CS677 Final Project

- All Pair Shortest Path

May.1.2019

By. Yiding Yang

1, APSP Revisited

-What's the problem and why is it important

- Aiming to determinate the shortest distances between every pair of vertices in a directed graph
- The result in the form of matrix called distance matrix

What we can do with the distance matrix:

- Get the shortest path from every pair of vertices
- Base problem for many other problems

2, CPU implementation

- Floyd-Warshall algorithm
  - an iteration algorithm over k(number of vertices)
  - relax the shortest distance for every pair of vertices by insert a new vertice
  - three for loops which runs in O(N^3) time

$k = 0$		j			
		1	2	3	4
i	1	0	∞	−2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k = 1$		j			
		1	2	3	4
i	1	0	∞	−2	∞
	2	4	0	**2**	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k = 2$		j			
		1	2	3	4
i	1	0	∞	−2	∞
	2	4	0	2	∞
	3	∞	∞	0	2
	4	**3**	−1	**1**	0

$k = 3$		j			
		1	2	3	4
i	1	0	∞	−2	**0**
	2	4	0	2	**4**
	3	∞	∞	0	2
	4	3	−1	1	0

$k = 4$		j			
		1	2	3	4
i	1	0	**−1**	−2	0
	2	4	0	2	4
	3	**5**	**1**	0	2
	4	3	−1	1	0

*picture from wikipedia

3, Why is GPU suitable for APSP

- In place computation
  - potential low memory bandwidth

- Almost 100% computing

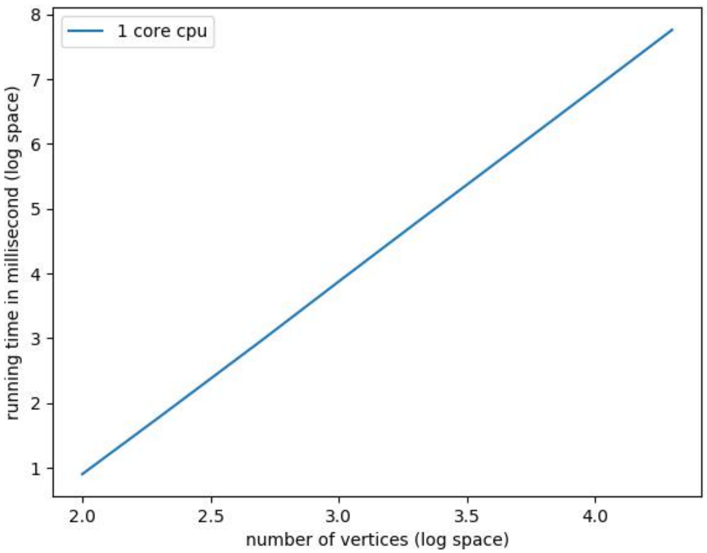
- An recursive algorithm which makes the compute for each pair of vertices independent
  - assign each pair of vertices to one thread without communication

4, Experiments setup

- Graph generation
  - random graph generated by Erdos-Renyi model
  - average degree is 6.5
    - doesn't influence the running time of algorithm unless we use a sparse version
  - number of vertices from 100 to 20000

5, CPU Performance

- CPU version
  - Floyd-Warshall algorithm
  - exact a linear curve when we set the x and y-axes in log space



