Parallelization of Shortest Path Graph Kernels on Multi-Core CPUs and GPU

Lifan Xu **Wei Wang**
Marco A. Alvarez **John Cavazos** **Dongping Zhang**

Department of Computer and Information Science
University of Delaware

Outline

- Introduction
  - Graph
  - Graph kernel
  - Shortest Path Graph Kernel (SPGK)
- Fast Computation of Shortest Path graph kernel (FCSP)
- Parallelization of FCSP on CPU and GPU
  - Two OpenMP implementations
  - Four GPU implementations
  - Hybrid method
- Experiments results
  - Synthetic datasets
  - Scientific datasets
- Conclusion and Future Work

Graph

- A *graph* G is a set of vertices $\boldsymbol{V}$ and edges $\boldsymbol{E}$, where $\boldsymbol{E} \subset \boldsymbol{V}^2$
- A *graph* G may have labels on vertices and/or edges
- The *adjacency matrix* $\boldsymbol{A}$ of G is defined as

$$[\boldsymbol{A}_{ij}] = \begin{cases} 1 & if\ (v_i, v_j) \in \boldsymbol{E} \\ 0 & otherwise \end{cases}$$

Graph Kernel

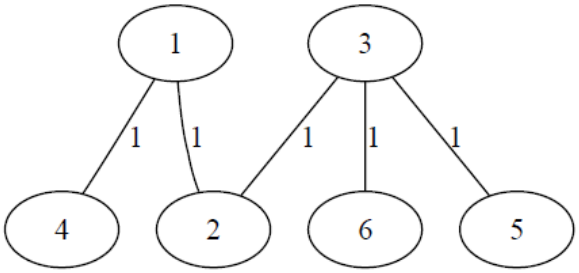
- Kernels (from machine learning) between pairs of graphs (roughly speaking -> graph similarities)

- Examples:
  - Random Walk Kernel
    - Comparing walks
  - Shortest Path Kernel
    - Comparing shortest paths
  - Subtree Kernel
    - Comparing subtree-like patterns
  - Cyclic Pattern Kernel
    - Comparing simple cycles
  - Graphlet Kernel
    - Counting subgraphs of limited size

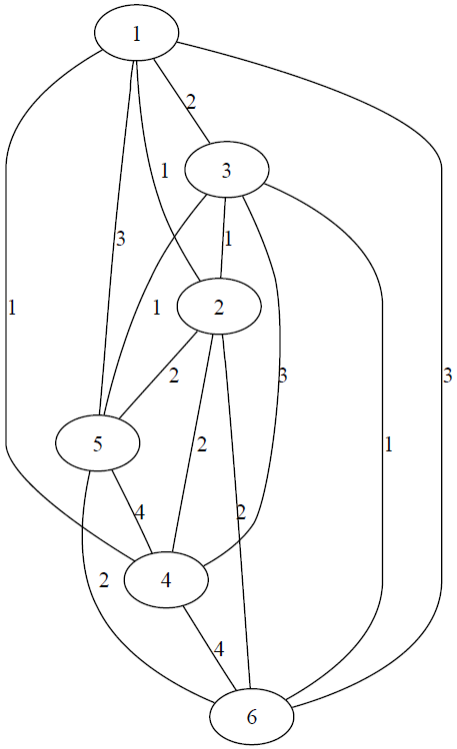
Shortest-Path Graph Kernel (SPGK)

- Convert graph to all pair shortest path graph
  - Can use Floyd-Warshall Algorithm

