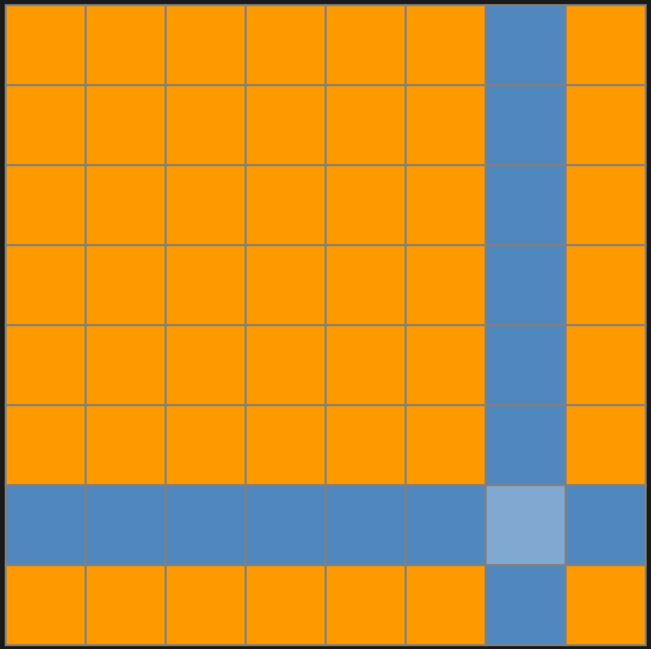
Increasing the Number of Blocks



Pass 6

- Primary blocks are along the diagonal
- Secondary blocks are the rows and columns of the primary block
- Tertiary blocks are all remaining blocks

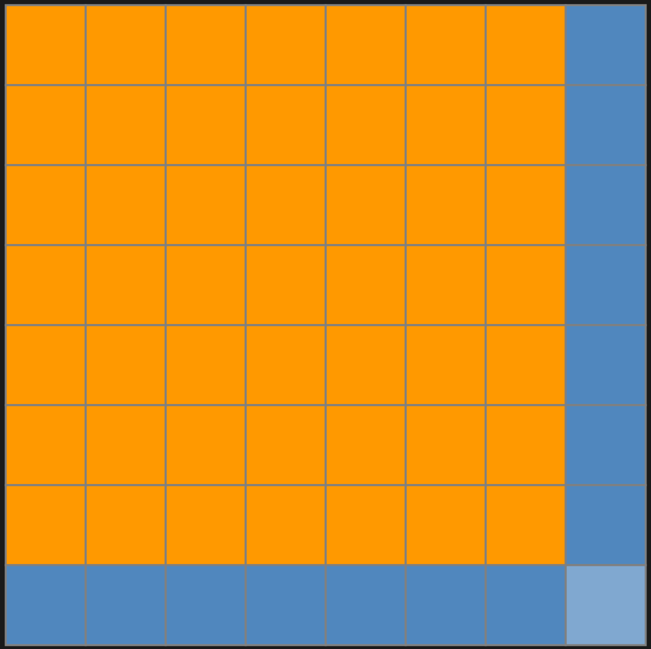
Increasing the Number of Blocks



Pass 7

- Primary blocks are along the diagonal
- Secondary blocks are the rows and columns of the primary block
- Tertiary blocks are all remaining blocks