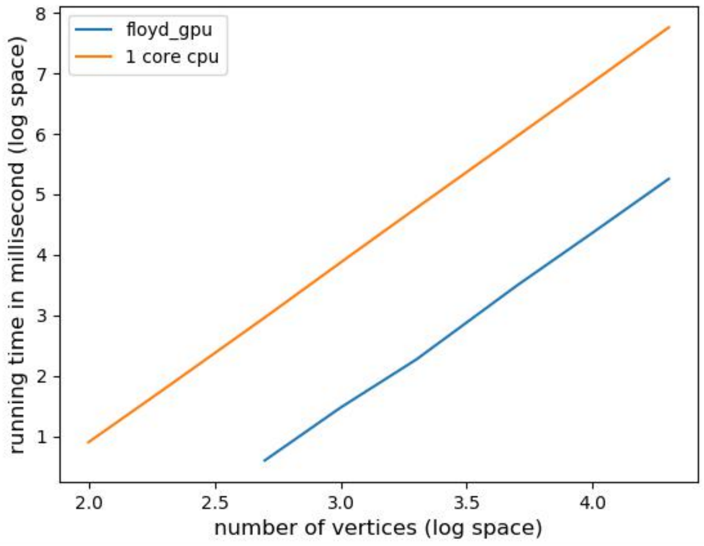
6, GPU implementation

- Iteration version which is same as CPU
  - GPU version Floyd-Warshall
- Recursive version which make it more suitable for GPU
  - R-Kleene algorithm
- Some improvement versions based on R-keene

6, GPU implementation

version 1: GPU version of Floyd-warshall with global memory

- For each k, call GPU kernel
- each thread calculate one pair of vertices by relaxing the shortest path for all other vertices
- The GPU kernel will be called #vertices times



Using shared memory next?

Problem: Each thread needs to access one row and one col of the whole matrix to get the result which make it hard to use shared memory

319x

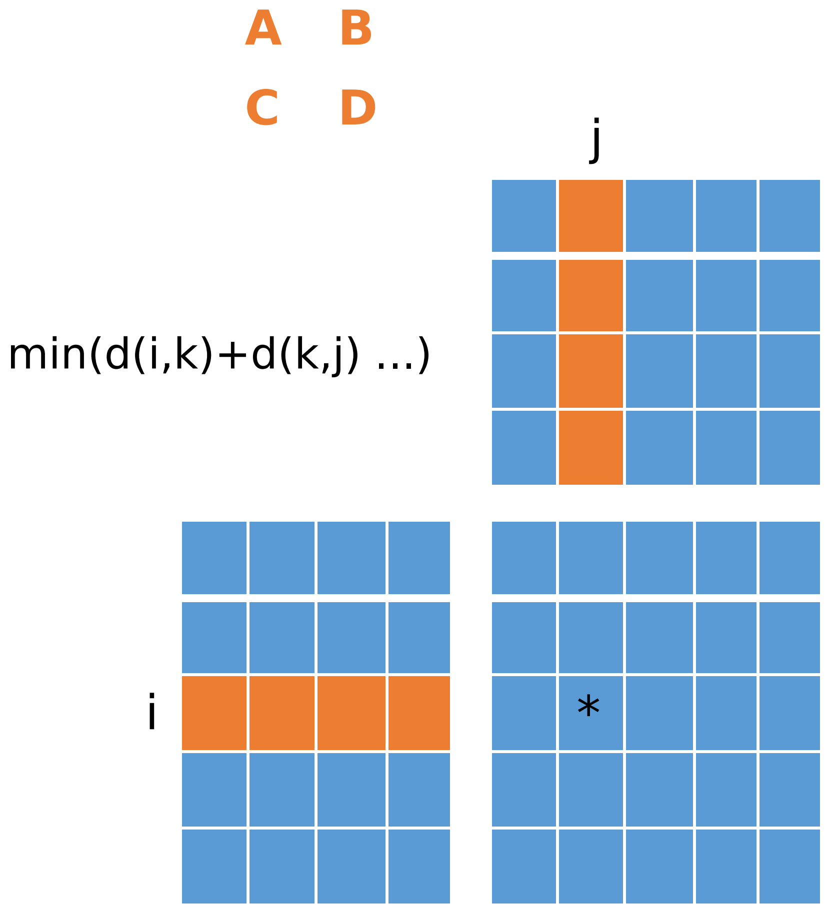
6, GPU implementation

version 2: Recursive with global memory

Recursive R-Kleene algorithm
is more friendly for GPU

R-Kleene(*):

1, R-Kleene(A)
2, Update B using A
3, Update C using A
4, Update D using C and B
4, R-Kleene(D)
5, Update B using D
6, Update C using D
7, Update A using B and C