Floyd-Warshall

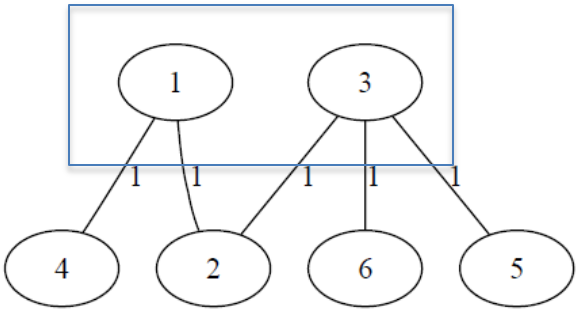


Original Graph

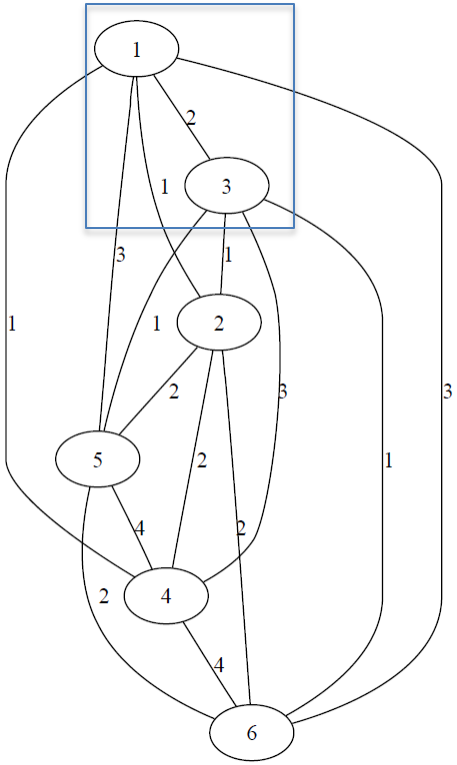


Shortest Path Graph

Floyd-Warshall



Original Graph



Shortest Path Graph

Shortest Path Graph Kernel

- Apply shortest path kernel
  - $K_{sp(G,G')} = \sum_{e \in E} \sum_{e' \in E'} K_{walk}(e, e')$

Shortest Path Graph Kernel

- Apply shortest path kernel
  - $K_{sp(G,G')} = \sum_{e \in E} \sum_{e' \in E'} K_{walk}(e, e')$
  - $K_{walk}(e, e') = K_{node}(u, u') \cdot K_{edge}(e, e') \cdot K_{node}(v, v')$

Shortest Path Graph Kernel

- Apply shortest path kernel
  - $K_{sp(G,G')} = \sum_{e \in E} \sum_{e' \in E'} K_{walk}(e, e')$
  - $K_{walk}(e, e') = K_{node}(u, u') \cdot K_{edge}(e, e') \cdot K_{node}(v, v')$
  - K_{node} is a valid kernel function for comparing two vertices
  - K_{edge} is a valid kernel function for comparing two edges

Shortest Path Graph Kernel

- Lines 2-4 loop through all paths in G1

```
1:  K ← 0
2:  for i, j = 0 → n_node[g1] do
3:      w1 ← sp_mat[g1][i][j]
4:      if i ≠ j AND w1 ≠ INF then
5:          for m, n = 0 → n_node[g2] do
6:              w2 ← sp_mat[g2][m][n]
7:              if m ≠ n AND w2 ≠ INF then
8:                  k_edge ← EdgeKernel(w1, w2)
9:                  if k_edge > 0 then
10:                     k_node1 ← NodeKernel(g1, g2, i, m)
11:                     k_node2 ← NodeKernel(g1, g2, j, n)
12:                     K+ = k_node1 ∗ k_edge ∗ k_node2
13:                 end if
14:             end if
15:         end for
16:     end if
17: end for
18: return K
```

Shortest Path Graph Kernel

- Lines 2-4 loop through all paths in G1
- Lines 5-7 loop through all paths in G2

```
1:  K ← 0
2:  for i, j = 0 → n_node[g1] do
3:      w1 ← sp_mat[g1][i][j]
4:      if i ≠ j AND w1 ≠ INF then
5:          for m, n = 0 → n_node[g2] do
6:              w2 ← sp_mat[g2][m][n]
7:              if m ≠ n AND w2 ≠ INF then
8:                  k_edge ← EdgeKernel(w1, w2)
9:                  if k_edge > 0 then
10:                     k_node1 ← NodeKernel(g1, g2, i, m)
11:                     k_node2 ← NodeKernel(g1, g2, j, n)
12:                     K+ = k_node1 ∗ k_edge ∗ k_node2
13:                 end if
14:             end if
15:         end for
16:     end if
17: end for
18: return K
```

