Parallel Programming hw4-2

tags: PP20

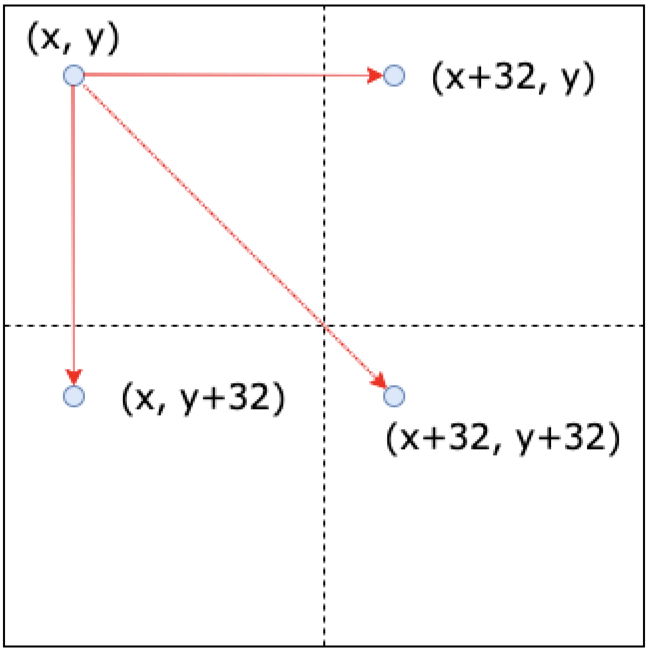
106062230 徐嘉欣

Implementation

首先，我在做input時，有做padding的部分，讓整個2D array的長寬都是64的倍數(因為**Blocking factor取64**)，這樣在device端就不用怕存取到超過memory範圍的部分。

```
fread(&v, sizeof(int), 1, file);
fread(&m, sizeof(int), 1, file);
if(v%64) n = v + (64 - v%64);
else n = v;
cudaMallocHost( &Dist, sizeof(int)*(n*n));
```

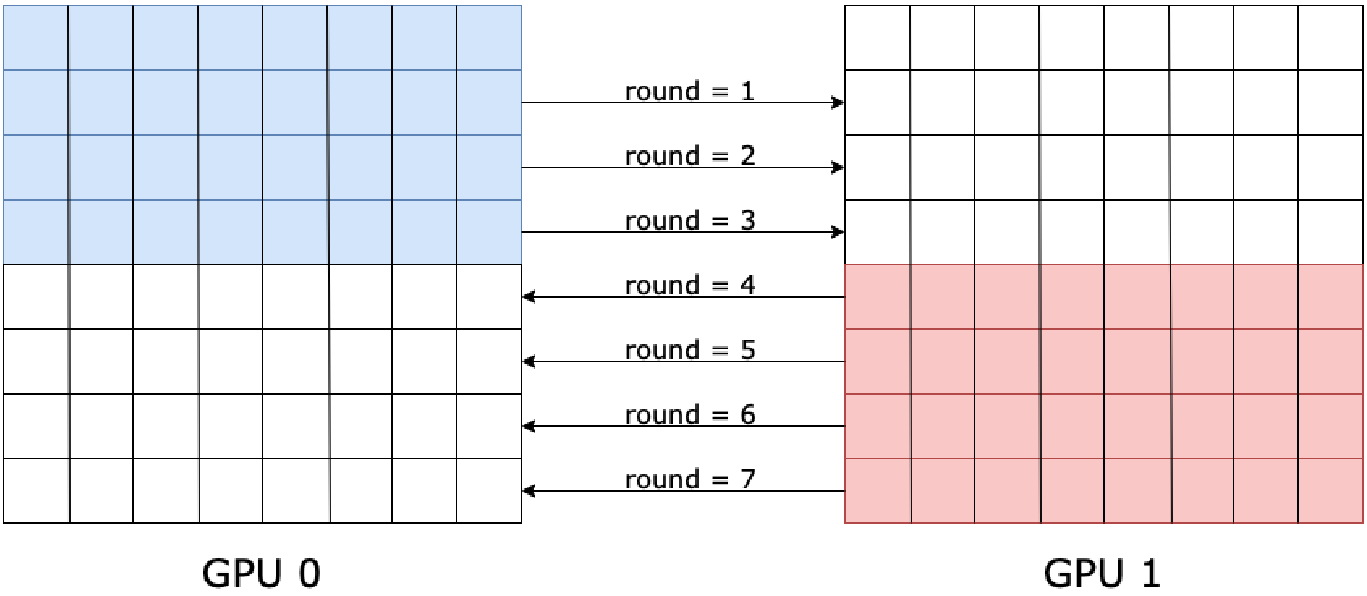
而在做blocked floyd warshall時，因為device的每個block最多只能包含1024個threads，因此這邊取的**thread是dim3(32, 32)**，比起每次都只有做一遍的computing，一次做4遍的computing(有點像在模擬(64,64)的thread)、盡可能最大地利用shared memory的大小會使效能變得好些。