Increasing the Number of Blocks



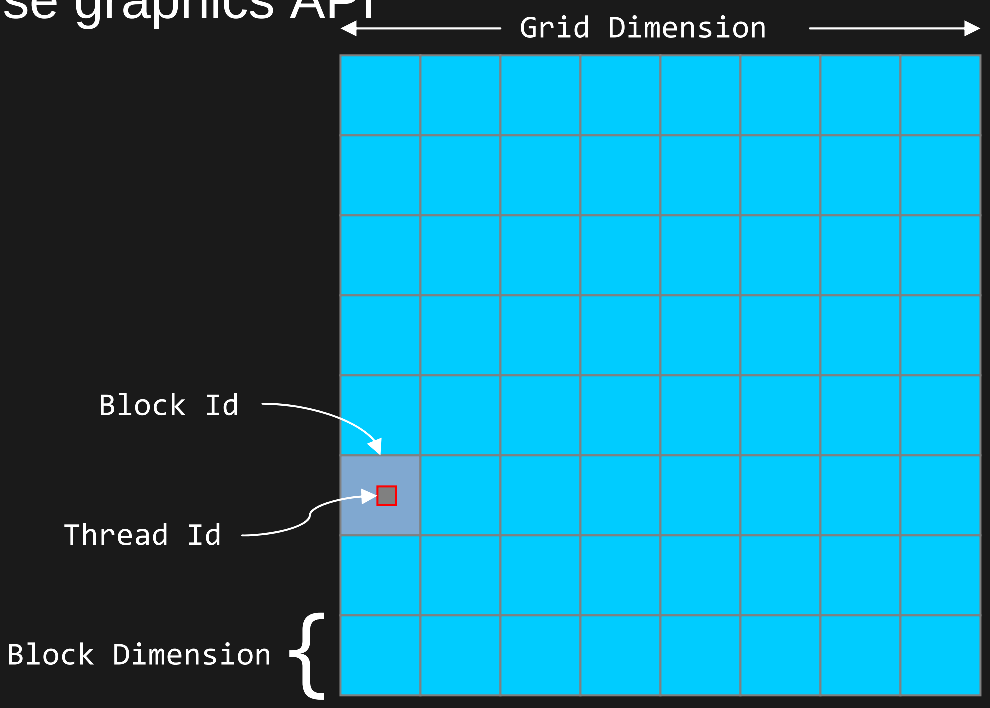
- Primary blocks are along the diagonal
- Secondary blocks are the rows and columns of the primary block
- Tertiary blocks are all remaining blocks

Pass 8

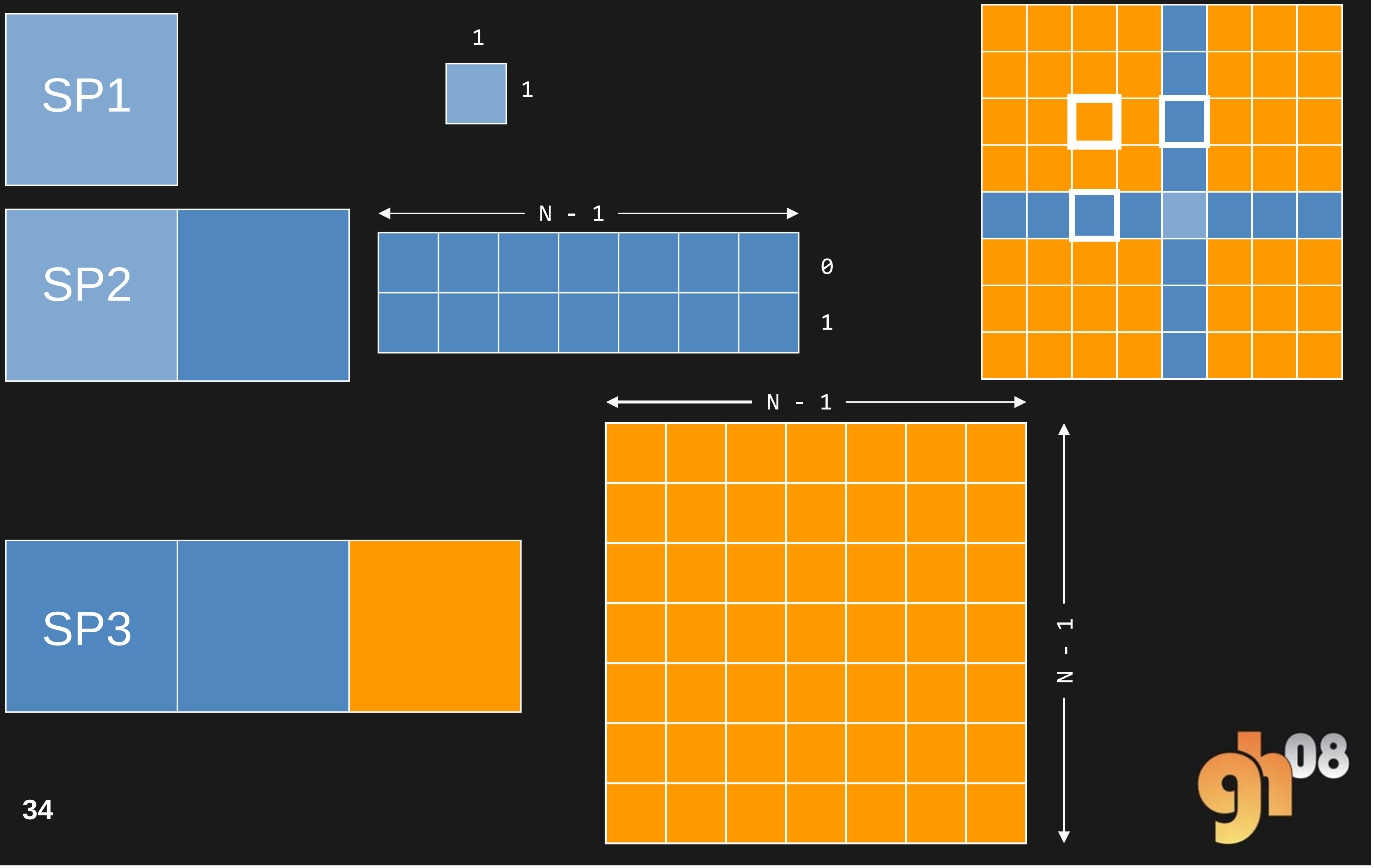
In Total:
N Passes
3 sub-passes per pass