Shortest Path Graph Kernel

- Lines 2-4 loop through all paths in G1
- Lines 5-7 loop through all paths in G2
- Line 8 calculates $K_{edge}(e, e')$

```
1:  K ← 0
2:  for i, j = 0 → n_node[g1] do
3:      w1 ← sp_mat[g1][i][j]
4:      if i ≠ j AND w1 ≠ INF then
5:          for m, n = 0 → n_node[g2] do
6:              w2 ← sp_mat[g2][m][n]
7:              if m ≠ n AND w2 ≠ INF then
8:                  k_edge ← EdgeKernel(w1, w2)
9:                  if k_edge > 0 then
10:                     k_node1 ← NodeKernel(g1, g2, i, m)
11:                     k_node2 ← NodeKernel(g1, g2, j, n)
12:                     K+ = k_node1 * k_edge * k_node2
13:                 end if
14:             end if
15:         end for
16:     end if
17: end for
18: return K
```

Shortest Path Graph Kernel

- Lines 2-4 loop through all paths in G1
- Lines 5-7 loop through all paths in G2
- Line 8 calculates $K_{edge}(e, e')$
- Lines 10-11 calculate $K_{node}(v, v')$

```
1:  K ← 0
2:  for i, j = 0 → n_node[g1] do
3:      w1 ← sp_mat[g1][i][j]
4:      if i ≠ j AND w1 ≠ INF then
5:          for m, n = 0 → n_node[g2] do
6:              w2 ← sp_mat[g2][m][n]
7:              if m ≠ n AND w2 ≠ INF then
8:                  k_edge ← EdgeKernel(w1, w2)
9:                  if k_edge > 0 then
10:                     k_node1 ← NodeKernel(g1, g2, i, m)
11:                     k_node2 ← NodeKernel(g1, g2, j, n)
12:                     K+ = k_node1 ∗ k_edge ∗ k_node2
13:                 end if
14:             end if
15:         end for
16:     end if
17: end for
18: return K
```

Shortest Path Graph Kernel

- Lines 2-4 loop through all paths in G1
- Lines 5-7 loop through all paths in G2
- Line 8 calculates $K_{edge}(e, e')$
- Lines 10-11 calculate $K_{node}(v, v')$
- Line 12 calculates $K_{walk}(e, e')$

```
1:  K ← 0
2:  for i, j = 0 → n_node[g1] do
3:      w1 ← sp_mat[g1][i][j]
4:      if i ≠ j AND w1 ≠ INF then
5:          for m, n = 0 → n_node[g2] do
6:              w2 ← sp_mat[g2][m][n]
7:              if m ≠ n AND w2 ≠ INF then
8:                  k_edge ← EdgeKernel(w1, w2)
9:                  if k_edge > 0 then
10:                     k_node1 ← NodeKernel(g1, g2, i, m)
11:                     k_node2 ← NodeKernel(g1, g2, j, n)
12:                     K+ = k_node1 ∗ k_edge ∗ k_node2
13:                 end if
14:             end if
15:         end for
16:     end if
17: end for
18: return K
```

Drawbacks of SPGK

- Four *for* loops and two *if* statements
- Redundant computation of $K_{node}(v, v')$
- Random memory access

Fast Computation of Shortest Path Graph Kernel (FCSP)

- Compute all $K_{node}(v, v')$ before $K_{walk}(e, e')$
- Convert shortest path adjacency matrix to coordinate lists (sparse matrix representation)
  - One array for value
  - One array for row
  - One array for column