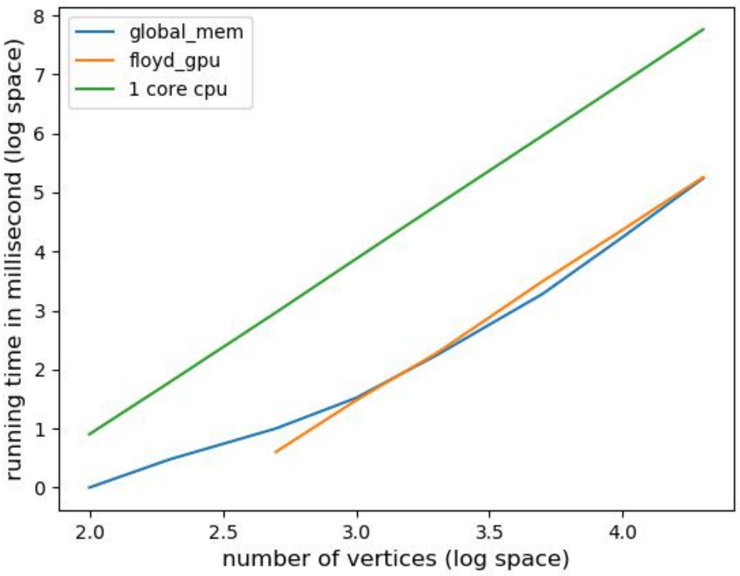


6, GPU implementation

version 2: Recursive with global memory

continue

- The performance of the R-Kleene is no better than the Floyd-Warshall algo.
- But now, we can make use of shared memory

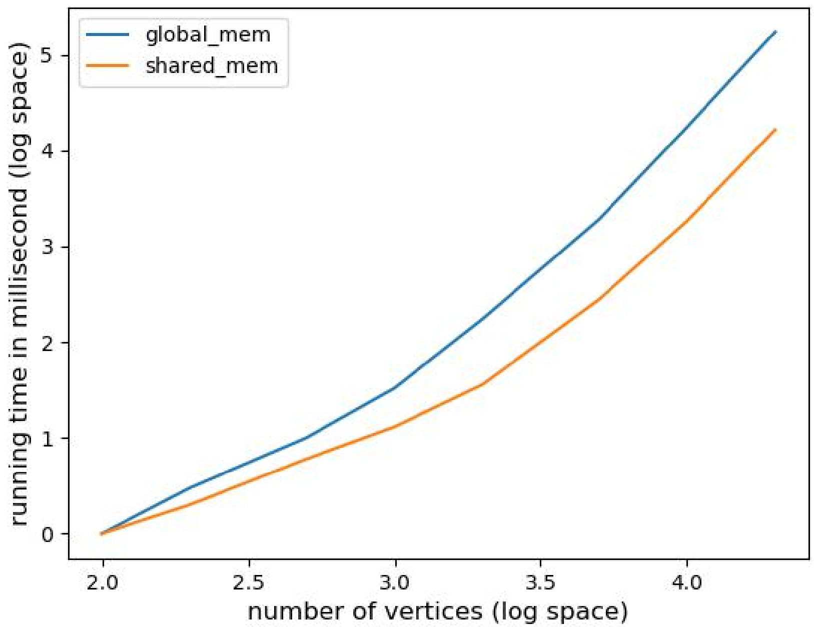


1x

6, GPU implementation

version 3: Recursive with shared memory

- For a single matrix minplus calculation, it is likes the matrix multiplication.
- We can now load a block of matrix once and calculate it's contribution to the target output block



