CS542200 Parallel Programming
HW 4-2: Blocked APSP (Multi-GPU)

106030009 葉蓁

1. Implementation

(1) Configuration:

I re-use the configuration in my hw4-1.

Configuration:

Blocking Factor : B = 32

 # blocks: Phase1 : 1.

Phase2 & 3 : 2D blocks: (height, width)

 # threads : 2-D threads (B,B), total 32*32 = 1024 threads per block.

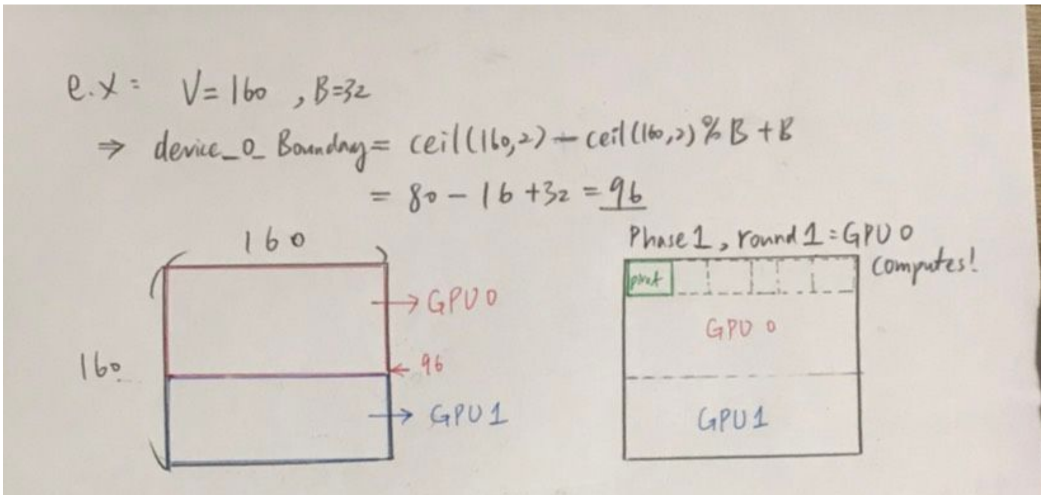
Why ? : For each B*B block, let as many cuda threads access concurrently as possible to exploit SIMD structure. For convenience, I use 2d cuda threads and blocks to map threads with Dist[i][j].

Why do I use B = 32 ? Since maximum threads per block is 1024 = 32*32. This is why I use B=32 and #threads (32, 32, 1) per block.

(2) How do you divide your data?

Final version:

A. Initially, partition whole n*n matrix into upper and lower part. (But still copy n*n from host to devices)



B. In each phase:

The device only computes data that are within its area.

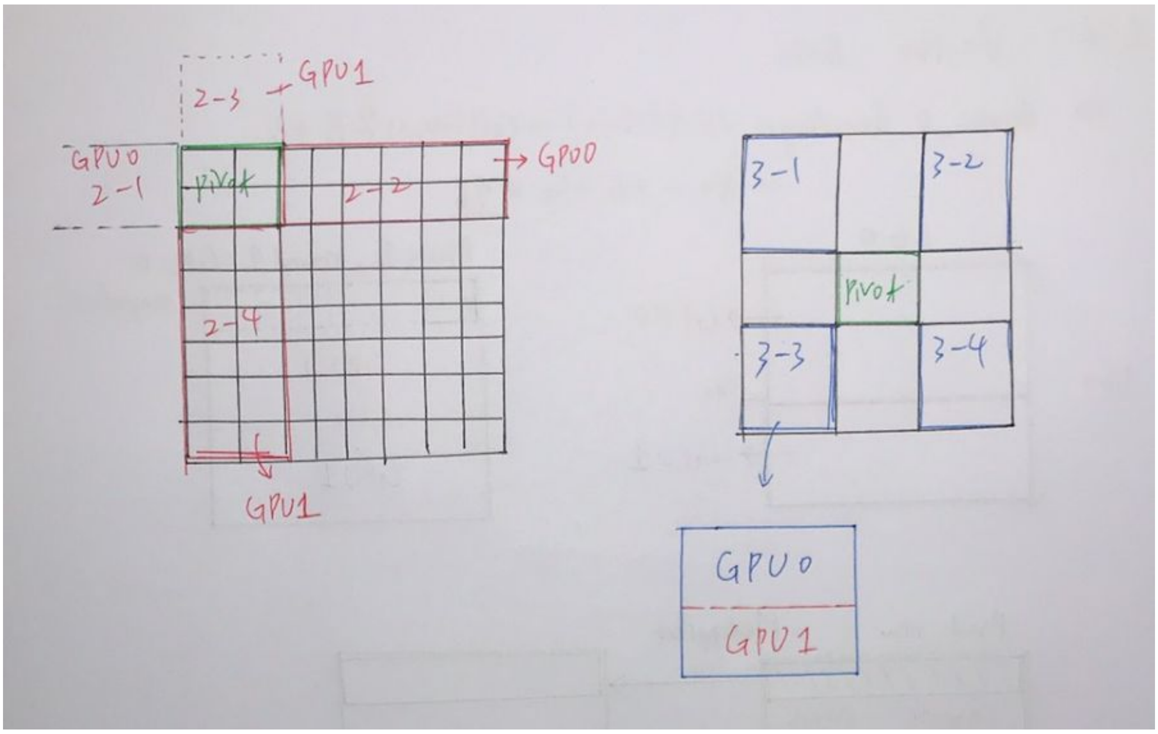
Old version:

For 3 phases, I modify seq.cc to hw4-2.cu.

Phase 1: I use only GPU 0.

Phase 2: I assign 2-1, 2-2 to GPU0 and 2-3, 2-4 to GPU 1.

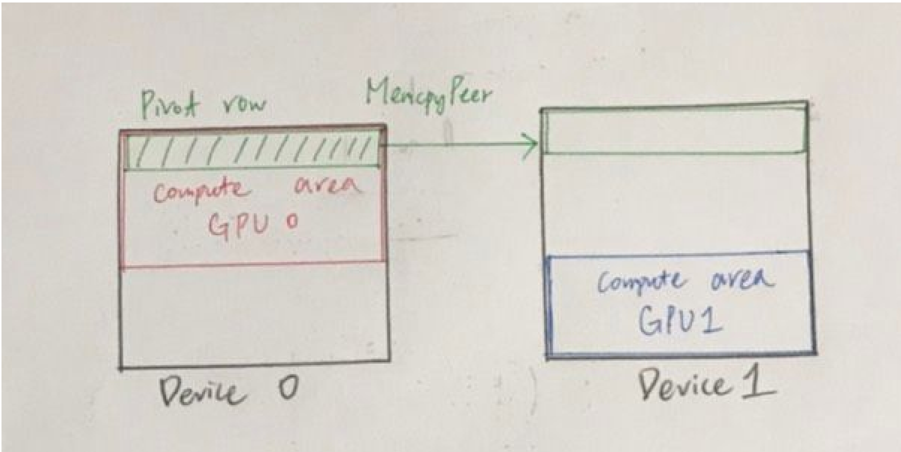
Phase 3: for each sub-phase (3-1 ~ 3-4), I **divide the sub-phase in the vertical direction into two parts**. **GPU 0** computes the **upper** part and **GPU 1** computes the **lower** part.



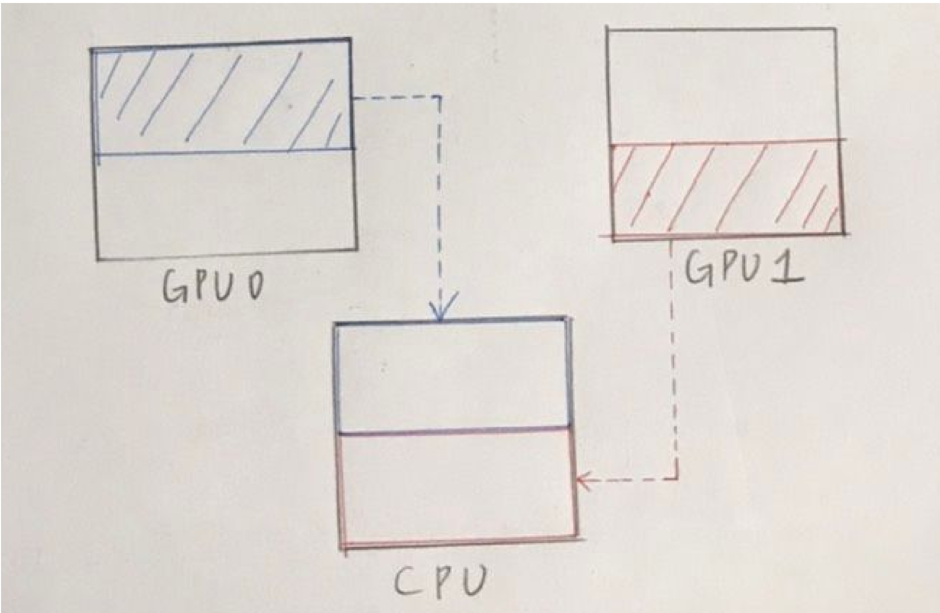
(3) How do you implement the communication?