- Lines 1-7 compute all $K_{node}(v, v')$

```
1: function VERTEX_KERNEL
2:     for i = 0 → n_node[g1] do
3:         for j = 0 → n_node[g2] do
4:             V[i][j] ← NodeKernel(g1, g2, i, j)
5:         end for
6:     end for
7: end function
8:
9: function WALK_KERNEL
10:    K ← 0
11:    for i = 0 → n_node[g1] do
12:        x1 ← edge_x1_g1[i]
13:        y1 ← edge_y1_g1[i]
14:        w1 ← edge_w1_g1[i]
15:        for j = 0 → n_node[g2] do
16:            x2 ← edge_x2_g2[j]
17:            y2 ← edge_y2_g2[j]
18:            w2 ← edge_w2_g2[j]
19:            k_edge ← EdgeKernel(w1, w2)
20:            if k_edge > 0 then
21:                k_node1 ← V[x1][x2]
22:                k_node2 ← V[y1][y2]
23:                K+ = k_node1 * k_edge * k_node2
24:            end if
25:        end for
26:    end for
27:    return K
28: end function
```

- Lines 1-7 compute all $K_{node}(v, v')$
- Lines 11-14 loop all paths in G1

```
1:  function VERTEX_KERNEL
2:      for i = 0 → n_node[g1] do
3:          for j = 0 → n_node[g2] do
4:              V[i][j] ← NodeKernel(g1, g2, i, j)
5:          end for
6:      end for
7:  end function
8:
9:  function WALK_KERNEL
10:     K ← 0
11:     for i = 0 → n_node[g1] do
12:         x1 ← edge_x1_g1[i]
13:         y1 ← edge_y1_g1[i]
14:         w1 ← edge_w1_g1[i]
15:         for j = 0 → n_node[g2] do
16:             x2 ← edge_x2_g2[j]
17:             y2 ← edge_y2_g2[j]
18:             w2 ← edge_w2_g2[j]
19:             k_edge ← EdgeKernel(w1, w2)
20:             if k_edge > 0 then
21:                 k_node1 ← V[x1][x2]
22:                 k_node2 ← V[y1][y2]
23:                 K+ = k_node1 * k_edge * k_node2
24:             end if
25:         end for
26:     end for
27:     return K
28: end function
```

- Lines 1-7 compute all $K_{node}(v, v')$
- Lines 11-14 loop all paths in G1
- Lines 15-18 loop all paths in G2

```
1: function VERTEX_KERNEL
2:     for i = 0 → n_node[g1] do
3:         for j = 0 → n_node[g2] do
4:             V[i][j] ← NodeKernel(g1, g2, i, j)
5:         end for
6:     end for
7: end function
8:
9: function WALK_KERNEL
10:    K ← 0
11:    for i = 0 → n_node[g1] do
12:        x1 ← edge_x1_g1[i]
13:        y1 ← edge_y1_g1[i]
14:        w1 ← edge_w1_g1[i]
15:        for j = 0 → n_node[g2] do
16:            x2 ← edge_x2_g2[j]
17:            y2 ← edge_y2_g2[j]
18:            w2 ← edge_w2_g2[j]
19:            k_edge ← EdgeKernel(w1, w2)
20:            if k_edge > 0 then
21:                k_node1 ← V[x1][x2]
22:                k_node2 ← V[y1][y2]
23:                K+ = k_node1 * k_edge * k_node2
24:            end if
25:        end for
26:    end for
27:    return K
28: end function
```

- Lines 1-7 compute all $K_{node}(v, v')$
- Lines 11-14 loop all paths in G1
- Lines 15-18 loop all paths in G2
- Line 19 computes $K_{edge}(e, e')$

```
1: function VERTEX_KERNEL
2:     for i = 0 → n_node[g1] do
3:         for j = 0 → n_node[g2] do
4:             V[i][j] ← NodeKernel(g1, g2, i, j)
5:         end for
6:     end for
7: end function
8:
9: function WALK_KERNEL
10:    K ← 0
11:    for i = 0 → n_node[g1] do
12:        x1 ← edge_x1_g1[i]
13:        y1 ← edge_y1_g1[i]
14:        w1 ← edge_w1_g1[i]
15:        for j = 0 → n_node[g2] do
16:            x2 ← edge_x2_g2[j]
17:            y2 ← edge_y2_g2[j]
18:            w2 ← edge_w2_g2[j]
19:            k_edge ← EdgeKernel(w1, w2)
20:            if k_edge > 0 then
21:                k_node1 ← V[x1][x2]
22:                k_node2 ← V[y1][y2]
23:                K+ = k_node1 * k_edge * k_node2
24:            end if
25:        end for
26:    end for
27:    return K
28: end function
```