10.5x

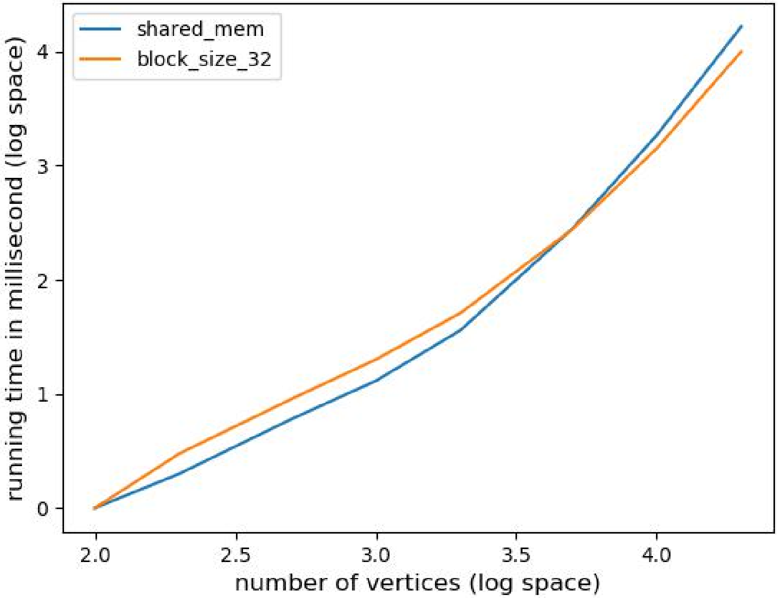
6, GPU implementation

version 4: Recursive with shared memory

- with larger blocksize (32)

- We can even improve the performance by simply making a larger blocksize
- We get a 1.65x speed up by change the blocksize from 16 to 32

Now, the problem is we have slow speed in small graph respect to the large one

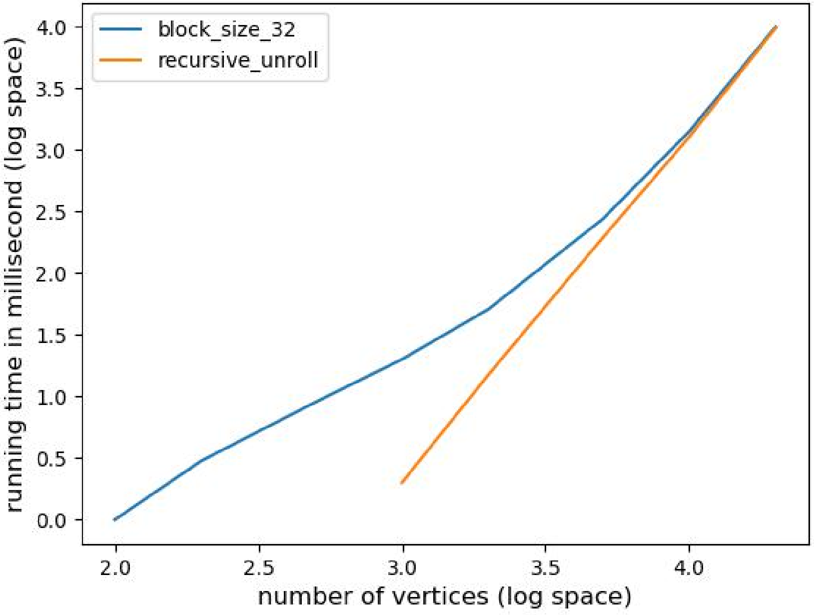


1.65x

6, GPU implementation

version 5: Unroll the recursive

- The bottleneck for small graph is in the end of recursive where the overhead kernel launch cann't be ignored
- Unroll in the end of recursive
- When we get a matrix which is smaller than blocksize, we can just calculate the APSP by using floyd-warshall algo.
    - Now, we can load the whole matrix to shared memory

