Parallel Programming

Blocked All Pairs Shortest Path

Peter Huang

1、 Analysis

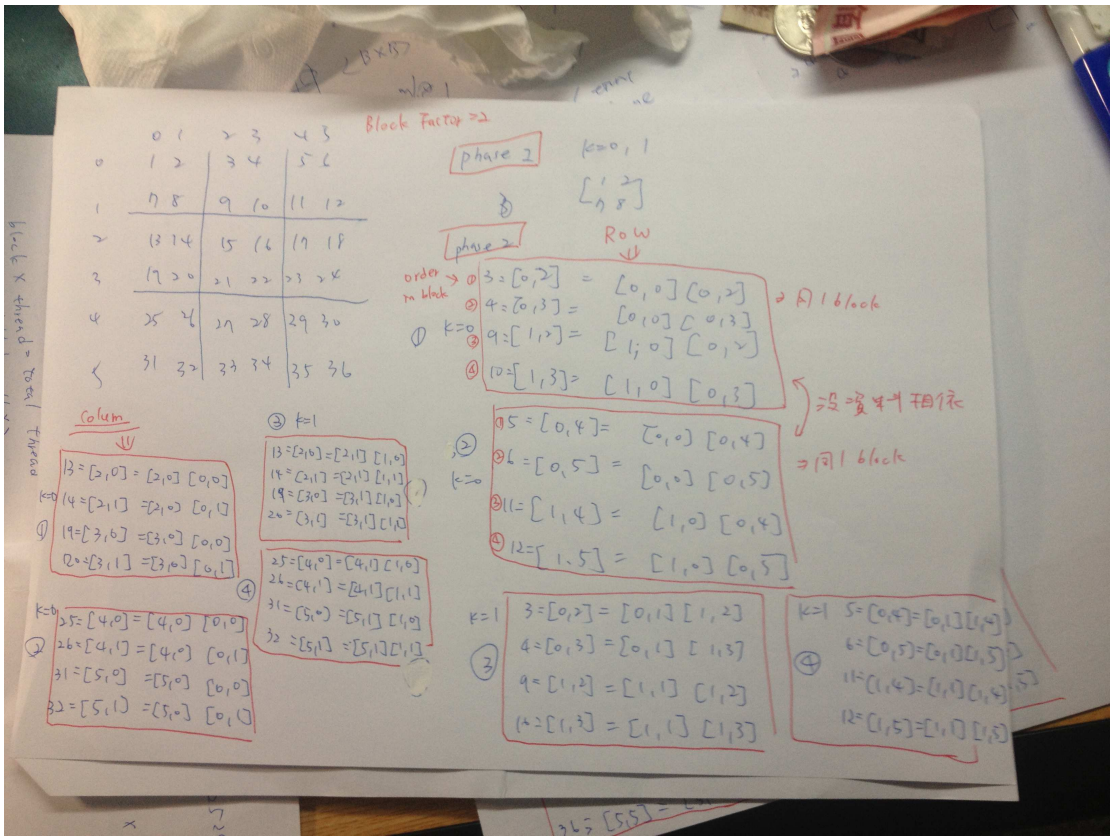
By draw one example, I can get following ideas and conclusions for the homework.

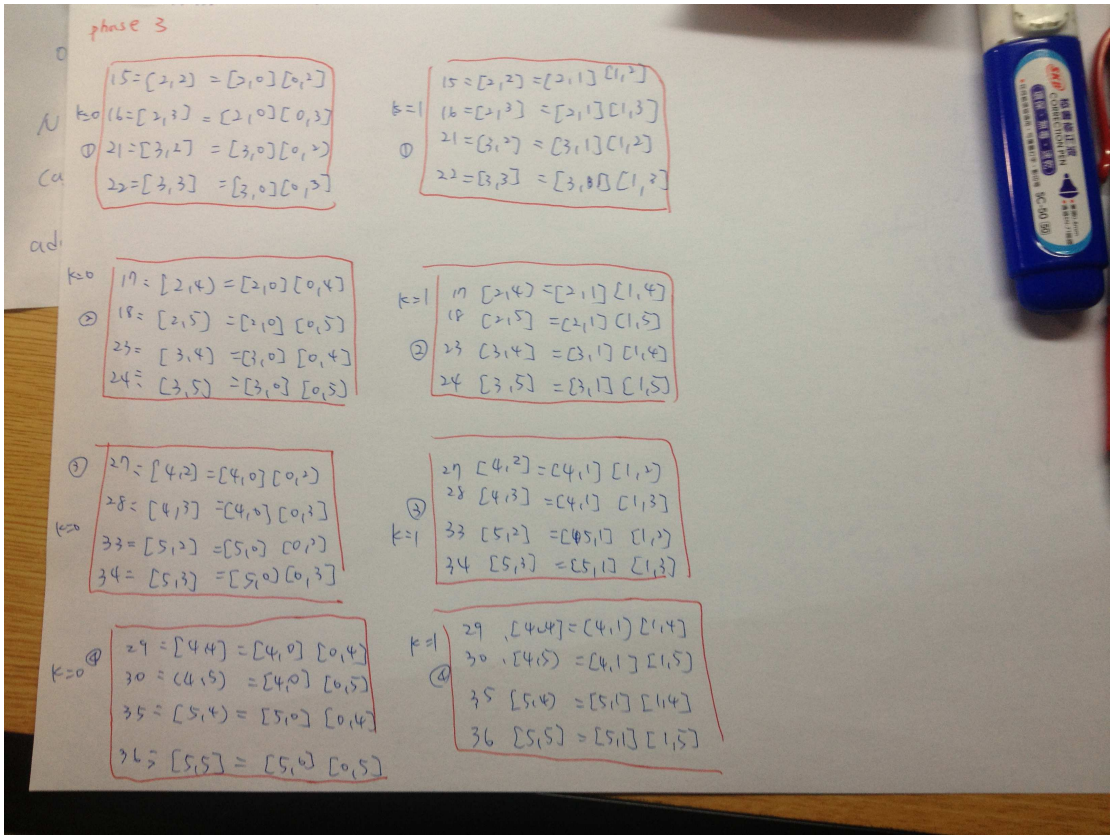
A. Inter-Phase has strong dependence like Phase 2 needs to calculate before Phase1 is done and Phase 3 needs to calculate before Phase2 is done.