- Lines 1-7 compute all $K_{node}(v, v')$
- Lines 11-14 loop all paths in G1
- Lines 15-18 loop all paths in G2
- Line 19 computes $K_{edge}(e, e')$
- Lines 21-22 fetch $K_{node}(v, v')$

```
1:  function VERTEX_KERNEL
2:      for i = 0 → n_node[g1] do
3:          for j = 0 → n_node[g2] do
4:              V[i][j] ← NodeKernel(g1, g2, i, j)
5:          end for
6:      end for
7:  end function
8:
9:  function WALK_KERNEL
10:     K ← 0
11:     for i = 0 → n_node[g1] do
12:         x1 ← edge_x1_g1[i]
13:         y1 ← edge_y1_g1[i]
14:         w1 ← edge_w1_g1[i]
15:         for j = 0 → n_node[g2] do
16:             x2 ← edge_x2_g2[j]
17:             y2 ← edge_y2_g2[j]
18:             w2 ← edge_w2_g2[j]
19:             k_edge ← EdgeKernel(w1, w2)
20:             if k_edge > 0 then
21:                 k_node1 ← V[x1][x2]
22:                 k_node2 ← V[y1][y2]
23:                 K+ = k_node1 * k_edge * k_node2
24:             end if
25:         end for
26:     end for
27:     return K
28: end function
```

- Lines 1-7 compute all $K_{node}(v, v')$
- Lines 11-14 loop all paths in G1
- Lines 15-18 loop all paths in G2
- Line 19 computes $K_{edge}(e, e')$
- Lines 21-22 fetch $K_{node}(v, v')$
- Line 23 computes $K_{walk}(e, e')$

```
1:  function VERTEX_KERNEL
2:      for i = 0 → n_node[g1] do
3:          for j = 0 → n_node[g2] do
4:              V[i][j] ← NodeKernel(g1, g2, i, j)
5:          end for
6:      end for
7:  end function
8:
9:  function WALK_KERNEL
10:     K ← 0
11:     for i = 0 → n_node[g1] do
12:         x1 ← edge_x1_g1[i]
13:         y1 ← edge_y1_g1[i]
14:         w1 ← edge_w1_g1[i]
15:         for j = 0 → n_node[g2] do
16:             x2 ← edge_x2_g2[j]
17:             y2 ← edge_y2_g2[j]
18:             w2 ← edge_w2_g2[j]
19:             k_edge ← EdgeKernel(w1, w2)
20:             if k_edge > 0 then
21:                 k_node1 ← V[x1][x2]
22:                 k_node2 ← V[y1][y2]
23:                 K+ = k_node1 * k_edge * k_node2
24:             end if
25:         end for
26:     end for
27:     return K
28: end function
```

Calculating a Kernel Matrix

- Given a set of graphs $\{g_1, g_2, \ldots, g_n\}$
- Calculate the kernel matrix $\boldsymbol{K_{nxn}}$
  - $\boldsymbol{K_{(i,j)}}$ is the similarity between g_i and g_j

FCSP on Multi-Core CPU using OpenMP

- OpenMP_In
  - Parallelize computation of a pair of graphs
  - Dynamic parallel for pragma
    - Vertex Kernel
    - Walk kernel
- OpenMP_Out
  - Parallelize computation of the whole kernel matrix
  - Each OpenMP thread fetches a pair of graphs until all computation are finished

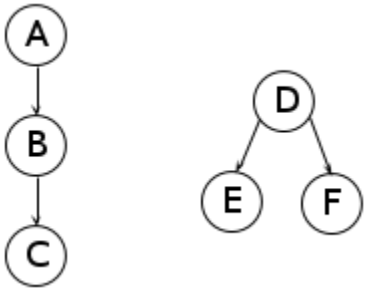
FCSP on GPU using OpenCL

- Three OpenCL kernels
  - Vertex kernel
  - Walk kernel
  - Reduction kernel
- Four implementations
  - GPU_1D
  - GPU_2D
  - GPU_1D_overlap
  - GPU_2D_overlap