Final version:

a. Initial memory copy: copy the n*n Dist from host to two devices using two omp threads.
b. In each round, after phase 1, the device which computes the pivot block should copy the whole pivot row to the other device.



c. In phase 2 and phase 3: each device only computes data within its area. (No memory copy here.)
d. After 'round' rounds, each device copies its data to the host independently. (in parallel)




```
c04.1    1.64   accepted
c01.1    0.89   accepted
c03.1    0.99   accepted
c02.1    0.73   accepted
c05.1    4.42   accepted
p31k1   70.00   time limit exceeded:  {timeout}
c07.1   80.01   time limit exceeded:  {timeout}
c06.1   60.00   time limit exceeded:  {timeout}
```

Old Version 2:

Phase1 & Phase 2 : GPU 0 computes.

Phase 3: **GPU 0** computes **upper** data, **GPU 1** computes **lower** data.

Then cudaMemcpyPeerAsync() to synchronize two devices.

```
ptxas info    : Used 16 registers, 12288 bytes smem, 348 bytes c
make: Leaving directory '/home/pp20/pp20s35/.judge.525178887'
c04.1     7.03   time limit exceeded:  {timeout}
c02.1     0.42   accepted
c01.1     0.32   accepted
c03.1     0.52   accepted
c05.1    18.00   time limit exceeded:  {timeout}
c06.1    60.00   time limit exceeded:  {timeout}
c07.1    80.00   time limit exceeded:  {timeout}
p31k1    70.00   time limit exceeded:  {timeout}
```

Old Version1:

For each round, after each phase, I let the two devices synchronize device_dist[] they have on their individual per-device global memory.

Phase 1: just GPU 0 copy the pivot block to GPU1.

Phase 2: GPU 0 copy 2-1, 2-2 Blocks to GPU 1.

GPU1 copy 2-3, 2-4 Blocks to GPU 0.

Phase 3: GPU 0 copy device_dist it computes to GPU1 and GPU 1 done the same thing.

```
em[0]
make: Leaving directory '/home/pp20/pp20s35/.judge.954353767'
c01.1     0.57   accepted
c02.1     0.37   accepted
c03.1     1.22   accepted
c05.1    18.28   time limit exceeded:  {timeout}
c04.1    15.30   time limit exceeded:  {timeout}
c06.1    67.81   time limit exceeded:  {timeout}
c07.1    80.18   time limit exceeded:  {timeout}
```

Description of implementation:

1. Include libraries and declare functions and important parameters.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include <cuda.h>
#include <cuda_profiler_api.h>

// CUDA runtime
#include <cuda_runtime.h>

#include <omp.h>


#include <time.h>
#define TIME

#define CUDA_NVPROF

// #define DEBUG_DIST
// #define DEBUG_DEVICE_DIST
// #define DEBUG_DEVICE_DIST1
// #define DEBUG_PHASE1
// #define DEBUG_PHASE2
// #define DEBUG_PHASE3
// #define CHECK_CORRECTNESS
```

```
const int BLOCKING_FACTOR = 32; // 32, 16, 8, 4, 2

const int INF = ((1 << 30) - 1);
// Global var stored in Data Section.
// const int V = 40010;
void input(char* inFileName);
void output(char* outFileName);

void print_ans(int num_V, char* ans_file);
void print_Dist(int num_V);

void block_FW(int B);
// void block_FW_small_n(int B);
void block_FW_MultiGPU_Old(int B);
void block_FW_MultiGPU(int B);
int ceil(int a, int b); // min num that >= a/b
                // floor: max num <= a/b
int floor(int a, int b);

__device__ inline int Addr(int matrixIdx, int i, int j, int N){
    return( N*N*matrixIdx + i*N + j);
}
```

2. In main function: I divide input cases into two classes.

class 1 : small n (n < B)

class 2 : large n (n > B)

For class 1, I use my older version since the total time is similar.

For class 2, I use my final version implementation.

```
    // Note: Since B*B threads, maximum
    int B;
    B = BLOCKING_FACTOR;

    if(n < B){
        block_FW_MultiGPU_Old(B);
    }
    else{
        block_FW_MultiGPU(B);
    }
```

3. In block_FW_MultiGPU(B) :

First Step : create two omp threads. Each thread sets up a device and allocates memory (device_dist, device_dist1) on it and copies memory from host (CPU) to device individually.

```
    #pragma omp parallel num_threads(NUM_THREAD) //reduction(+:pixels)
    {
        int omp_id, omp_thread_num;
        omp_id = omp_get_thread_num();
        omp_thread_num = omp_get_num_threads();

        if(omp_id==0){
            cudaSetDevice(0);
            cudaDeviceCanAccessPeer ( &canGPU0AccessGPU1, device_0, device_1 );
            if(canGPU0AccessGPU1==1){
                printf("Can 0 access 1? %d\n",canGPU0AccessGPU1);

                cudaDeviceEnablePeerAccess ( device_1, cudaEnablePeerAccess_Flags );
                cudaMalloc(&device_Dist,   n * n* sizeof(unsigned int));

                #ifdef TIME
                    cudaEventRecord(Commstart);
                #endif
                cudaMemcpyAsync(device_Dist,   Dist, n* n*sizeof(unsigned int), cudaMemcpyHostToDevice);

                // cudaMemcpyAsync(device_Dist,   Dist, n*device_0_Boundary*sizeof(unsigned int), cudaMemcpy
                printf("omp t%d allocate & copy gpu 0\n",omp_id);
            }
            else{
                printf("Error, gpu 0 cannot directly access gpu 1\n");
                // return 2;
            }
        }
```

Device 1 :

```
            }
        }
        else{
            cudaSetDevice(1);
            cudaDeviceCanAccessPeer ( &canGPU1AccessGPU0, device_1, device_0 );
            if(canGPU1AccessGPU0==1){
                printf("Can 1 access 0? %d\n",canGPU1AccessGPU0);

                cudaDeviceEnablePeerAccess ( device_0, cudaEnablePeerAccess_Flags );
                // cudaGetDevice(&cur_device_number);
                cudaMalloc(&device_Dist_1, n * n* sizeof(unsigned int));
                cudaMemcpyAsync(device_Dist_1,   Dist, n* n*sizeof(unsigned int), cudaMemcpyHostToDevice);
                // cudaMemcpyAsync(device_Dist_1+device_0_Boundary*n, Dist+device_0_Boundary*n, ( n*n -n*de
                printf("omp t%d allocate & copy gpu 1\n",omp_id);
            }
            else{
                printf("Error, gpu 1 cannot directly access gpu 0\n");
                // return 2;
            }
```

Then, for each phase, there is some modification.

Phase 1: only the device which owns the pivot blocks should compute. After it finishes computation, it should send the pivot row to the other device.

```
    if(r*B < device_0_Boundary) { // Device 0 do pivot
        // printf("Pivot at GPU 0!\n");
        cudaSetDevice(0);
        cal<<< 1, num_threads , sizeof(int)*B*B>>> (device_0_Boundary, device_Dist, n, B, r,   r, r);

        #ifdef TIME
            cudaEventRecord(Commstart);
        #endif
        // Copy WHOLE pivot ROW to the other device.
        for(int i= r*B; i<(r+1)*B && i<n; i++)
            cudaMemcpyPeer(device_Dist_1+i*n,1, device_Dist+i*n,0 , n*sizeof(unsigned int));

        #ifdef TIME
            float Commtime_Phase1;
            cudaEventRecord(Commstop);
            cudaEventSynchronize(Commstop); // WAIT until 'stop' complete.
            cudaEventElapsedTime(&Commtime_Phase1, Commstart, Commstop);
            // printf("Phase1 mem copy took %.8f seconds\n",Commtime_Phase1/1000);
            Total_comm_time += Commtime_Phase1;
        #endif


    }
```

```
    }
    else{       // Device 1 do then copy to the other.
        cudaSetDevice(1);
        // printf("Pivot at GPU 1!\n");
        cal_1<<< 1, num_threads , sizeof(int)*B*B>>> (device_0_Boundary, device_Dist_1, n, B, r,   r, r);
        // Copy pivot ROW to the other device.
        #ifdef TIME
            cudaEventRecord(Commstart_device_1);
        #endif
        for(int i= r*B; i<(r+1)*B && i<n; i++)
            cudaMemcpyPeer(device_Dist+i*n,0, device_Dist_1+i*n,1 , n*sizeof(unsigned int));

        #ifdef TIME
            float Commtime_Phase1;
            cudaEventRecord(Commstop_device_1);
            cudaEventSynchronize(Commstop_device_1); // WAIT until 'stop' complete.
            cudaEventElapsedTime(&Commtime_Phase1, Commstart_device_1, Commstop_device_1);
            // printf("Phase1 mem copy took %.8f seconds\n",Commtime_Phase1/1000);
            Total_comm_time += Commtime_Phase1;
        #endif

    }
```

Phase 2 & Phase 3 : Both devices calculate the blocks within their area.