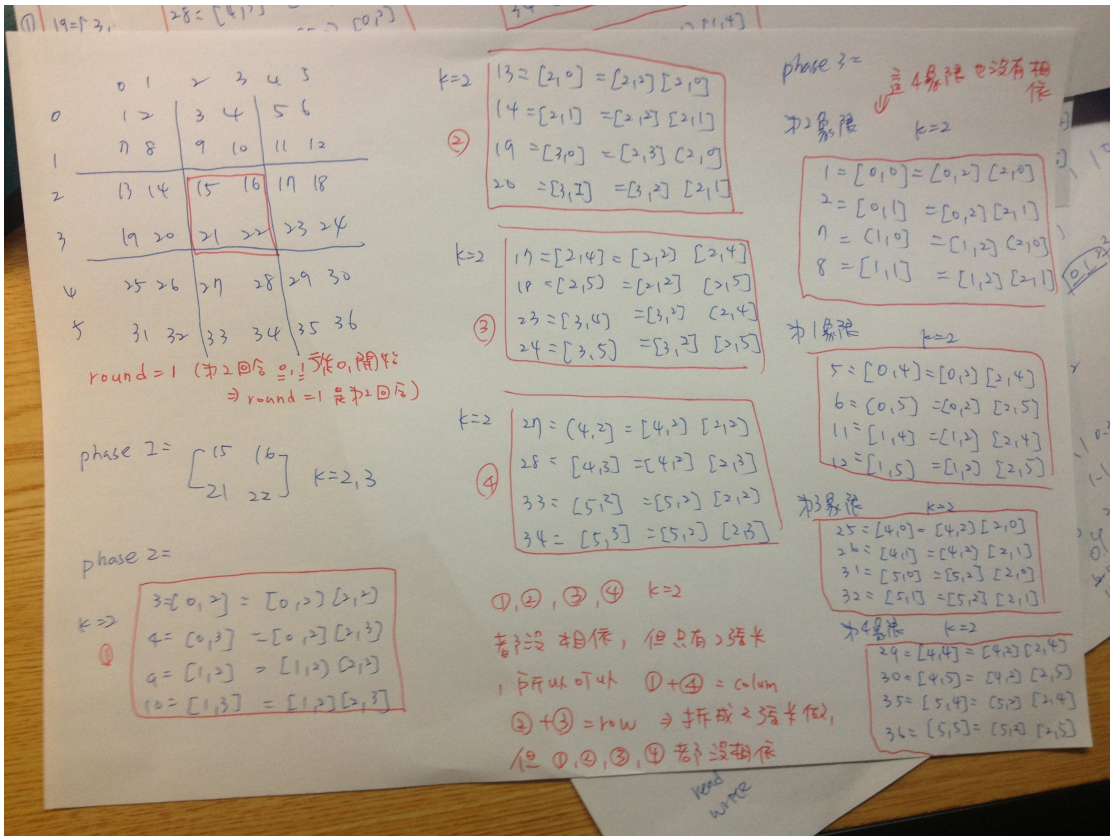
B. Each cal (calculation action) in Phase 1 or 3 are independent so that we can parallelized any calculation actions within Phase1-3.

C. Phase 3 dominated all calculation action, because it has the most amount of blocks between Phase1 – Phase3

D. All the calculation actions in Blocked APSP is equal with classic Floyd-Warshall algorithm. Just sequence of calculation action is different only.


Block Factor = 2
phase 2
phase 3
Row
Column
order in block
k=0, 1
k=0
k=1


phase 3
k=0
k=1



2、 Implementation

Single GPU:

A. How to divide data:

Handle out data by blocked APSP with 3 Phase

B. How to distribute data and configuration setting:

GridDim(Block_Height, Block_Width, 1)
BlockDim(Blocking_Factor, 1) ,
Optimal Blocking Factor:128

Because the GPU card in one SM which has 8 blocks, there are up to 1024 threads running. 1024 / 8 = 128 threads in each block, which can maximally utilize GPU card’s performance.