GPU_1D

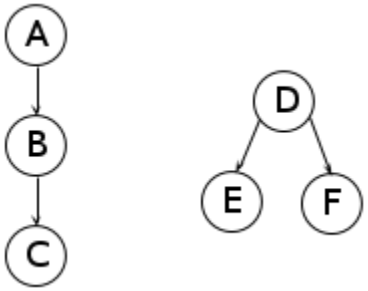
- 2D domain decomposition for ***Vertex Kernel***
- 1D domain decomposition for ***Walk Kernel***

GPU_1D



Input graphs

GPU_1D



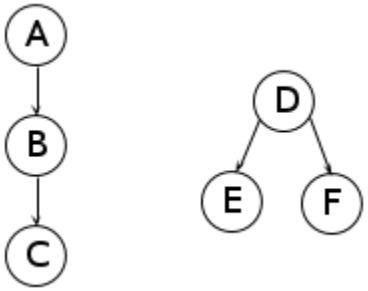
Input graphs

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

Adjacency matrix

GPU_1D



Input graphs

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

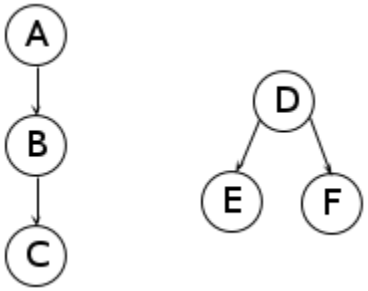
Adjacency matrix

	A	B	C
A	0	1	2
B	0	0	1
C	0	0	0

	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

Shortest Path Adjacency matrix

GPU_1D



Input graphs

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

Adjacency matrix

	A	B	C
A	0	1	2
B	0	0	1
C	0	0	0

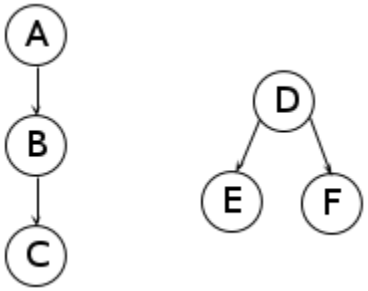
	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

Shortest Path Adjacency matrix

$_0$ A D	$_1$ A E	$_2$ A F
$_3$ B D	$_4$ B E	$_5$ B F
$_6$ C D	$_7$ C E	$_8$ C F

Vertex Kernel

GPU_1D



Input graphs

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

Adjacency matrix

	A	B	C
A	0	1	2
B	0	0	1
C	0	0	0

	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

Shortest Path Adjacency matrix

0 A D	1 A E	2 A F
3 B D	4 B E	5 B F
6 C D	7 C E	8 C F

Vertex Kernel

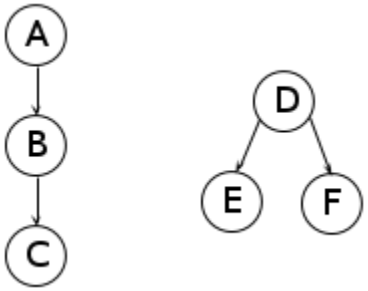
0	A→B D→E	A→B D→F
1	A→C D→E	A→C D→F
2	B→C D→E	B→C D→F

Walk Kernel

GPU_2D

- 2D domain decomposition for ***Vertex Kernel***
- 2D domain decomposition for ***Walk Kernel***