Running it on the GPU

- Using CUDA
    - Written by NVIDIA to access GPU as a parallel processor
    - Do not need to use graphics API
- Memory Indexing
    - CUDA Provides
        - Grid Dimension
        - Block Dimension
        - Block Id
        - Thread Id



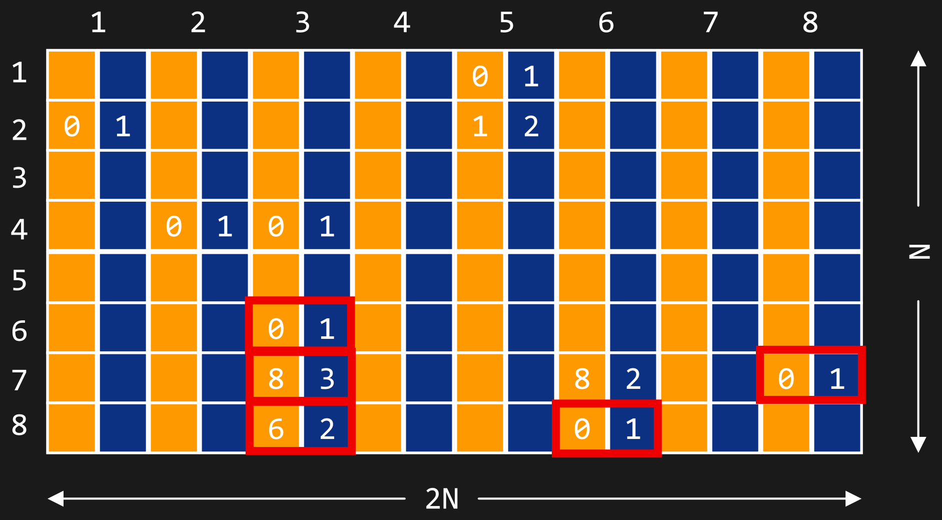
Partial Memory Indexing



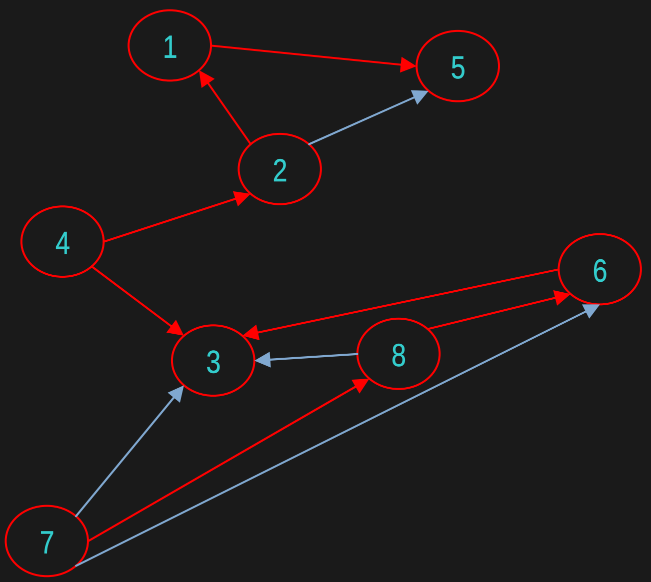