C. How to do blocked All Pairs Shortest Path in GPU kernel:

In kernel functions, because my BlockDim is one dimension, I need a for loop in kernel function to simulate the two dimension actions to calculated all points. However to know which Block in GPU do calculated action, I use X, Y coordinate as offset to do mapping. In block (0,0) will always handle out data of (X,Y) block like:

i = blockIdx.x+x,
j = blockIdx.y+y;

Multi GPU:

A. How to divide data:

Parallelized Phase3 for two GPU cards

Before divide data for parallelized computation, I analyze the computation time between phase1-3. I discover that phase3 dominated all computation action. In single GPU, Blocking Factor: 128 , All computation time is 14.341 sec when using in5 as input file. However, the phase 3 take 11.790 sec.

B. How to distribute data and configuration setting:

The setting of Multi GPU follows as Single GPU setting.

C. Multi GPU - implementation:

Using bit/char vector to records which value is be modified by GPU card and the bit/char vector size is N*N. When data has been modified when set 1 to the value. Then, when I need to map the data, the program scan the bit/char vector once, and outdate the value by the modified bit/char.

3、 Profiling Result

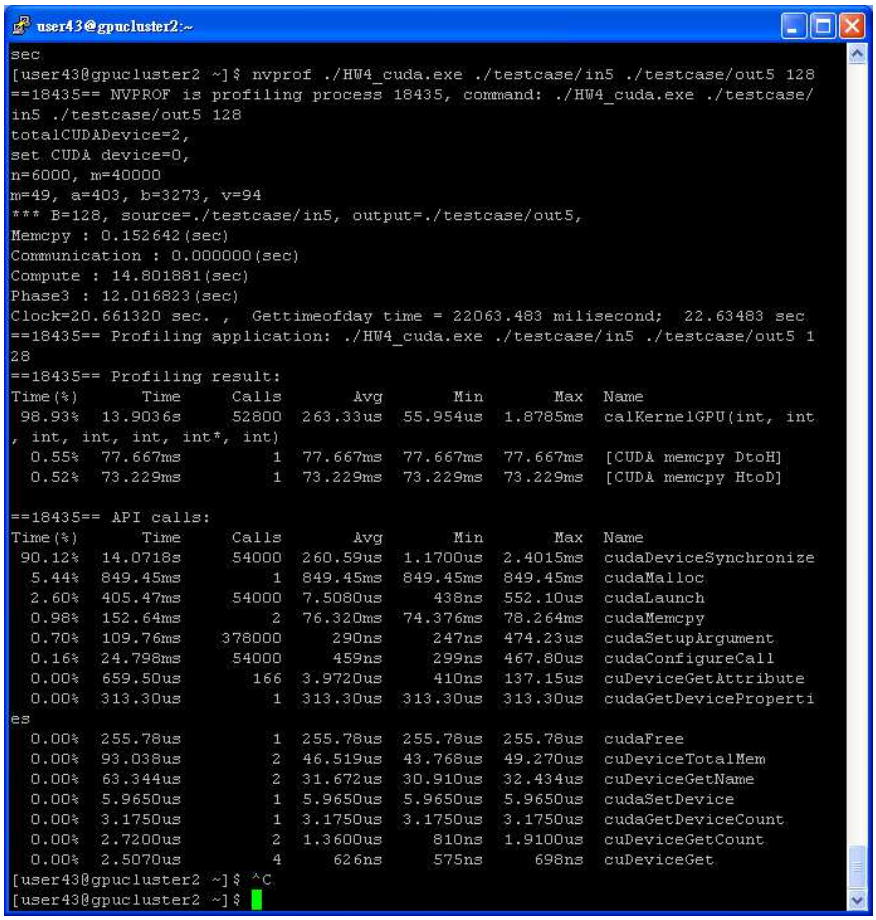
Why Blocking Factor=128 gets best performance:

BlockDim(Blocking_Factor, 1)