```

        /* ----------- Phase 2 ------------- */
        cudaSetDevice(0);
        // Compute four sub-phase
        if(r*B < device_0_Boundary){…
        }
        // Compute ONLY 2-3
        else{…
        }

        cudaSetDevice(1);
        // Compute ONLY 2-4
        if(r*B < device_0_Boundary){    …
        }
        // Compute four sub-phase
        else{…
        }
```

```
        /* ---------- Phase 2 ------------ */
        cudaSetDevice(0);
        // Compute four sub-phase
        if(r*B < device_0_Boundary){

            // TODO : Modify cal() and cal3() : Need to pass boundary into!!
            // 2-1
            if(r !=0){
                dim3 nB(1,r);
                cal3<<< nB, num_threads , sizeof(int)*B*B*3>>>(device_0_Boundary, device_Dist, n,  B, r,         r, 0);
            }
            // 2-2
            if(round -r-1 !=0){
                dim3 nB(1,round - r - 1);
                cal3<<< nB, num_threads , sizeof(int)*B*B*3 >>>(device_0_Boundary, device_Dist, n, B, r,       r, r + 1);
            }
            //2-3
            if(r!=0){
                dim3 nB(r,1);
                cal3<<< nB, num_threads , sizeof(int)*B*B*3>>>(device_0_Boundary, device_Dist, n, B, r,             0, r);
            }

            // 2-4
            if(round-r-1 !=0) {
                dim3 nB(round - r - 1,1);
                cal3<<< nB , num_threads, sizeof(int)*B*B*3 >>>(device_0_Boundary, device_Dist, n, B, r,  r + 1, r);
            }

        }
        // Compute ONLY 2-3
        else{…
        }
```

```
        /* ----------- Phase 3 ------------- */

            cudaSetDevice(0);
            if(r != 0){…
            }

            if(r !=0 && (round-r-1) !=0){…
            }

            if(r !=0 && round-r-1 !=0){…
            }

            if(round-r-1 !=0){…
            }

            cudaSetDevice(1);
            if(r != 0){…
            }

            if(r !=0 && (round-r-1) !=0){…
            }

            if(r !=0 && round-r-1 !=0){…
            }

            if(round-r-1 !=0){…
            }

```

```
        cudaSetDevice(1);
        if(r != 0){
            dim3 nB(r,r);
            // cal3<<< nB, num_threads            , sizeof(int)*B*B*3       >>>(device_Dist, n, B, r, 0, 0, r, r);
            cal3_1<<< nB, num_threads            , sizeof(int)*B*B*3       >>>(device_0_Boundary,device_Dist_1, n, B, r, 0, 0);
        }

        if(r !=0 && (round-r-1) !=0){
            dim3 nB(r,(round-r-1));
            // cal3<<< nB, num_threads       , sizeof(int)*B*B*3    >>>(device_Dist, n, B, r,   0, r + 1,   round - r - 1, r);
            cal3_1<<< nB, num_threads       , sizeof(int)*B*B*3    >>>(device_0_Boundary,device_Dist_1, n, B, r,   0, r + 1);
        }

        if(r !=0 && round-r-1 !=0){
            dim3 nB((round-r-1),r);
            // cal3<<< nB  ,num_threads       , sizeof(int)*B*B*3    >>>(device_Dist, n, B, r, r + 1, 0, r, round - r - 1);
            cal3_1<<< nB  ,num_threads       , sizeof(int)*B*B*3    >>>(device_0_Boundary, device_Dist_1, n, B, r, r + 1, 0);
        }

        if(round-r-1 !=0){
            dim3 nB_p3(round - r - 1, round - r - 1);
            // cal3<<< nB_p3, num_threads, sizeof(int)*B*B*3      >>>(device_Dist, n, B, r, r + 1, r + 1, round - r - 1, round - r
            cal3_1<<< nB_p3, num_threads, sizeof(int)*B*B*3      >>>(device_0_Boundary,device_Dist_1, n, B, r, r + 1, r + 1);
        }

    #ifdef DEBUG_DEVICE_DIST⋯
    #endif
```

4. Device function:

I add one more argument ‘device_Boundary’ to the device functions.

```

// PHASE 1 : ONE Block do k iterations with B*B threads.
// __global__ void cal(int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y, int block_width, int blo
__global__ void cal(int device_Boundary, int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y){⋯
}

// phase 1 for device 1
__global__ void cal_1(int device_Boundary, int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y){⋯
}

// __global__ void cal3(int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y, int block_width, int bl
__global__ void cal3(int device_Boundary, int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y){⋯
}

// Phase 3 for device 1.
__global__ void cal3_1(int device_Boundary, int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y){⋯
}

```

Inside these functions, simply replace the original if(i<n) to if(i<device_Boundary)

```
// __global__ void cal3(int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y, int block_width, int b
__global__ void cal3(int device_Boundary, int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y){

    __shared__ int S[32*32*3];
    // int i = block_start_y* B + blockIdx.y * B + threadIdx.y;
    // int j = block_start_x* B + blockIdx.x * B + threadIdx.x;
    int i = block_start_x* B + blockIdx.x * B + threadIdx.y;
    int j = block_start_y* B + blockIdx.y * B + threadIdx.x;


    // S[Addr(1, threadIdx.y, ((Round*B + threadIdx.x)%B), B)] = device_Dist[Addr(0,i,(Round*B + threadIdx.x),n)];
    // S[Addr(2, ((Round*B + threadIdx.y)%B), threadIdx.x, B)] = device_Dist[Addr(0,(Round*B + threadIdx.y),j,n)];

    if(i<device_Boundary && (Round*B + threadIdx.x) <n) S[Addr(1, threadIdx.y, ((Round*B + threadIdx.x)%B), B)] = device_Dist[A
    if(j<n && (Round*B + threadIdx.y)<n) S[Addr(2, ((Round*B + threadIdx.y)%B), threadIdx.x, B)] = device_Dist[Addr(0,(Round*B


    if(i<device_Boundary && j<n){
    // For each thread, calculate one edge.⋯
    }
}

```

for device_1, it should be if(i>=device_Boundary && i<n)

```
// Phase 3 for device 1.
__global__ void cal3_1(int device_Boundary, int* device_Dist, int n, int B, int Round, int block_start_x, int block_start_y){

    __shared__ int S[32*32*3];
    int i = block_start_x* B + blockIdx.x * B + threadIdx.y;
    int j = block_start_y* B + blockIdx.y * B + threadIdx.x;

    if( i<n && (Round*B + threadIdx.x) <n) S[Addr(1, threadIdx.y, ((Round*B + threadIdx.x)%B), B)] = device_Dist[Addr(0,i,(Round*
    if((Round*B + threadIdx.y)<n &&  j<n ) S[Addr(2, ((Round*B + threadIdx.y)%B), threadIdx.x, B)] = device_Dist[Addr(0,(Round*B 


    if(i>=device_Boundary && i<n && j<n){
    // For each thread, calculate one edge.
        S[ Addr(0,threadIdx.y, threadIdx.x, B)  ] = device_Dist[Addr(0,i,j,n)];
        __syncthreads();

            for (int iter = 0; iter<B &&  Round*B+iter <n; iter++){ //k = Round * B; k < (Round + 1) * B && k < n; ++k) {
                        S[Addr(0,threadIdx.y, threadIdx.x, B)] = min(S[Addr(1, threadIdx.y, iter, B)]+ S[Addr(2, iter, threadIdx.
            }
            device_Dist[Addr(0,i,j,n)] = S[Addr(0,threadIdx.y, threadIdx.x, B)];
    }
}
```

5. After ‘round’ iterations, device_0 and device_1 copies their ‘half-adj_matrix’ data back to CPU host (to Dist).