Memory Format for All-Pairs Solution

All-Pairs requires twice the memory footprint of Transitive Closure

Connecting Node	Distance



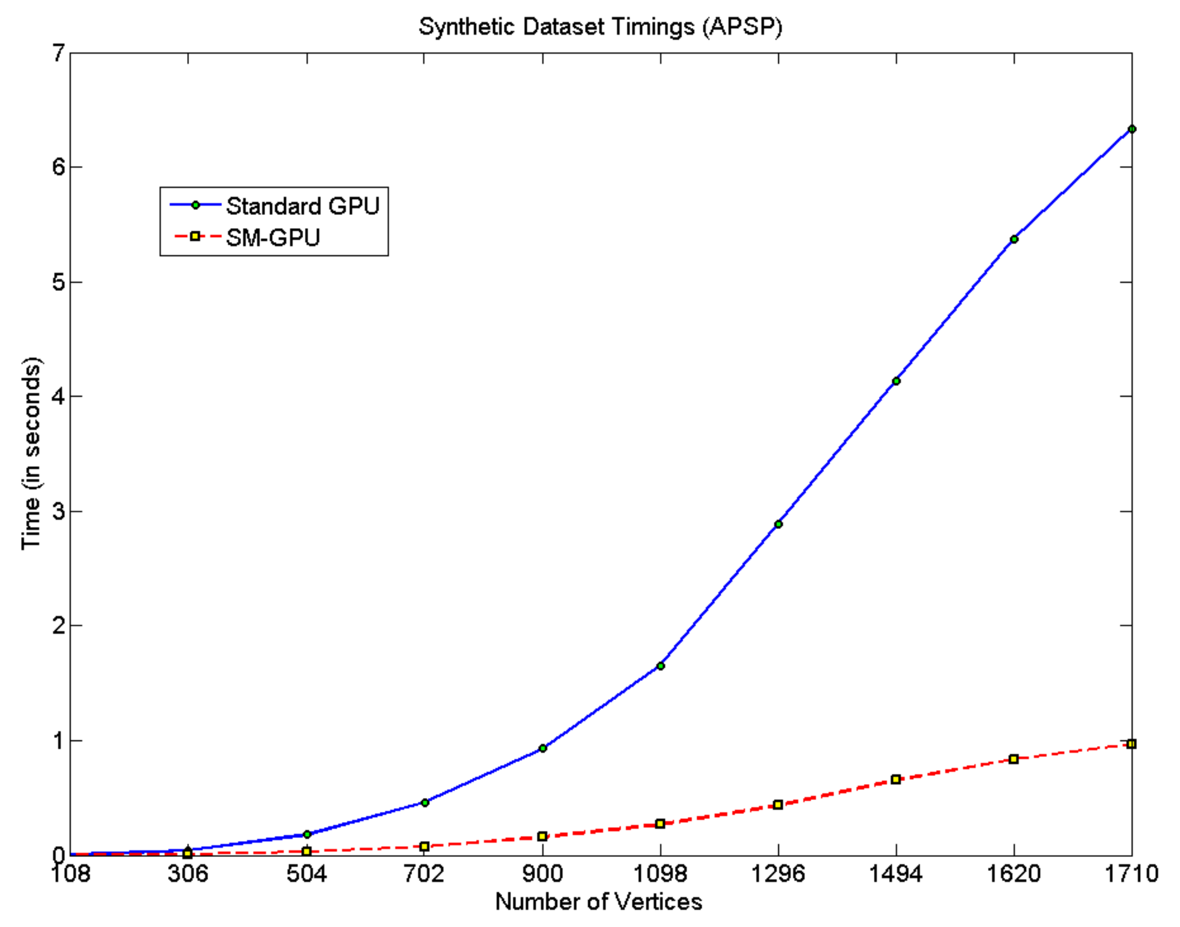
	1		2		3		4		5		6		7		8	
1									0	1						
2	0	1							1	2						
3																
4			0	1	0	1										
5																
6					0	1										
7					8	3					8	2			0	1
8					6	2					0	1				