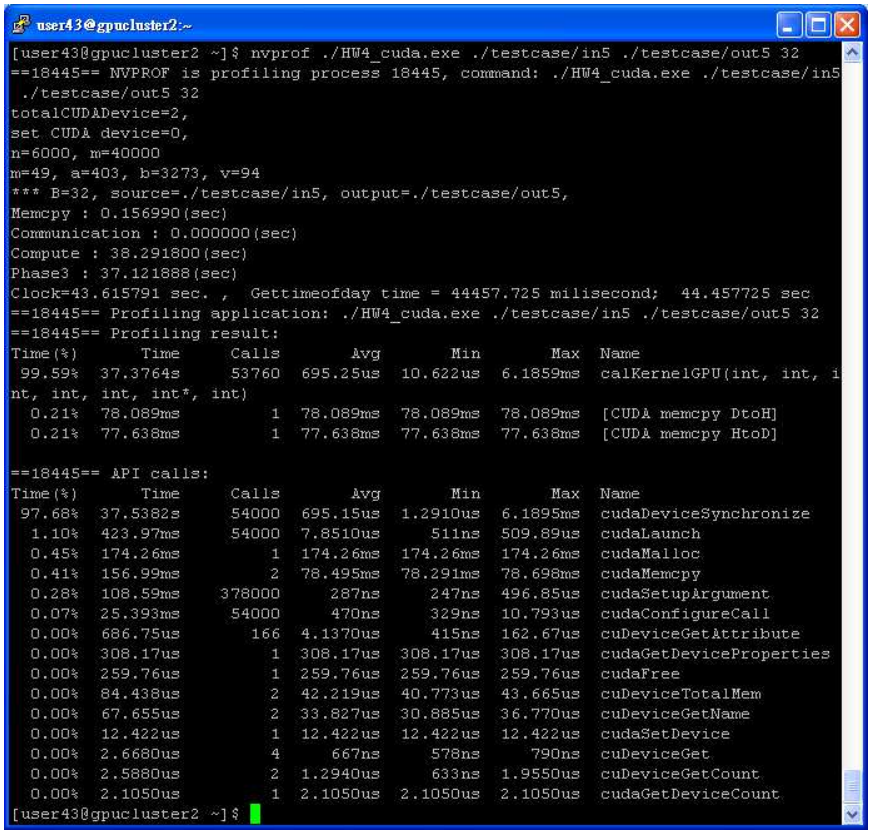
Because the GPU card in one SM which has 8 blocks, there are up to 1024 threads running. 1024 / 8 = 128 threads in each block, which can maximally utilize GPU card's performance. If I use more threads Like Blocking Factor=256. For on SM, there are 256*8= 2048 threads in One SM, which means there are 1024 threads will idle and needs to consuming context switch time. (Ps. because running threading available at the same time is up to 1024 threads.)

Profiling in gpucluster2 server using Main device:0.

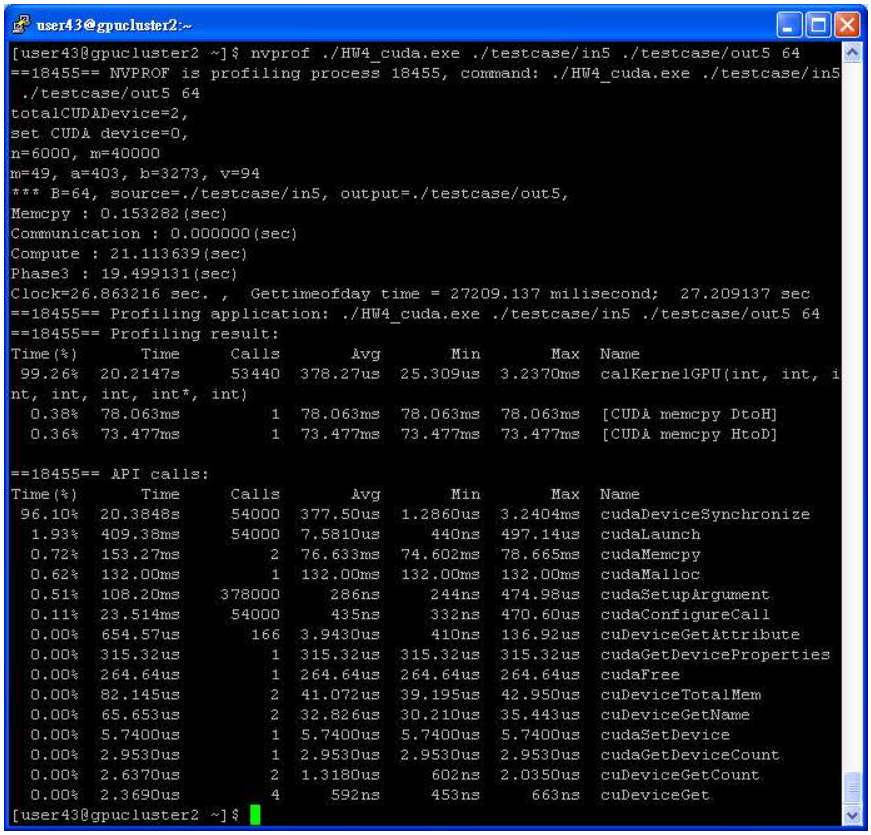
Block factor:128 (Best Performance in GPU)