GPU_2D



Input graphs

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

Adjacency matrix

	A	B	C
A	0	1	2
B	0	0	1
C	0	0	0

	D	E	F
D	0	1	1
E	0	0	0
F	0	0	0

Shortest Path Adjacency matrix

0 A D	1 A E	2 A F
3 B D	4 B E	5 B F
6 C D	7 C E	8 C F

Vertex Kernel

0 A→B, D→E	1 A→B, D→F
2 A→C, D→E	3 A→C, D→F
4 B→C, D→E	5 B→C, D→F

Edge Kernel

GPU_1D_overlap

- 2D domain decomposition for ***Vertex Kernel***
- 1D domain decomposition for ***Walk Kernel***
- Computation and Communication overlap
  - Issue non-blocking memory transfer after Reduction Kernel
  - Assign next pair of graphs to non-blocking Vertex Kernel and Walk Kernel
  - CPU accumulates results from Reduction kernel meanwhile

GPU_2D_overlap

- 2D domain decomposition for ***Vertex Kernel***
- 2D domain decomposition for ***Walk Kernel***
- Computation and Communication overlap
  - Issue non-blocking memory transfer after Reduction Kernel
  - Assign next pair of graphs to non-blocking Vertex Kernel and Walk Kernel
  - CPU accumulates results from Reduction kernel meanwhile

CPU and GPU Hybrid Implementation

- Combine OpenMP_In and GPU_1D_overlap
- Set a threshold T for number of shortest paths
  - Both input graphs smaller than T
    - OpenMP_In
  - Otherwise
    - GPU_1D_overlap

Execution Environment

- **CPU** – Two Intel 5530 Quad core @ 2.4 GHz with 8MB cache (16 OpenMP threads)
- **GPU** - One NVIDIA C2050 (448 Cores @ 1.15GHz) with 3GB GDDR5 1.5 GHZ ECC RAM

Homogeneous Synthetic Datasets

- Nine homogeneous datasets
  - 256 graphs per dataset
  - Each dataset contains graphs of same sizes