溝通的部分則是因為每個round中，data dependency其實只有pivot-row和pivot-column，因此理論上若兩個GPU一人做上半部、一人做下半部，則必須要將更動過後的pivot-row和pivot-column傳給對方，**但實際上只需要傳pivot-row就可以了**，因為pivot-column可以在phase2的時候透過運算自己更新。

所以phase1與phase2時，因為花費的時間很少，所以直接讓兩個GPU都做，而phase3的切割資料則是讓GPU0做上半部，GPU1做下半部，並且在`r < round/2`時，由GPU0傳遞pivot-row給GPU1(因為此時的pivot-row會由GPU0在phase3時更新)，其餘情況則是GPU1傳遞pivot-row給GPU0(因為此時的pivot-row會由GPU1在phase3時更新)，全部計算完後GPU0將上半部的資料傳回CPU，GPU1將下半部的資料傳回CPU。

下圖則為一個例子，若round=8，則在round0時，因為此時的pivot-row都是由CPU assign過來的，因此不需要誰傳給誰。各自做完round0後，因為GPU0做的是上半部，因此round1時使用到的pivot-row必須要傳給GPU1，以此類推直到round3。而到了round4時，因為GPU1做的是上半部，因此這時候需要將使用到的pivot-row傳給GPU0，以此類推直到round7。




```
dim3 grid2(round-1, 2);
dim3 grid3(round-1, (round/2)+1);
dim3 block(64, 16);
dim3 block2(32, 32);

phase1<<< 1, block2, 4096*sizeof(int) >>>(B, 0,  0,  0,  1,  1, n,
d_dist[thread_num], n);
phase2<<< grid2, block, 8192*sizeof(int) >>>(B, 0, n, d_dist[thread_num],
n);
phase3<<<grid3, block2>>>(B, 0, n, d_dist[thread_num], n, thread_num,
round);

for (int r = 1; r < round; ++r) {

    #pragma omp barrier

    if (r <= (round/2) && thread_num == 1) {
        cudaMemcpyPeer(d_dist[1] + r * B * n, 1, d_dist[0] + r * B * n, 0,
B * n * sizeof(int));
    } else if (r > (round/2) && thread_num == 0) {
        cudaMemcpyPeer(d_dist[0] + r * B * n, 0, d_dist[1] + r * B * n, 1,
B * n * sizeof(int));
    }

    #pragma omp barrier

    phase1<<< 1, block2, 4096*sizeof(int) >>>(B, r,  r,  r,  1,  1, n,
d_dist[thread_num], n);
    phase2<<< grid2, block, 8192*sizeof(int) >>>(B, r, n,
d_dist[thread_num], n);
```

```
    phase3<<<grid3, block2>>>(B, r, n, d_dist[thread_num], n, thread_num,
round);

}

if (thread_num == 0)
    cudaMemcpy(Dist, d_dist[0], (round/2) * B * n * sizeof(int),
cudaMemcpyDeviceToHost);
else if (thread_num == 1)
    cudaMemcpy(&Dist[(round/2) * B * n], d_dist[1] + (round/2) * B * n, (n
- (round/2) * B) * n * sizeof(int), cudaMemcpyDeviceToHost);
```