7 83 6

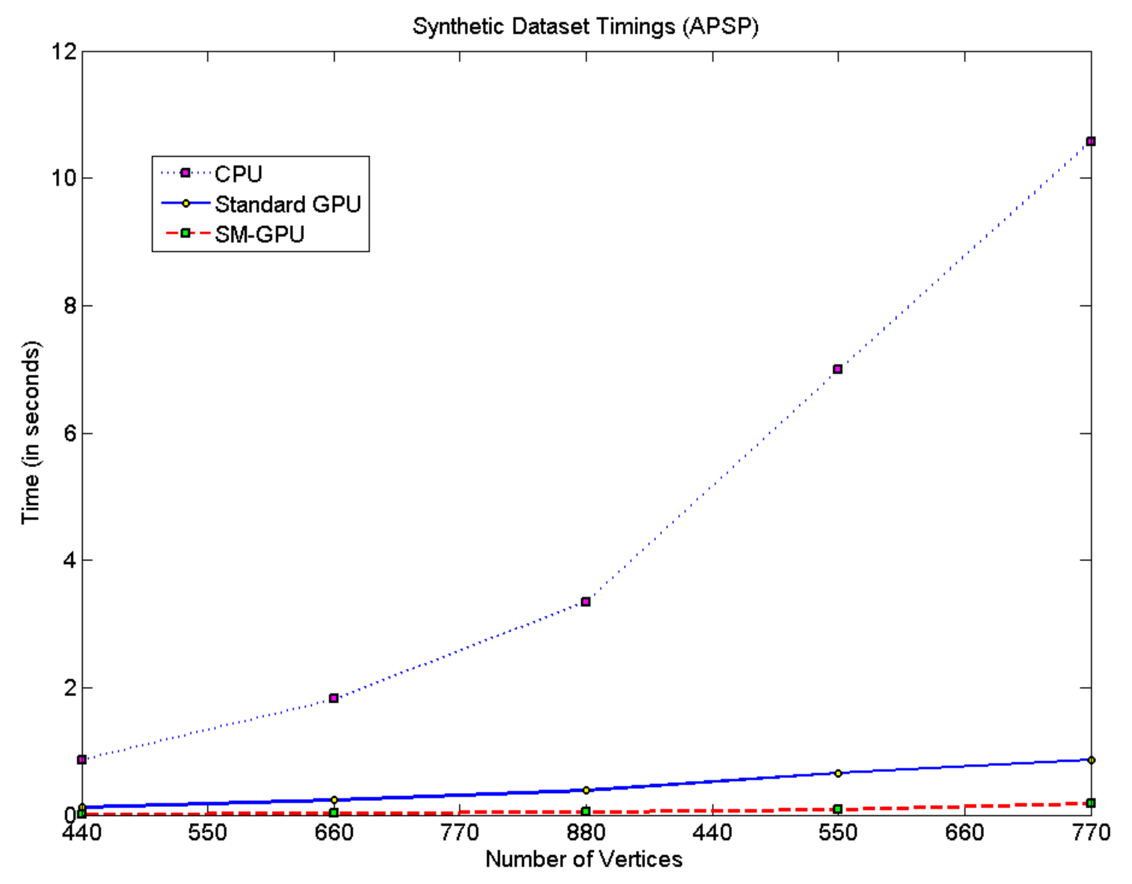
Shortest Path

Results



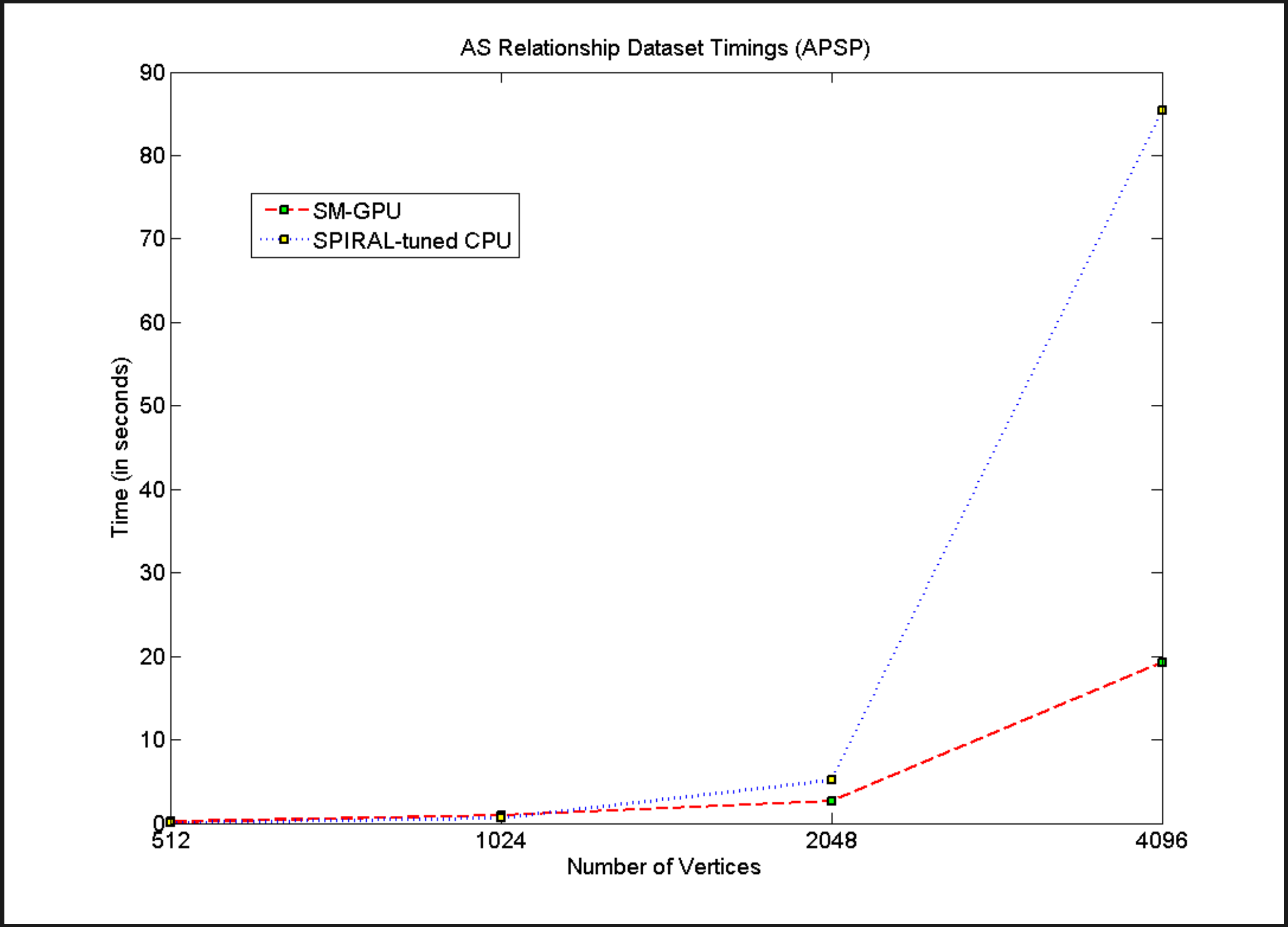
SM cache efficient GPU implementation compared to standard GPU implementation