```

    // Independently copy back to CPU
    cudaMemcpyAsync(Dist, device_Dist, n*device_0_Boundary*sizeof(unsigned int), cudaMemcpyDeviceToHost);
    cudaMemcpyAsync(Dist+device_0_Boundary*n, device_Dist_1+device_0_Boundary*n, ( n*n - n*device_0_Boundary) *sizeof(unsigned int), cu
    cudaDeviceSynchronize();

```

6. Then CPU writes output buffer to the output file.

2. Experiment & Analysis

(1) System Spec

I use the `hades` server for the experiments.

(2) Weak Scalability

I generate my own test cases to measure the execution time.

How I measure computing / memory copy / I/O time:

For **Input/ Output time** in CPU, I use the code below to measure the time took by below functions:

For input() :

```c
    #ifdef TIME
        // struct timespec start, end, temp;
        struct timespec total_starttime;
        struct timespec total_temp;
        struct timespec start;
        struct timespec end;
        struct timespec temp;
        double IO_time=0.0;
        double Total_time = 0.0;
        clock_gettime(CLOCK_MONOTONIC, &total_starttime);
        clock_gettime(CLOCK_MONOTONIC, &start);
    #endif

    input(argv[1]);

    #ifdef DEBUG_DEVICE_DIST…
    #endif

    #ifdef TIME
        clock_gettime(CLOCK_MONOTONIC, &end);
        if ((end.tv_nsec - start.tv_nsec) < 0) {
            temp.tv_sec = end.tv_sec-start.tv_sec-1;
            temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
        } else {
            temp.tv_sec = end.tv_sec - start.tv_sec;
            temp.tv_nsec = end.tv_nsec - start.tv_nsec;
        }
        IO_time += temp.tv_sec + (double) temp.tv_nsec / 1000000000.0;
    #endif
```

For output() :

```c
    #ifdef TIME
        clock_gettime(CLOCK_MONOTONIC, &start);
    #endif

    output(argv[2]);

    #ifdef TIME
        clock_gettime(CLOCK_MONOTONIC, &end);
        // IO Time
        if ((end.tv_nsec - start.tv_nsec) < 0) {
            temp.tv_sec = end.tv_sec-start.tv_sec-1;
            temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
        } else {
            temp.tv_sec = end.tv_sec - start.tv_sec;
            temp.tv_nsec = end.tv_nsec - start.tv_nsec;
        }
        // Total Time
        if ((end.tv_nsec - total_starttime.tv_nsec) < 0) {
            total_temp.tv_sec = end.tv_sec-total_starttime.tv_sec-1;
            total_temp.tv_nsec = 1000000000 + end.tv_nsec - total_starttime.tv_nsec;
        } else {
            total_temp.tv_sec = end.tv_sec - total_starttime.tv_sec;
            total_temp.tv_nsec = end.tv_nsec - total_starttime.tv_nsec;
        }

        IO_time += temp.tv_sec + (double) temp.tv_nsec / 1000000000.0;
        Total_time = total_temp.tv_sec + (double) total_temp.tv_nsec / 1000000000.0;
    #endif

    #ifdef TIME
        printf("IO Time: %.8f seconds\n", IO_time);
        printf("Total Time: %.8f seconds\n",Total_time);
    #endif
```

And I simply add two measurements to get IO_time in double precision.

For the **memory copy time** from CPU to GPU (host to device) or from GPU to CPU (device to host), I use cuda api : cudaEventElapsedTime()

Host to Device:

```
                cudaMalloc(&device_Dist,   n * n* sizeof(unsigned int));

                #ifdef TIME
                    cudaEventRecord(Commstart);
                #endif
                cudaMemcpyAsync(device_Dist,   Dist, n* n*sizeof(unsigned int), cudaMemcpyHostToDevice);
```

```
    #ifdef TIME
        float Commtime;
        cudaSetDevice(0);
        cudaEventRecord(Commstop);
        cudaEventSynchronize(Commstop); // WAIT until 'stop' complete.
        cudaEventElapsedTime(&Commtime, Commstart, Commstop);

        printf("H2D copy took %.8f seconds\n",Commtime/1000);
        Total_comm_time += Commtime;
    #endif...
    dim3 num_threads(B,B);
```

Device to Host :

```
#ifdef TIME
    cudaSetDevice(0);
    cudaEventRecord(Commstart);
#endif

// Independently copy back to CPU
cudaMemcpyAsync(Dist, device_Dist, n*device_0_Boundary*sizeof(unsigned int), cudaMemcpyDeviceToHost);
cudaMemcpyAsync(Dist+device_0_Boundary*n, device_Dist_1+device_0_Boundary*n, ( n*n - n*device_0_Boundary) *sizeof(unsigned int), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();

#ifdef TIME
    float Commtime_D2H;
    cudaEventRecord(Commstop);
    cudaEventSynchronize(Commstop); // WAIT until 'stop' complete.
    cudaEventElapsedTime(&Commtime_D2H, Commstart, Commstop);
    printf("D2H copy took %.8f seconds\n",Commtime_D2H/1000);
    // printf("Took %.8f milliseconds",time);
    Total_comm_time += Commtime_D2H;
    printf("Communication %.8f seconds\n",Total_comm_time/1000);
#endif
```

Device to Device (GPU0 -> GPU 1 or GPU 1 -> GPU 0)

```
            // Copy pivot ROW to the other device.
            #ifdef TIME
                cudaEventRecord(Commstart_device_1);
            #endif
            for(int i= r*B; i<(r+1)*B && i<n; i++)
                cudaMemcpyPeer(device_Dist+i*n,0, device_Dist_1+i*n,1 , n*sizeof(unsigned int));

            #ifdef TIME
                float Commtime_Phase1;
                cudaEventRecord(Commstop_device_1);
                cudaEventSynchronize(Commstop_device_1); // WAIT until 'stop' complete.
                cudaEventElapsedTime(&Commtime_Phase1, Commstart_device_1, Commstop_device_1);
                // printf("Phase1 mem copy took %.8f seconds\n",Commtime_Phase1/1000);
                Total_comm_time += Commtime_Phase1;
            #endif
```

I divide it by 1000 to get the unit in seconds.

For the Computation Time, I use the same cuda API with different cudaEvent variables to measure the time took by the kernel computations.

```c
    // Record Computation time
    #ifdef TIME
        cudaSetDevice(0);
        cudaEvent_t start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start);
    #endif
```

```c
    #ifdef TIME
        cudaEventRecord(stop);
        cudaEventSynchronize(stop); // WAIT until 'stop' complete.
        float time;
        cudaEventElapsedTime(&time, start, stop);
        // printf("Took %.8f milliseconds",time);
        printf("Computation(raw): Took %.8f seconds\n",(time)/1000);
        printf("Computation: Took %.8f seconds\n",(time-Total_comm_time)/1000);
    #endif
```

The sample result from running one test case:

```
========== Comparing results... ===========
Job Finished
[pp20s35@hades02 hw4-2]$ ./hw4-2 cases/p31k1 mycases1231/p31k1.out cases/p31k1.out.sha256
V: 31000, E: 15125277
Large n :
Blocking factor: 32 (num of pixel(adj entries) in a Block)
 32 * 32 block
ceil(31000, 2) :15500
ceil % B remainder : 12
device_0_Boundary: 15520
Can 0 access 1? 1
Can 1 access 0? 1
omp t1 allocate & copy gpu 1
omp t0 allocate & copy gpu 0
H2D copy took 0.65168601 seconds
D2H copy took 0.44236004 seconds
Communication 6.10602903 seconds
Computation(raw): Took 74.14246368 seconds
Computation: Took 68.03643036 seconds
IO Time: 74.21362948 seconds
Total Time: 148.57487986 seconds
========== Comparing results... ===========
Job Finished
```

Weak Scalability measurements method:

First, generate different sizes of test cases.

For each test case, run the sequential version of Blocked APSP algorithm and measure sequential time. I run it on apollo server.

```
[pp20s35@apollo31 hw3]$ ./hw3_ta hw4-2cases/v10.in v10.out
Computation Time: 0.00000724 seconds
[pp20s35@apollo31 hw3]$ ./hw3_ta hw4-2cases/v30.in v30.out
Computation Time: 0.00009289 seconds
[pp20s35@apollo31 hw3]$ ./hw3_ta hw4-2cases/v50.in v50.out
Computation Time: 0.00034320 seconds
[pp20s35@apollo31 hw3]$ ./hw3_ta hw4-2cases/v100.in v100.out
Computation Time: 0.00305902 seconds
[pp20s35@apollo31 hw3]$ ./hw3_ta hw4-2cases/v500.in v500.out
Computation Time: 0.18051960 seconds
[pp20s35@apollo31 hw3]$ ./hw3_ta hw4-2cases/v1000.in v1000.out
Computation Time: 1.20472436 seconds
[pp20s35@apollo31 hw3]$ ./hw3_ta hw4-2cases/v5000.in v5000.out
Computation Time: 141.01815443 seconds
[pp20s35@apollo31 hw3]$ ./hw3_ta hw4-2cases/v10000.in v10000.out
```

Note: I check the correctness using my ./Verification_twoFiles program.