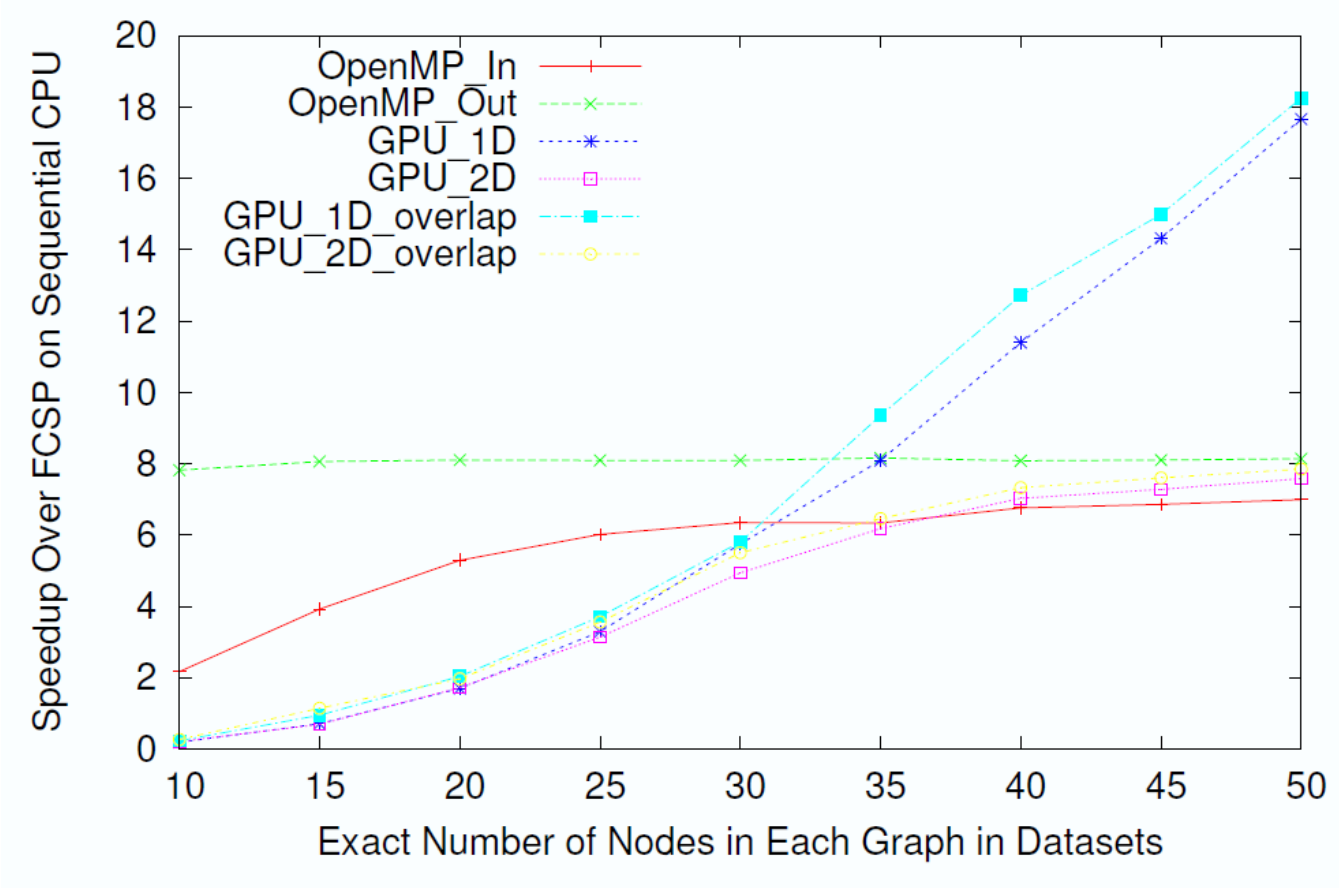
Homogeneous Datasets Statistics

Dataset	Avg. Nodes	Avg. Edges	Avg. SP
10-nodes	10	90	90
15-nodes	15	210	210
20-nodes	20	380	380
25-nodes	25	600	600
30-nodes	30	870	870
35-nodes	35	1190	1190
40-nodes	40	1560	1560
45-nodes	45	1980	1980
50-nodes	50	2450	2450

Speedup of sequential FCSP over sequential SPGK on CPU

Dataset	SPGK time(sec)	FCSP time(sec)	Speedup
10-nodes	127.99	2.35	54.56
15-nodes	695.24	11.69	59.46
20-nodes	2275.60	37.16	61.25
25-nodes	5668.74	91.42	62.00
30-nodes	11990.24	190.82	62.83
35-nodes	26220.74	355.50	73.76
40-nodes	45850.24	609.67	75.21
45-nodes	74817.26	983.35	76.08
50-nodes	115728.37	1513.69	76.45

Speedup of Parallel FCSP over Sequential FCSP



Mixed Synthetic Dataset

- 180 10-nodes graph
- 76 50-nodes graphs

OpenMP_In	*OpenMP_Out*	*GPU_1D*	*GPU_2D*	*GPU_1D_overlap*	*GPU_2D_overlap*	*Hybrid*
24.446	20.349	22.137	32.252	19.751	29.336	**19.042**

Different Implementation Running Time(seconds) on the Mixed Dataset

Scientific Datasets

Dataset	Num. of Graphs	Avg. Nodes	Avg. Edges	Min. SP	Max. SP	Avg. SP
MUTAG	188	17	39	90	756	324
ENZYMES	600	32	124	2	15500	1215
NCI1	4110	29	64	6	11130	1005
NCI109	4127	29	64	12	11130	995

Speedup over OpenMP_In on Scientic Datasets

Dataset	*OpenMP_Out*	*GPU_1D*	*GPU_2D*	*GPU_1D_overlap*	*GPU_2D_overlap*	*Hybrid*
MUTAG	1.33	0.28	0.21	0.33	0.38	0.96
ENZYMES	1.05	1.73	0.78	1.89	0.94	1.92
NCI1	1.13	1.45	0.73	1.66	0.90	1.78
NCI109	1.11	1.38	0.69	1.58	0.87	1.70

Conclusion and Future Work

- We introduce Fast Computation of shortest Path graph kernel
- Sequential FCSP is able to achieve **76x** speedup over sequential SPGK
- Two CPU parallelizations
- Four GPU implementations
- One Hybrid method
- We are going to accelerate other graph kernels in the future

Thanks!

Questions?