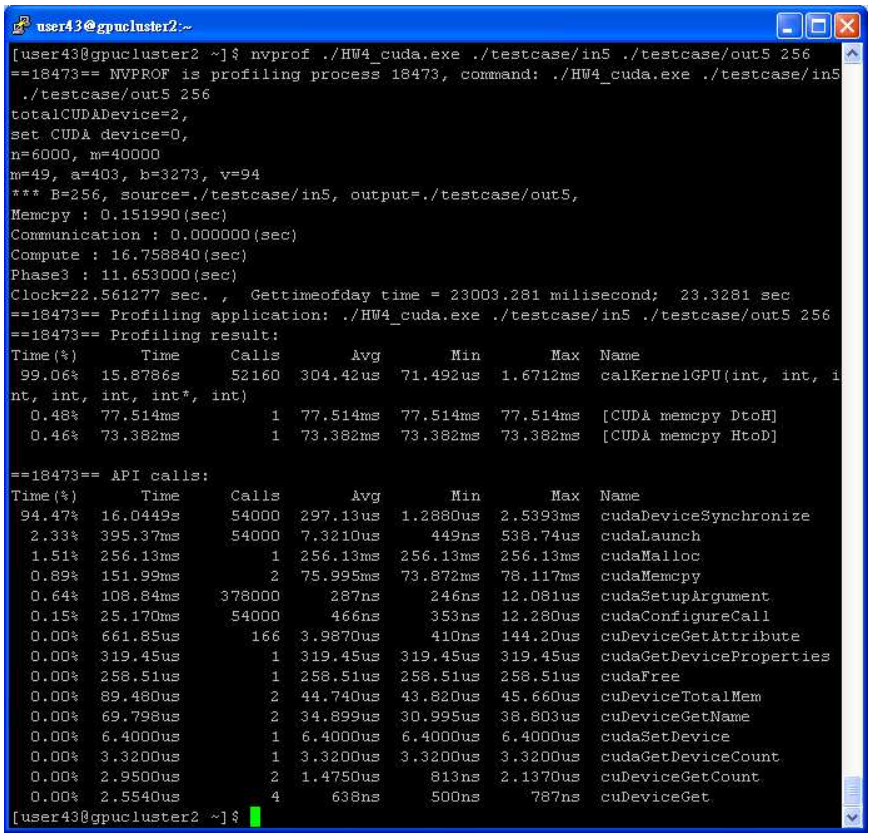
Block factor: 32



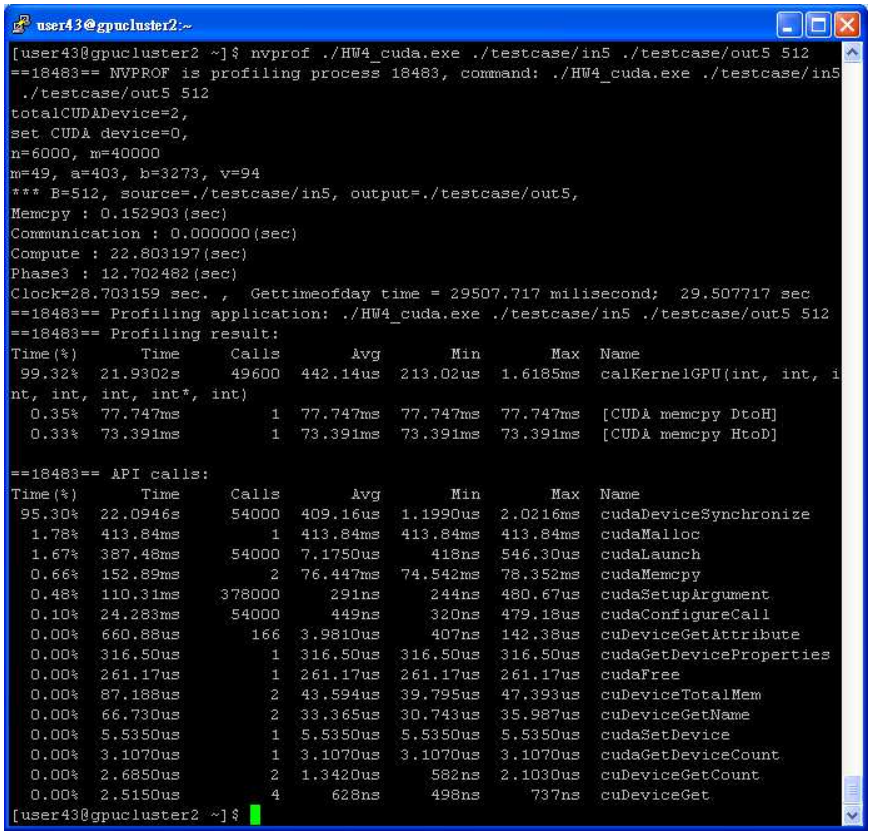
Block factor: 64



Blocking factor:256



Blocking factor:512

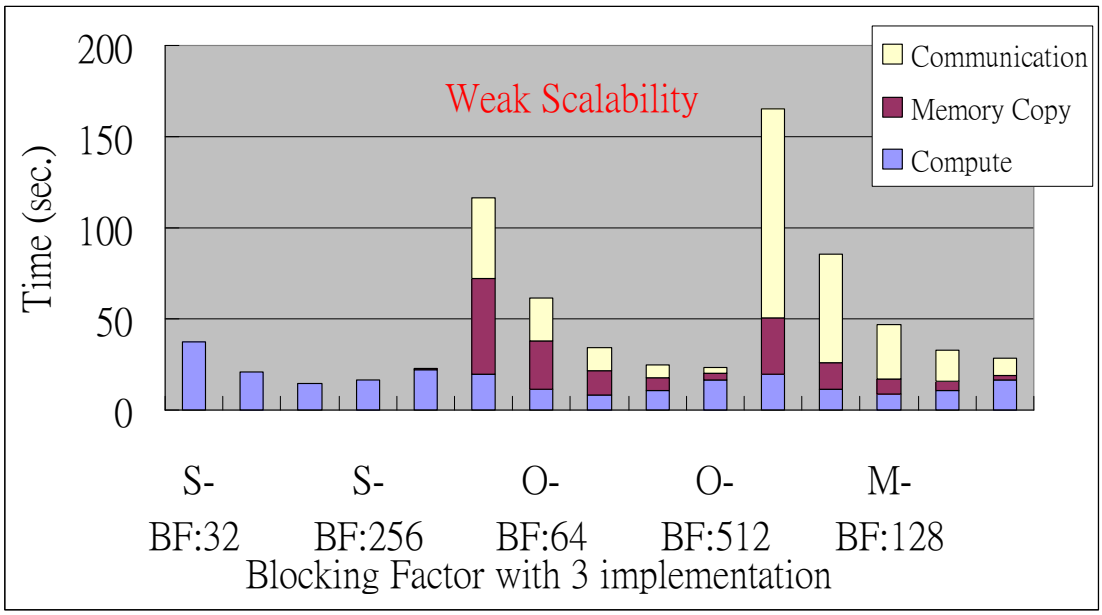


4、 Experiment And Analysis

1、 System Spec

Running in gpucluster2 with all cases by using in5 file.

2、 Weak scalability & Time distribution



Single GPU (S)					
BF=Blocking Factor	**BF:32**	**BF:64**	**BF:128**	**BF:256**	**BF:512**
Memory Copy	0.139685	0.139703	0.139994	0.283836	0.139745
Communication	0	0	0	0	0
Compute	37.482757	20.601837	14.341786	16.361533	22.38672
phase3	36.558651	19.223927	11.790387	11.461537	12.514091

Multi GPU- OpenMP (O)					
BF=Blocking Factor	**BF:32**	**BF:64**	**BF:128**	**BF:256**	**BF:512**
Memcpy	52.6527	26.639823	13.210143	6.934426	3.748701
Communication	44.215727	23.427156	12.749118	7.141067	3.508327
Compute	19.539619	11.20195	8.528045	10.623591	16.411127
Total	116.408	61.26893	34.48731	24.69908	23.66816

Mulit GPU- MPI (M)					
BF=Blocking Factor	**BF:32**	**BF:64**	**BF:128**	**BF:256**	**BF:512**
Memcpy	31.185486	14.956198	8.425116	4.76195	2.315235
Communication	114.608579	59.551292	30.047409	17.700295	9.605734
Compute	19.488218	11.186668	8.599019	10.744805	16.392509
Total	165.2823	85.694158	47.071544	33.20705	28.313478

Analysis:

In single GPU

The blocking factor is 128 gets best performance as we predicted previous section. And the phase 3 are dominated on blocked all pairs shortest path.

GridDim(Block_Height, Block_Width, 1)
BlockDim(Blocking_Factor, 1) ,
Optimal Blocking Factor:128