Results



SM cache efficient GPU implementation compared to standard CPU implementation and cache-efficient CPU implementation

Results



SM cache efficient GPU implementation compared to best variant of Han et al.'s tuned code

Conclusion

•Advantages of Algorithm

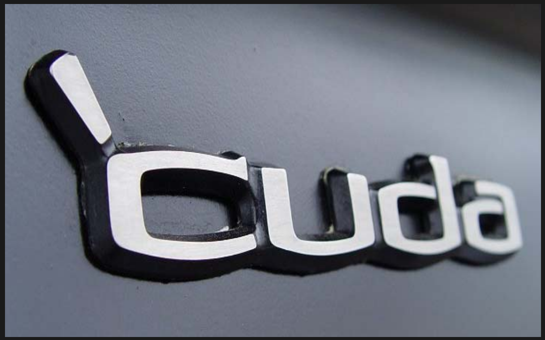
- Relatively Easy to Implement
- Cheap Hardware
- Much Faster than standard CPU version
- Can work for any data size

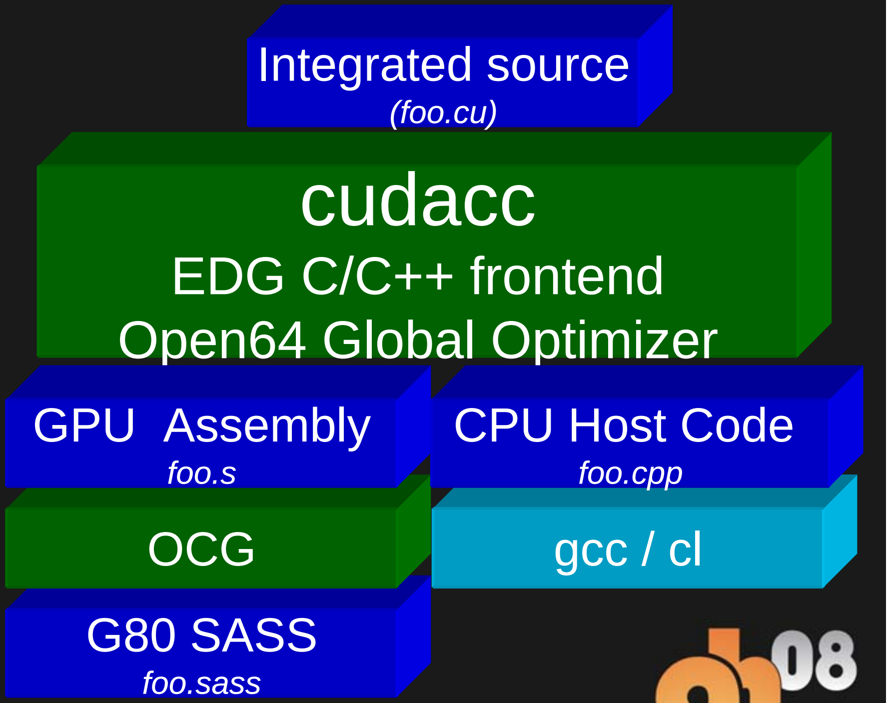
Special thanks to NVIDIA for supporting our research

Backup

CUDA

- CompUte Driver Architecture
- Extension of C
- Automatically creates thousands of threads to run on a graphics card
- Used to create non-graphical applications
- Pros:
  - Allows user to design algorithms that will run in parallel
  - Easy to learn, extension of C
  - Has CPU version, implemented by kicking off threads
- Cons:
  - Low level, C like language
  - Requires understanding of GPU architecture to fully exploit





Integrated source
(foo.cu)

cudacc
EDG C/C++ frontend
Open64 Global Optimizer

GPU Assembly
foo.s

CPU Host Code
foo.cpp

OCG

gcc / cl

G80 SASS
foo.sass