Experiment & Analysis

System Spec

使用hades來做實驗與測量。

Weak Scalability

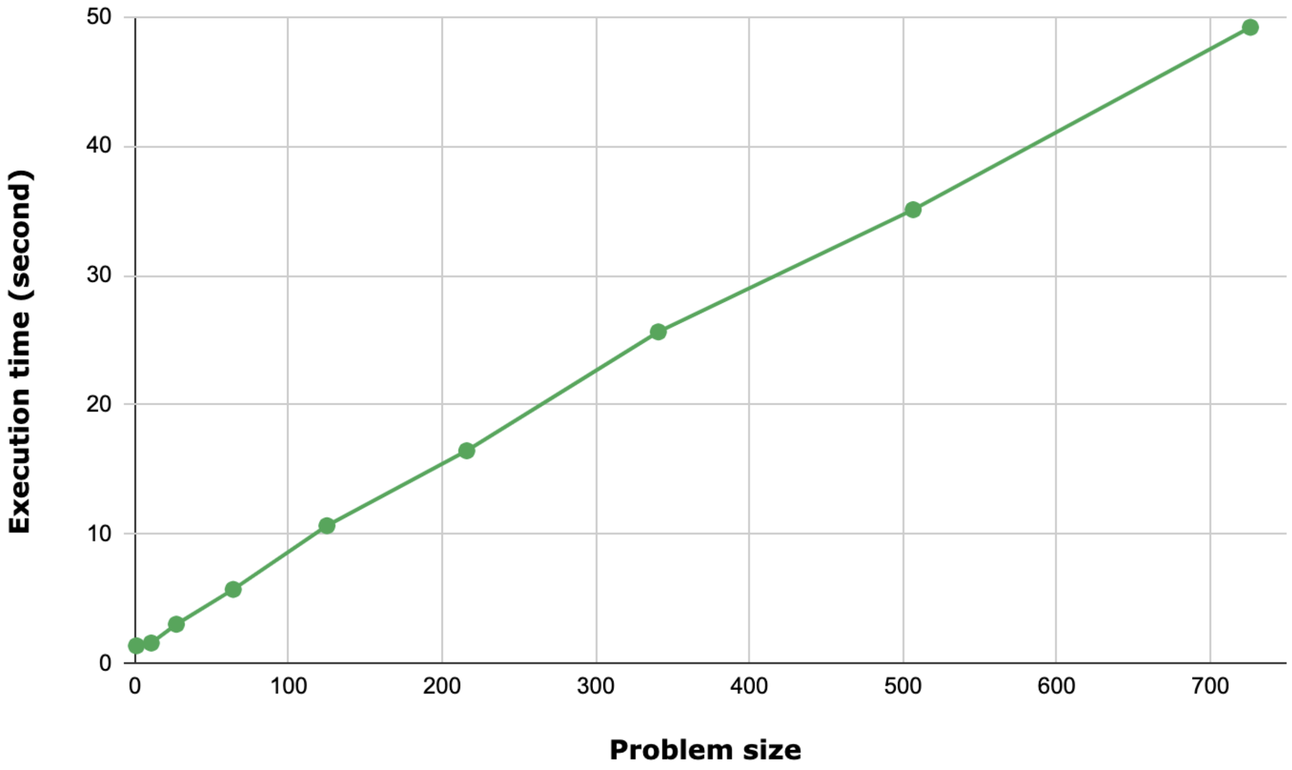
測量Execution time的方式是在main function的最一開始及最後return前分別取`std::chrono::steady_clock::now()`，最後再相減，即可獲得整體執行的時間。

```
int main(int argc, char* argv[])
{
    std::chrono::steady_clock::time_point t1 =
std::chrono::steady_clock::now();
    /* do calculation here */
    std::chrono::steady_clock::time_point t2 =
std::chrono::steady_clock::now();
    std::cout << "Execution took " <<
std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count() <<
"us.\n";
    return 0;
}
```

下表是將#vertices = 5000的problem size設成基準點1，其餘的再以`(#vertices/5000)^3`當成problem size。

testcases	# vertices	# edges	problem size	Execution time (second)
c04.1	5000	10723117	1(基準點)	1.391353
c05.1	11000	505586	10.648	1.604161
p15k1(hw4-1)	15000	5591272	27	3.055467
p20k1(hw4-1)	20000	264275	64	5.745622
p25k1(hw4-1)	25000	5780158	125	10.667851

testcases	# vertices	# edges	problem size	Execution time (second)
p30k1(hw4-1)	30000	3907489	216	16.460118
p35k1	34921	28054826	340.6826385	25.644699
c06.1	39857	4232291	506.5284076	35.084338
c07.1	44939	2418733	726.0394169	49.175928