Because the GPU card in one SM which has 8 blocks, there are up to 1024 threads running. 1024 / 8 = 128 threads in each block, which can maximally utilize GPU card's performance.

In Multi-GPU

A. In OpenMP & MPI there are the almost same computation time in GPU, because the calculated pointed in the programs are the same and with the same parallelized action in phase 3.

B. Total Time = Compute Time + (2 cards' communication time, data separation and merger action)

In OpenMP there are almost double memory copy time, because in OpenMP I records all memory copy action which happens in One processor. However, in MPI there are two processors, so I just records memory copy in rank=0's processor.

This is why we get almost two double memory copy time in openMP.

In communication time the MPI take a lot time than OpenMP, because MPI mechanism needs to use send and receive to communicate with another GPU card and OpenMP is shared memory, so all the communication happens between processors only. Hence, the MPI take a lot of communication time.

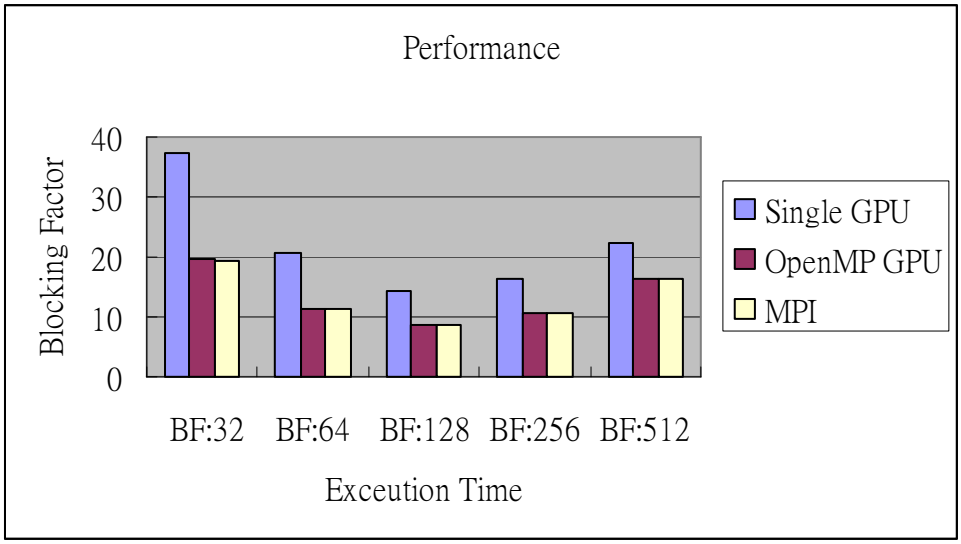
However, based on the result, if we want to get better performance, we need to trade off Compute Time and 2 cards' communication time, data separation and merger action. Although blocking factor has best performance in computation in GPU card, it take too much on data's separated, merged and communicated action. So In multi-GPU card if we need to decide how to divide our data by these evaluation. Finally, we can see if we use BF=512, it take a little time on data handling and also its computation time in GPU are acceptable. And, then we get best performance in my experiment between BF=32 ~ BF=512.