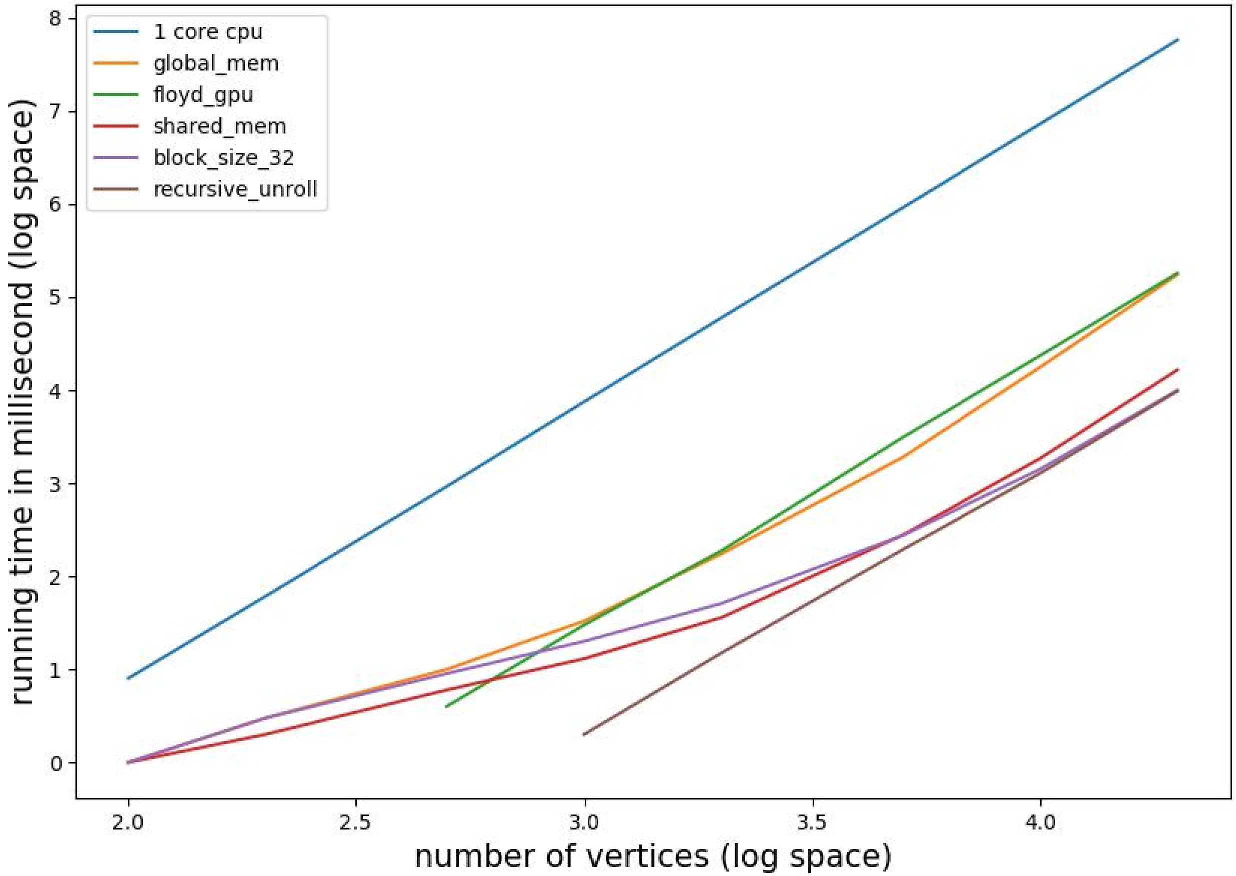


Finally, we get an almost linear curve in log space

6, GPU implementation

- version of gpu-floyd-warshall
  - Used 12 registers, 336 bytes cmem[0]
- version with global memory
  - Used 32 registers, 385 bytes cmem[0], 4 bytes cmem[2]
- version with shared memory
  - Used 27 registers, 2048 bytes smem, 385 bytes cmem[0]
- version with shared memory and recursive unroll
  -threads per block increace to 32
  - Used 28 registers, 8192 bytes smem, 385 bytes cmem[0]
  - Used 12 registers, 4096 bytes smem, 344 bytes cmem[0]
- all of these versions have a 100% occupancy

7, A big picture



16 hours' computing can now be reduced to only 10 seconds