從上圖可以看到execution time與problem size大約是linear的關係，problem size每放大為大約110倍，執行時間就會增加10倍。

Time Distribution

computing time、與memory copy time(H2D, D2H)、communication time都是透過`nvprof`來測量，其中computing time為phase1, phase2, 與phase3三者的時間總和，memory copy time是看Cuda memcpy HtoD和Cuda memcpy DtoH，communication time是看cudaMemcpyPeer的時間。而I/O time則是透過以下的方式測量：

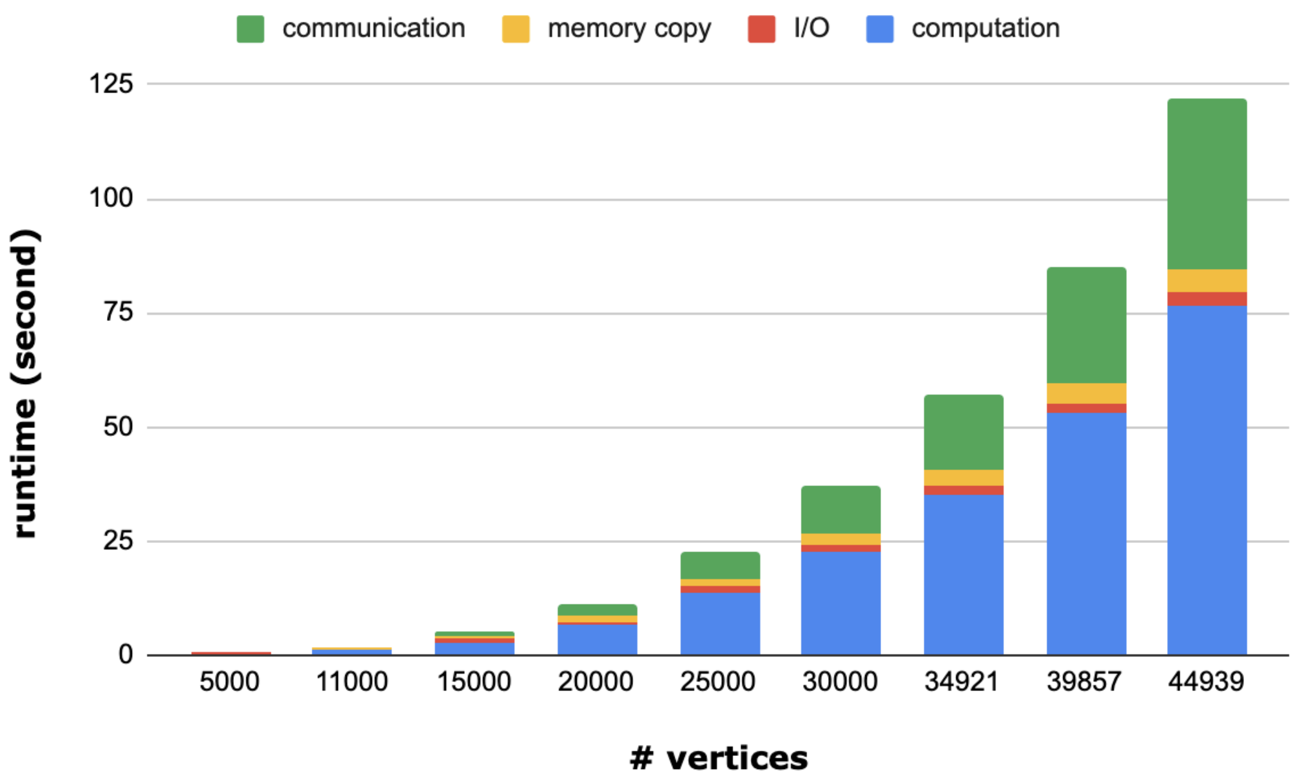
```
std::chrono::steady_clock::time_point t1 =
std::chrono::steady_clock::now();
/* doing I/O here */
std::chrono::steady_clock::time_point t2 =
std::chrono::steady_clock::now();
std::cout << "Reading(or writing) file took " <<
std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count() <<
"us.\n";
```