5、 Blocking Factor

Test file :in5 n=6000

Because blocked APSP's calculated action is equal with classic floyd-warshall, we can evaluate that calculation action in kernel mode is the same as class floyd-warshall. O(N^3)

A. Execution Time Performance



Excution time (sec.)	BF:32	BF:64	BF:128	BF:256	BF:512
Single GPU	37.482757	20.601837	14.341786	16.361533	22.38672
OpenMP GPU	19.539619	11.20195	8.528045	10.623591	16.411127
MPI	19.488218	11.186668	8.599019	10.744805	16.392509

Analysis

In single GPU there are bad performance in computation, because we only use one card to calculate all data. However, in multiple GPU we always get better performance on computation because there are more cards to parallelize to calculate data.

B.B.Performance (GFLOPs)

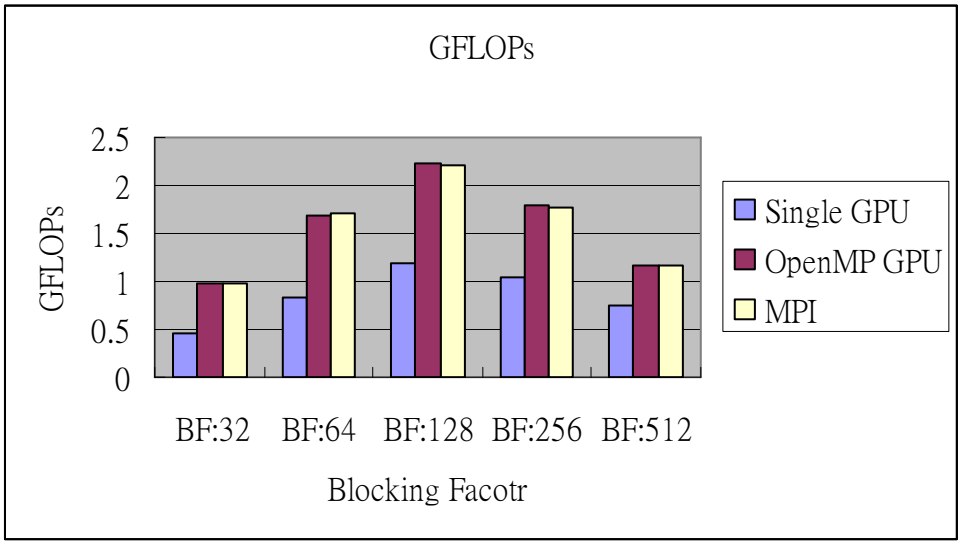
Single GPU GFlops= O (n^3*operation) = O(n^3*17) = 6000*6000*6000*17 (GPU kernel with 17 operation)

Multi GPU GFlops= O (n^3*operation) = 6000*6000*6000*19

1data needs how much instruction	BF:32	BF:64	BF:128	BF:256	BF:512
Single GPU	17/37.48	17/20.60	17/14.34	17/16.36	17/22.38
OpenMP GPU	19/19.53	19/11.20	19/8.52	19/10.62	19/16.41

MPI	19/19.48	19/11.18	19/8.59	19/10.74	19/16.39

Gflops	BF:32	BF:64	BF:128	BF:256	BF:512
Single GPU	0.45357524	0.825242718	1.185495119	1.039119804	0.759606792
OpenMP GPU	0.972862263	1.696428571	2.230046948	1.789077213	1.157830591
MPI	0.975359343	1.699463327	2.211874272	1.769087523	1.159243441