```
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles v10.out hw4-2cases/v10.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles v30.out hw4-2cases/v30.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles v50.out hw4-2cases/v50.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles v100.out hw4-2cases/v100.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles v500.out hw4-2cases/v500.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles v1000.out hw4-2cases/v1000.out
Files are equal
[pp20s35@apollo31 hw3]$ ./Verification_twoFiles v5000.out hw4-2cases/v5000.out
Files are equal
```

Then, run the multi-GPU version of Blocked APSP on hades and measure the running time.

Note: How do I generate my test cases?

I use the method in hw3:

First, user type desired v(# of vertices) and e (# of edges) in the termi

```
Work took 0.00020242 seconds
[pp20s35@apollo31 hw3]$ ./gen_input 100 1000 hw4-2cases/v100.in hw4-2cases/v100.out
v: 100
e: 1000
Done writing to argv[3].in
Start generating answer...
Numebr of Vertices: 100
Number of Edges: 1000
Tooks 0.00246918 on Computation
Done writing output file.
Tooks 0.01394578 on Output
Work took 0.01650948 seconds
```

Then, the gen_input.cc read in v and e you input.

For each entry in the adjacency matrix, it randoms an integer (weight) and 0<=w<=1000.

```

    // // ========== Initialize adj matrix ============ //

    adj_matrix = (int*)malloc(sizeof(int)*v*v);

    for(int i=0; i<v; i++){
        for(int j=0; j<v; j++){
            if(i==j) adj_matrix[i*v+j] = 0;
            else adj_matrix[i*v+j] = INF;
        }
    }

```

```
// // ============== NOW, for-loop to generate Edges ============== //
for(int i=0; i<e; i++){
    // 1. generate src
    int src = rand() % (vertex_max +1 - vertex_min) + vertex_min;
    // 2. generate dst  (dst != src)
    int dst = rand() % (vertex_max +1 - vertex_min) + vertex_min;
    while(dst==src || adj_matrix[src*v+dst] != INF){
        dst = rand() % (vertex_max +1 - vertex_min) + vertex_min;
    }

    // 3. generate weight (0~1000)
    int weight = rand()% (w_max +1 - w_min) + w_min;
    // printf("i:%d  src %d , dst %d, weight: %d\n",i , src,dst,weight);
    adj_matrix[src*v+dst] = weight;
    // printf("i: %d, src: %d, dst: %d, w: %d\n",i, src, dst, adj_matrix[src][dst]);
    output_buf[2 + i*3] = src;
    output_buf[2 + i*3 +1] = dst;
    output_buf[2 + i*3 +2] = weight;
}
```

Note that we cannot generate repeated edges so I use while() to check if the edge was already specified.

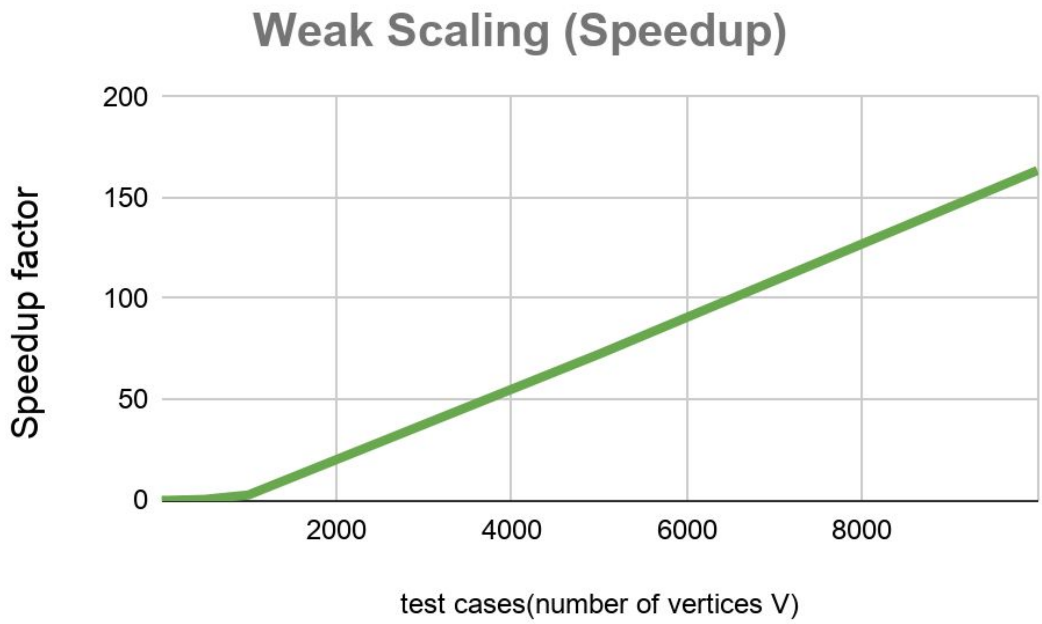
Finally, write the generated test case into output file (xxx.in)

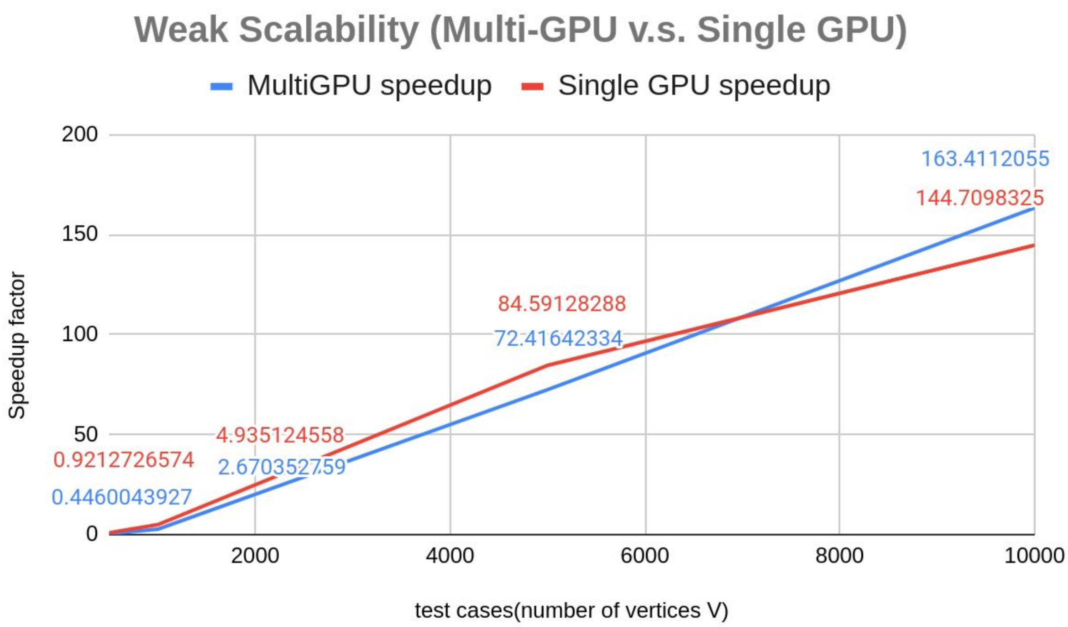
```
    // // =============== Write to OUTPUT FILE =================== //

    FILE *pFile;
    // argv[1]:v, argv[2]: e
    pFile = fopen(argv[3],"wb");
    if( NULL == pFile ){
        printf( "open file failure" );
        return 1;
    }else{
        fwrite(output_buf, sizeof(int), 2+3*e ,pFile);
    }

    fclose(pFile);

    free(output_buf);

    printf("Done writing to argv[3].in\n");

// Read INPUT_FILE and print offset.
```

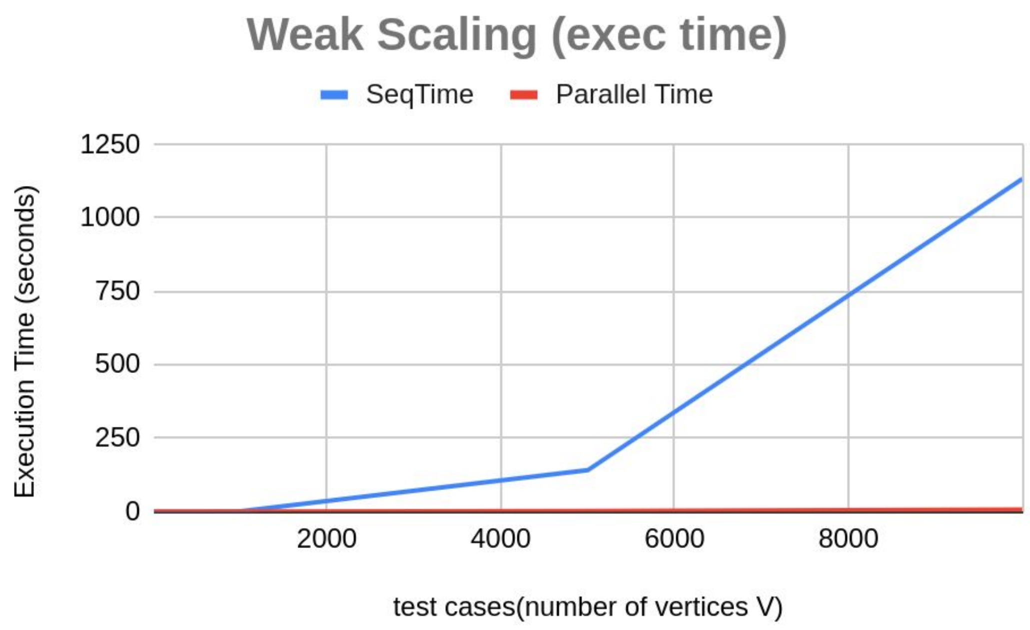
At the same time, run the sequential Floyd-Warshall algorithm to generate corresponding correct output answer file. (xxx.out) This is for checking correctness of the parallel implementation code.