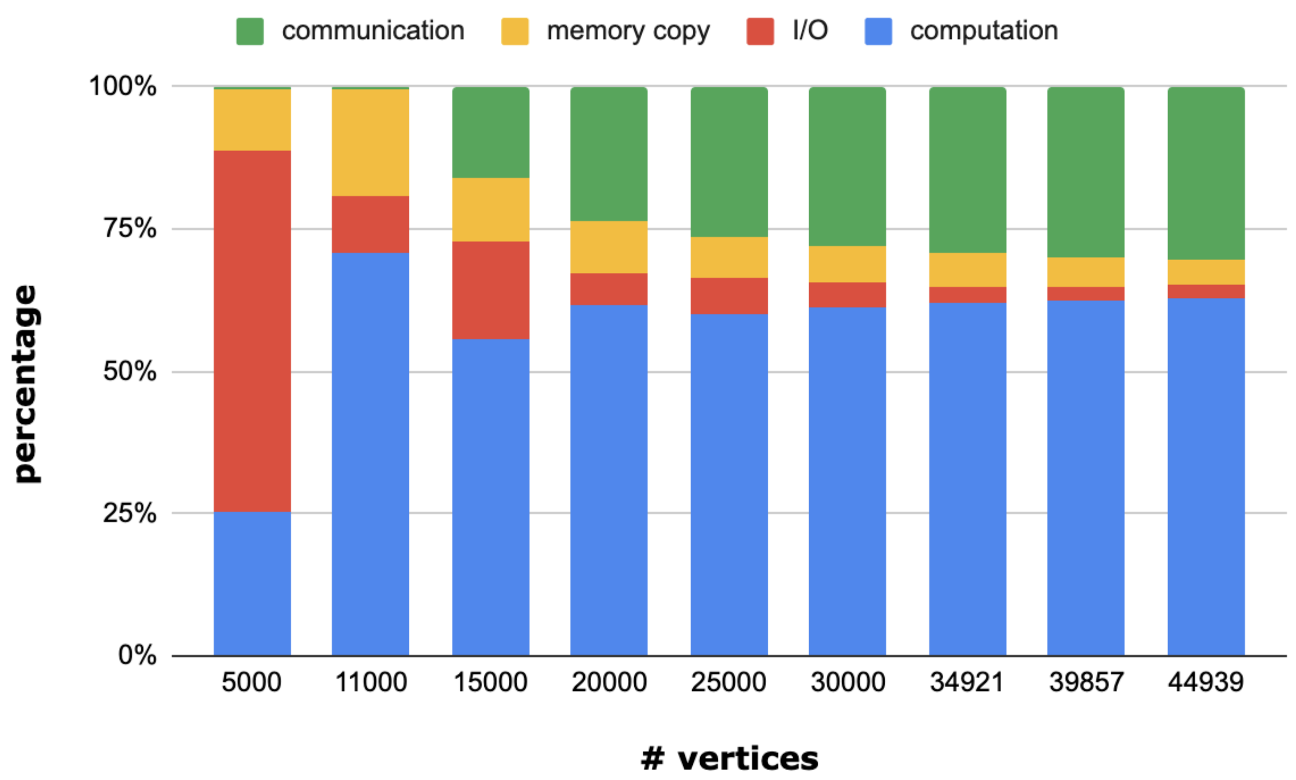
範例結果如下:

```
================44939================
==147599== NVPROF is profiling process 147599, command: ./hw4-2 /home/pp20/share/hw4-2/cases/c07.1 /dev/shm/out.out
n: 44939, m: 2418733
Reading file took 15248us.
Writing file took 2785133us.
==147599== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c07.1 /dev/shm/out.out
==147599== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   91.01%  74.6180s      1406  53.071ms  51.322ms  54.334ms  phase3(int, int, int, int*, int, int, int)
                    4.03%  3.30023s       704  4.6878ms  1.6746ms  1.34936s  [CUDA memcpy HtoD]
                    2.65%  2.17005s       704  3.0825ms  1.7378ms  611.73ms  [CUDA memcpy DtoH]
                    2.28%  1.86524s      1406  1.3266ms  1.2578ms  1.3698ms  phase2(int, int, int, int*, int)
                    0.04%  34.051ms      1406  24.218us  22.976us  24.960us  phase1(int, int, int, int, int, int, int, int*, int)
      API calls:   76.15%  37.1551s       702  52.928ms  59.430us  2.53311s  cudaMemcpyPeer
                   16.55%  8.07661s         4  2.01915s  681.16ms  3.16756s  cudaMemcpy
                    3.74%  1.82257s         1  1.82257s  1.82257s  1.82257s  cudaHostAlloc
                    3.01%  1.47081s         2  735.41ms  12.554ms  1.45826s  cudaMalloc
                    0.43%  209.81ms         1  209.81ms  209.81ms  209.81ms  cudaDeviceSetCacheConfig
                    0.12%  58.758ms      4218  13.930us  3.2000us  75.272us  cudaLaunchKernel
                    0.00%  642.34us         2  321.17us  319.69us  322.65us  cuDeviceTotalMem
                    0.00%  363.88us       202  1.8010us     146ns  77.621us  cuDeviceGetAttribute
                    0.00%  40.888us         2  20.444us  15.570us  25.318us  cuDeviceGetName
                    0.00%  14.241us         2  7.1200us  7.1040us  7.1370us  cudaSetDevice
                    0.00%  4.8350us         2  2.4170us  1.7530us  3.0820us  cuDeviceGetPCIBusId
                    0.00%  1.4980us         3     499ns     223ns     968ns  cuDeviceGetCount
                    0.00%  1.1500us         4     287ns     164ns     590ns  cuDeviceGet
                    0.00%     601ns         2     300ns     257ns     344ns  cuDeviceGetUuid
```