C. Bandwidth [GB/sec]

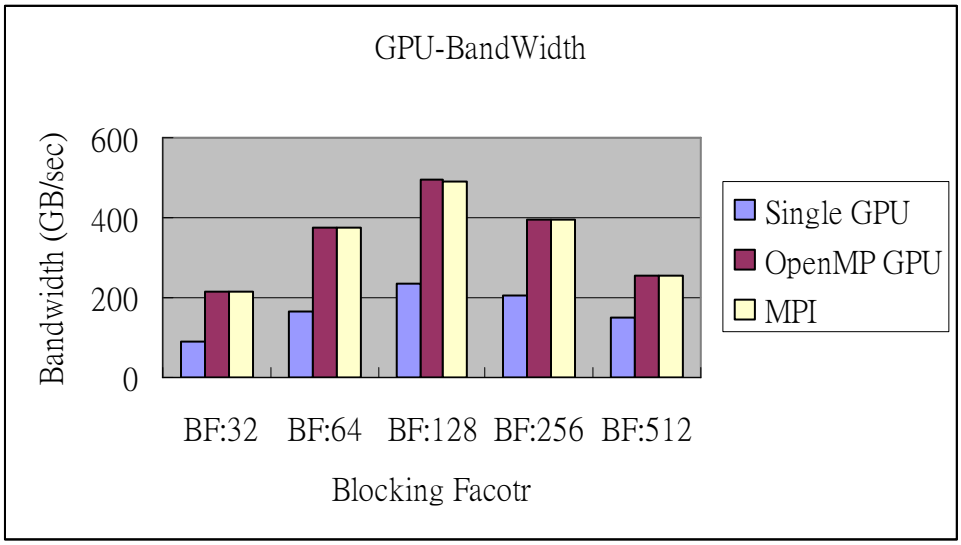
Single GPU bandwidth = O(n^3*4) (Ps. 3 input data and 1 write back data) = 6000*6000*6000*4*4 byte (PS. int = 4 byte) = 3375 GB / execution time

Single GPU	BF:32	BF:64	BF:128	BF:256	BF:512
GB/sec	3375	3375	3375	3375	3375
GPU execution time	37.48	20.60	14.34	16.36	22.38
Result	90.04	163.82	235.32	206.27	150.75

Multi GPU bandwidth = O(n^3*5) (Ps. 3 input data and 2 write back data) = 6000*6000*6000*5*4 byte = 4219 GB / execution time

OpenMP GPU	BF:32	BF:64	BF:128	BF:256	BF:512
GB/sec	4219	4219	4219	4219	4219
GPU execution time	19.53	11.20	8.52	10.62	16.41
Result	215.92	376.63	494.72	397.13	257.08

MPI	BF:32	BF:64	BF:128	BF:256	BF:512
GB/sec	4219	4219	4219	4219	4219
GPU execution time	19.48	11.18	8.59	10.74	16.39
Result	216.48	377.14	490.63	392.65	257.37



GB/sec	BF:32	BF:64	BF:128	BF:256	BF:512
Single GPU	90.04	163.82	235.32	206.27	150.75
OpenMP GPU	215.92	376.63	494.72	397.13	257.08
MPI	216.48	377.14	490.63	392.65	257.37

Analysis

Based on Gflops and bandwidth, we can know all programs are limited by calculation capability or bandwidth capability. When the program ups to the limit of bandwidth, I know I need to improve the access of memory to achieve better performance. However, if

we find that our program cannot maximally utilize our calculation, which means the GPU card's performance is not be utilized, because the amount of data is not large enough to utilized all our GPU card performance.

6、 Compare 3 implementation

Sequential Code with Blocking Factor=128 takes 947.07 sec in in5.

Single Cuda with Blocking Factor=128 takes 14.481 sec in in5.

OpenMP with Blocking Factor=128 takes 34.487 sec in in5 and it takes 26 sec in Memory Copy and Communication.

MPI with Blocking Factor=128 takes 47.071 sec in in5 and it takes 38 sec in Memory Copy and Communication.

Speedup

Single cuda:	947.07 / 14.481= 65.3
OpenMp:	947.07 / 34.487= 27.4
MPI:	947.07 / 47.071= 20.1

Performance: Single cuda > OpenMP > MPI

Analysis:

OpenMP: the computation gets better in 2 GPU cards than in 1card. If we only compare the communication time which is reduced from 14.341 to 8.528. However, because there are two cards, my program get more cost in memory copy and communication, because my programs use D2H and H2D not D2D. And, also I only copy memory for main programs. Finally, I get more time in Memory copy and communication.

MPI: the computation also gets better in 2 GPU cards than in 1 card. If we only compare the communication time which is reduced from 14.341 to 8.599. However, there are more communication time between processors and MPI's communication take most of time, which is why MPI get poor performance, when compared with OpenMP and single cuda.

Improvement:

In OpenMp the 2 cards' communications uses D2H and then call H2D in another device. If we can D2D in our 2 GPU card' devices, then the time for Memory Copy and Communication can be reduced. Then, get better performance.

7、 Experience/Conclusion

a、 What I have learned from this assignment

1、 How to divide data and assign data into SIMD (single instruction and multiple data). For example: when there are nxn data, in cuda we can parallelized the nxn action in kernel function, and these nxn data at least take n^2 computation in sequential code. However, in cuda we only need take one time in kernel function with different threads, which largely saves computation time.

2、 Learn some cuda's conception and analysis data's dependence and learn how design GridDim and BlockDim to cuda device. And how to get correct computation in cuda kernel function.

b、 What difficulty did I encounter when

implementing this assignment

1、 Before programming cuda program, I get many bugs in dividing data. And don’t know how to parallelized my codes. However, finally, I try to simulate cuda action with OpenMP. Try to validate my programs in CPU version and it is good way to validate all my idea before implementing cuda code.

2、 When bugging in cuda, I use printf and compared the right data in sequential code with different phase to find out which part my calculation is wrong or which part makes the bugs when I try to communication data in two cuda device.