```
   // ============= All Pair Shortest Path ================= //
    double Computation_starttime;
    Computation_starttime = omp_get_wtime();

    #ifdef APSP

    // #pragma omp parallel for schedule(dynamic) collapse(3)
    for( int k=0; k<num_of_vertices; k++){
        for(int i=0; i<num_of_vertices; i++){
            // #pragma omp parallel for schedule(dynamic)
            for(int j=0; j<num_of_vertices; j++){
                // if(i==j) continue;
                // if( answer_buf[i*num_of_vertices+j] > answer_buf[i*num_of_vertices + k] + answe
                //     answer_buf[i*num_of_vertices+j] = answer_buf[i*num_of_vertices + k] + answe
                // }

                answer_buf[i*num_of_vertices+j] = std::min(answer_buf[i*num_of_vertices+j] , answe

                // else do nothing.
            }
        }
    }
    #endif
```

Then plot the weak scalability plot:
Vertical axis = Speedup = tseq / tparallel

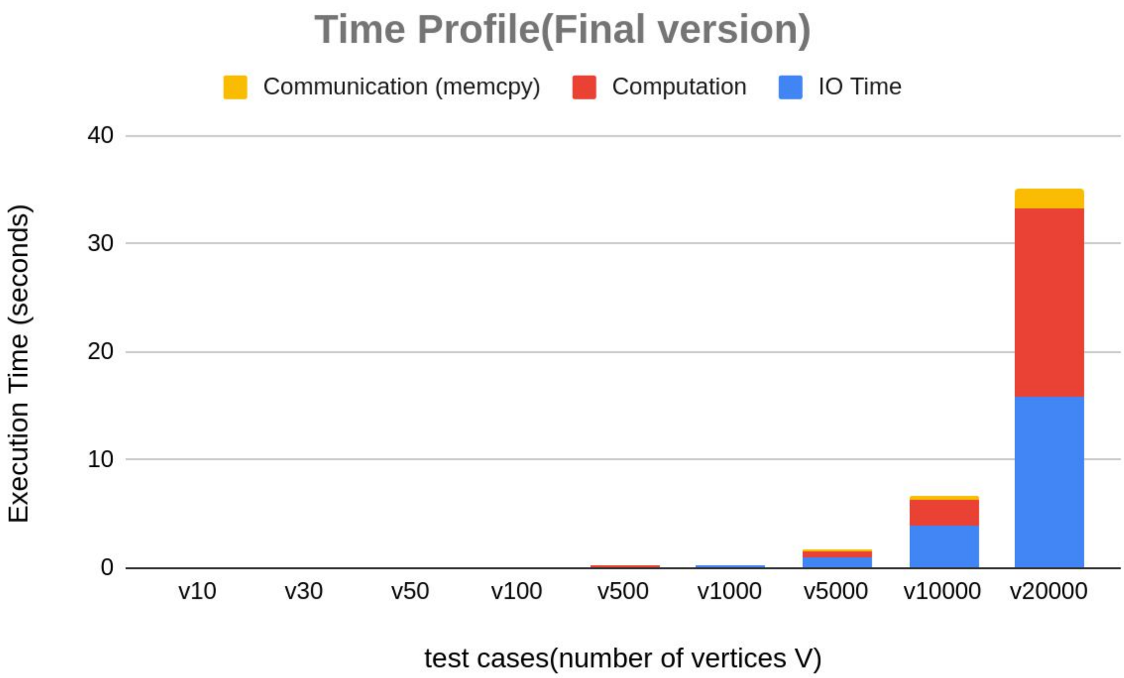




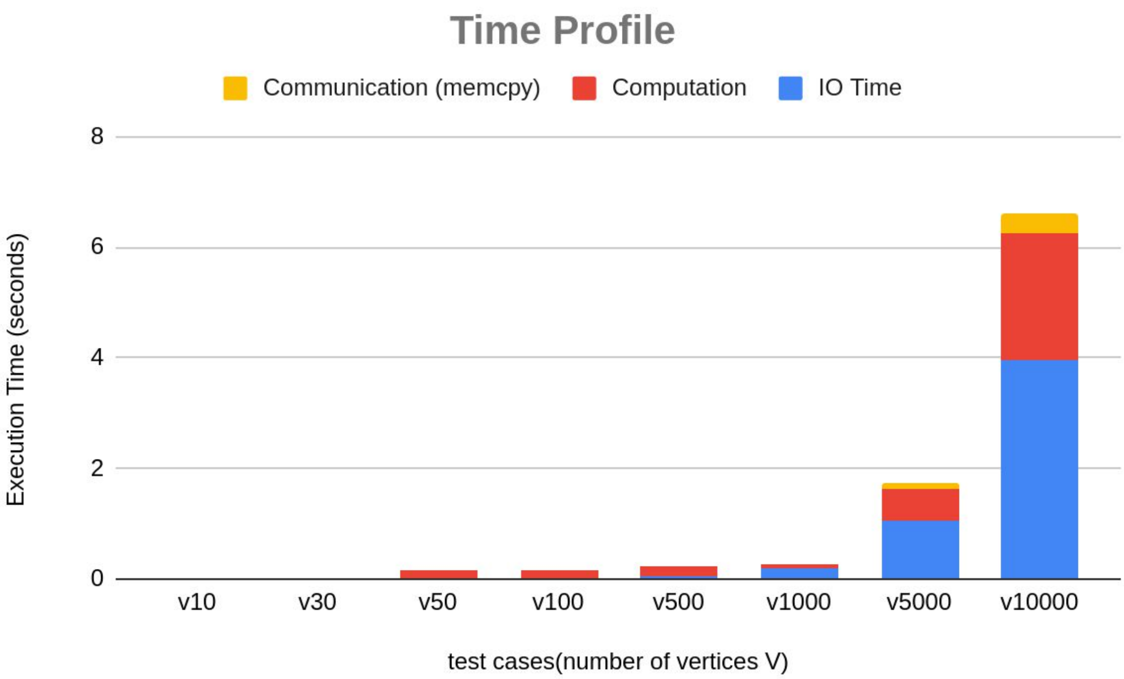


Since **GPU programs are IO-bound programs**, if we use weak scaling we can maintain the communication overhead to be approximately constant with respect to different test case size, then linear speedup is easier to achieved.

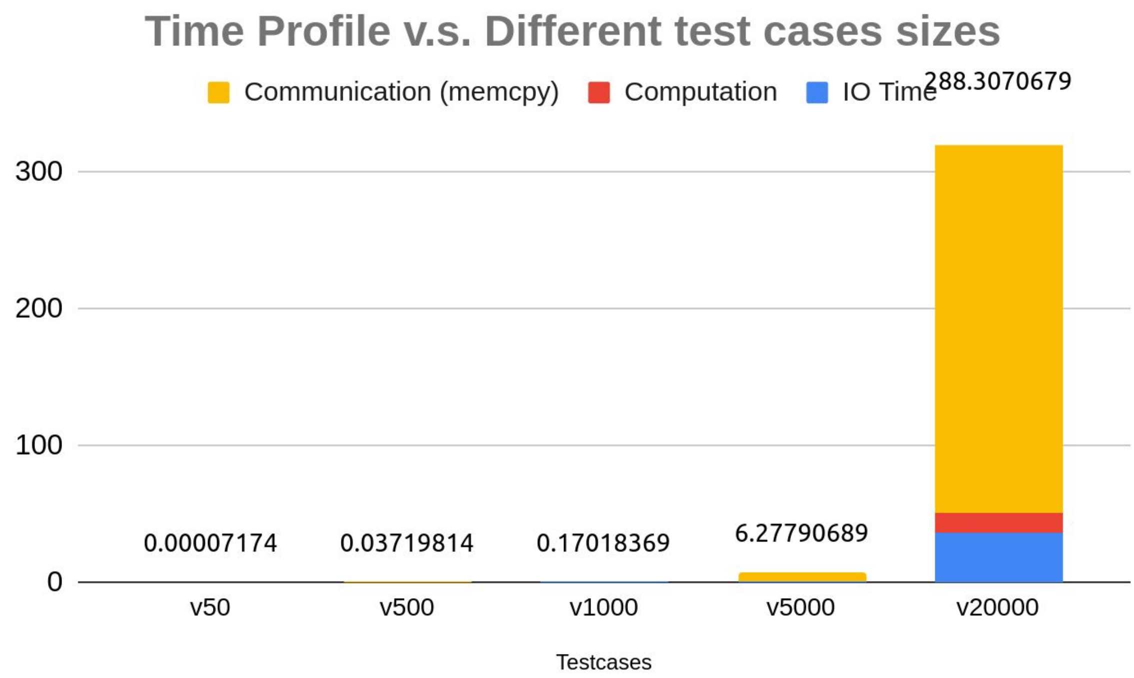
(2) Time Profile
Time Profile with different size of test cases (own generated)
Final version:



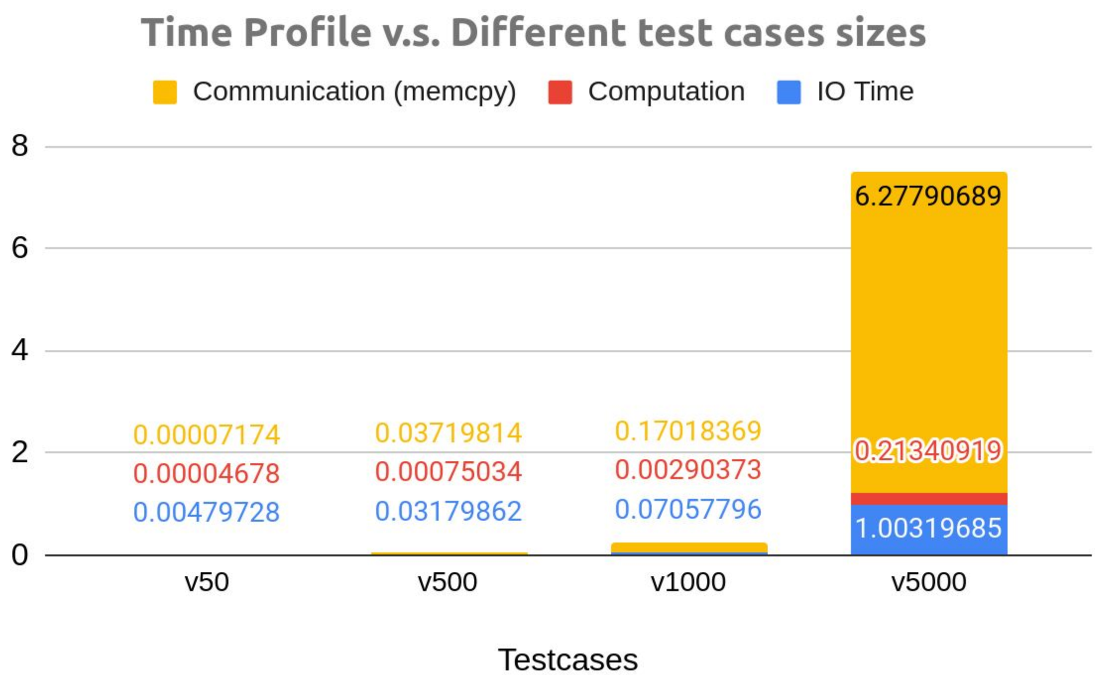
without v20000 :



Older version:



Since the scale is not observable, remove v2000 case and replot.



可以看到 I/O 和 Computation 時間相近，Bottleneck 在 communication time

(3) CUDA Profile results.

a. occupancy, sm_efficiency, shared_memory/dram throughput.