testcases	# vertices	# edges	input	output	communication
c04.1	5000	10723117	0.057633	0.302403	0.0028
c05.1	11000	505586	0.003990	0.163405	0.0058
p15k1(hw4-1)	15000	5591272	0.575220	0.312495	0.8236
p20k1(hw4-1)	20000	264275	0.029904	0.549940	2.5967
p25k1(hw4-1)	25000	5780158	0.592465	0.856178	6.0107
p30k1(hw4-1)	30000	3907489	0.403223	1.230616	10.390
p35k1	34921	28054826	0.121019	1.673196	16.689
c06.1	39857	4232291	0.028990	2.182926	25.517
c07.1	44939	2418733	0.015248	2.785133	37.155

testcases	# vertices	# edges	H2D	D2H	computation
c04.1	5000	10723117	0.0384	0.0235	0.14491
c05.1	11000	505586	0.1890	0.1209	1.17718
p15k1(hw4-1)	15000	5591272	0.3580	0.2320	2.90113
p20k1(hw4-1)	20000	264275	0.6433	0.4162	6.85874
p25k1(hw4-1)	25000	5780158	1.0105	0.6713	13.7704
p30k1(hw4-1)	30000	3907489	1.4548	0.9665	22.6893
p35k1	34921	28054826	1.9724	1.3095	35.3592
c06.1	39857	4232291	2.5817	1.7072	53.0892
c07.1	44939	2418733	3.3002	2.1700	76.5173





從上圖可以看到，communication在vertices數增加的過程中，佔比逐漸提升，到最後communication time、memory copy time、I/O time、computation time似乎達成了一種穩定的狀態，佔比最大的是computation time，大約62%左右，而communication time佔比則為第二大，大約為30%左右，也是花費了很大的一段時間。

Experience & conclusion

這次作業是使用2顆GPU，最一開始的時候想法很單純，想著兩個GPU個別做一半(上半&下半)，再把自己的運算結果傳到對方那，這樣的溝通成本實在太高，大約跟computing的時間差不多了，非常不划算，使用兩顆GPU反而比使用一顆GPU還要慢，因此發現苗頭不對，肯定不是這樣實作。中間一度很迷茫，想著別人的時間竟然差不多會是使用一個GPU的一半，腦門大開地懷疑該不會是一人做一半的round，然後再做merge之類的手法嗎(divide and conquer的想法)？但最後想不到merge那邊要怎麼做XD，所以就覺得應該不會是這種奇怪的想法。

正如同老師上課講的，GPU的計算很快，所以傳遞資料就會是bottleneck，這次的作業若單純讓gpu分開計算，然後將半個array傳給對方，就能發現communication真的佔據了非常大的一部份，因此就要減少溝通的次數或資料的大小。通過這次的作業，讓我更加了解blocked floyd warshall這個演算法、知道phase間的dependency，並且對cuda更加地熟悉，也學到不是越多GPU就能使效能變得更好，是個很值得花時間的一個作業。