```
==39339== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==39339== Profiling result:
==39339== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                          achieved_occupancy                        Achieved Occupancy    0.440015    0.939980    0.839004
    Kernel: cal(int, int*, int, int, int, int, int)
         79                          achieved_occupancy                        Achieved Occupancy    0.496256    0.496968    0.496429
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                          achieved_occupancy                        Achieved Occupancy    0.215674    0.496736    0.492822
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                          achieved_occupancy                        Achieved Occupancy    0.232061    0.928872    0.835738
[pp20s35@ha
```

```
Total Time: 10.38221640 seconds
========== Comparing results... ===========
Job Finished
==39875== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==39875== Profiling result:
==39875== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                                sm_efficiency                   Multiprocessor Activity       1.18%      99.73%      84.92%
    Kernel: cal(int, int*, int, int, int, int, int)
         79                                sm_efficiency                   Multiprocessor Activity       3.49%       3.68%       3.62%
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                                sm_efficiency                   Multiprocessor Activity       1.94%       3.73%       3.60%
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                                sm_efficiency                   Multiprocessor Activity       1.83%      99.73%      85.06%
[pp20s35@hades01 hw4-2]$
```

```
========== Comparing results... ===========
Job Finished
==15473== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==15473== Profiling result:
==15473== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                        shared_load_throughput            Shared Memory Load Throughput  0.00000B/s  2051.8GB/s  1599.6GB/s
    Kernel: cal(int, int*, int, int, int, int, int)
         79                        shared_load_throughput            Shared Memory Load Throughput  47.486GB/s  57.798GB/s  52.110GB/s
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                        shared_load_throughput            Shared Memory Load Throughput  8.7226GB/s  52.738GB/s  50.947GB/s
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                        shared_load_throughput            Shared Memory Load Throughput  0.00000B/s  1834.5GB/s  1462.4GB/s
[pp20s35@hades01 hw4-2]$
des01 hw4-2
```

```
========== Comparing results... ===========
Job Finished
==40353== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==40353== Profiling result:
==40353== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                       shared_store_throughput           Shared Memory Store Throughput  2.3374GB/s  1187.5GB/s  1023.7GB/s
    Kernel: cal(int, int*, int, int, int, int, int)
         79                       shared_store_throughput           Shared Memory Store Throughput  16.884GB/s  21.497GB/s  20.597GB/s
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                       shared_store_throughput           Shared Memory Store Throughput  3.8317GB/s  21.734GB/s  20.821GB/s
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                       shared_store_throughput           Shared Memory Store Throughput  3.7844GB/s  1188.3GB/s  987.31GB/s
[pp20s35@ha
```

```
========== Comparing results... ===========
Job Finished
==15537== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==15537== Profiling result:
==15537== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                          dram_read_throughput             Device Memory Read Throughput  0.00000B/s  95.072GB/s  25.172GB/s
    Kernel: cal(int, int*, int, int, int, int, int)
         79                          dram_read_throughput             Device Memory Read Throughput  0.00000B/s  1.0486GB/s  567.05MB/s
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                          dram_read_throughput             Device Memory Read Throughput  0.00000B/s  754.06MB/s  297.47MB/s
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                          dram_read_throughput             Device Memory Read Throughput  0.00000B/s  57.424GB/s  26.764GB/s
[pp20s35@hades01 hw4-2]$
```

```
========== Comparing results... ===========
Job Finished
==15751== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==15751== Profiling result:
==15751== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                        dram_write_throughput            Device Memory Write Throughput  0.00000B/s  45.394GB/s  23.877GB/s
    Kernel: cal(int, int*, int, int, int, int, int)
         79                        dram_write_throughput            Device Memory Write Throughput  0.00000B/s  152.59MB/s  70.007MB/s
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                        dram_write_throughput            Device Memory Write Throughput  82.416MB/s  557.40MB/s  283.63MB/s
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                        dram_write_throughput            Device Memory Write Throughput  0.00000B/s  40.975GB/s  22.324GB/s
[pp20s35@hades01 hw4-2]$
ades01 hw4
```

b. inst_integer

```
========== Comparing results... ===========
Job Finished
==15504== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==15504== Profiling result:
==15504== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                                 inst_integer                   Integer Instructions       26888  3826312640   597318438
    Kernel: cal(int, int*, int, int, int, int, int)
         79                                 inst_integer                   Integer Instructions      203776      203776      203776
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                                 inst_integer                   Integer Instructions       13376      204800      202345
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                                 inst_integer                   Integer Instructions       47376  4222612224   658412578
[pp20s35@hades01 hw4-2]$
```

c. gld_throughput, and gst_throughput

Global load throughput:

```
========== Comparing results... ===========
Job Finished
==39265== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==39265== Profiling result:
==39265== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                               gld_throughput                  Global Load Throughput  103.80MB/s  14.768GB/s  14.447GB/s
    Kernel: cal(int, int*, int, int, int, int, int)
         79                               gld_throughput                  Global Load Throughput  265.37MB/s  302.90MB/s  281.45MB/s
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                               gld_throughput                  Global Load Throughput  38.728MB/s  294.83MB/s  278.25MB/s
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                               gld_throughput                  Global Load Throughput  146.72MB/s  16.357GB/s  15.228GB/s
[pp20s35@hades01 hw4-2]$
```

Global store throughput

```
========== Comparing results... ===========
Job Finished
==39370== Profiling application: ./hw4-2 /home/pp20/share/hw4-2/cases/c04.1 c04.1.out
==39370== Profiling result:
==39370== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int, int*, int, int, int, int, int)
       1014                               gst_throughput                 Global Store Throughput  0.00000B/s  47.678GB/s  25.127GB/s
    Kernel: cal(int, int*, int, int, int, int, int)
         79                               gst_throughput                 Global Store Throughput  467.70MB/s  589.71MB/s  566.14MB/s
Device "GeForce GTX 1080 (1)"
    Kernel: cal_1(int, int*, int, int, int, int, int)
         78                               gst_throughput                 Global Store Throughput  97.813MB/s  595.46MB/s  569.23MB/s
    Kernel: cal3_1(int, int*, int, int, int, int, int)
       1011                               gst_throughput                 Global Store Throughput  0.00000B/s  43.178GB/s  23.570GB/s
[pp20s35@hades0
```

v.s. Single GPU version:

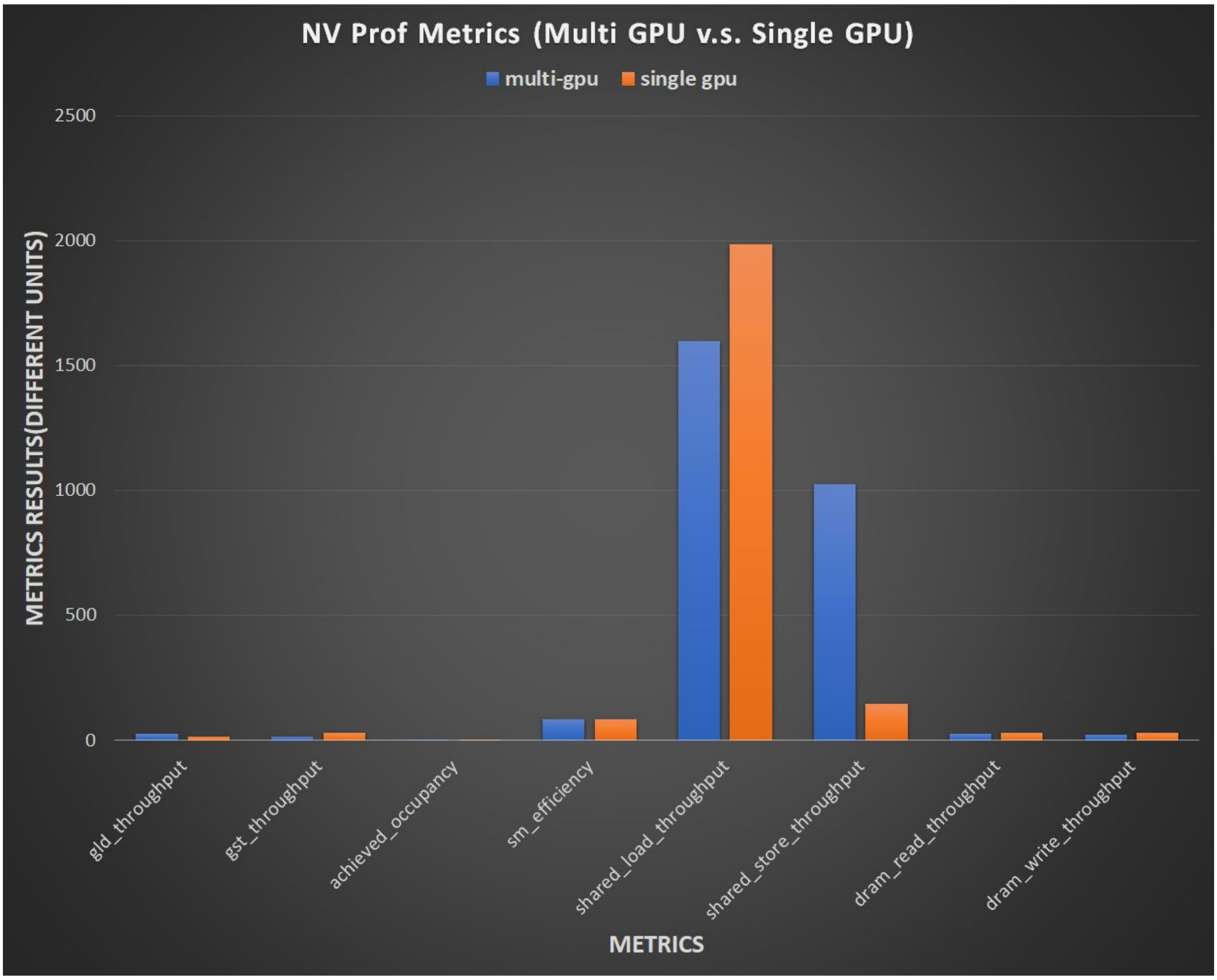
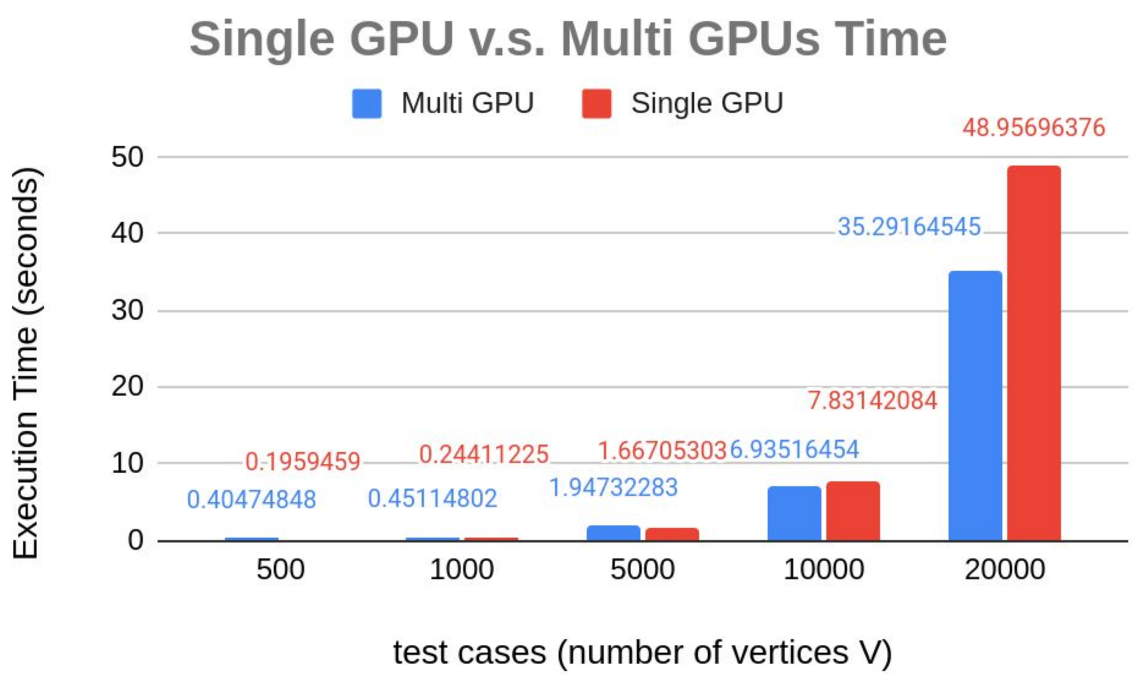


Table:

c04.1	V: 5000 E: 10723117	use: cal3() average	-	x
metrics	multi-gpu	single gpu	unit	Changes
inst_integer	597313438	839492053	x	decrease
gld_throughput	25.127	13.988	GB/s	increase
gst_throughput	14.447	30.396	GB/s	decrease
achieved_occupancy	0.839004	0.858612	x	decrease
sm_efficiency	84.92	83.31	%	increase
shared_load_throughput	1599.6	1984.9	GB/s	decrease
shared_store_throughput	1023.7	146.44	GB/s	increase
dram_read_throughput	25.172	28.762	GB/s	decrease
dram_write_throughput	23.877	29.916	GB/s	decrease

As you can see, only **shared_store_throughput** , **sm_efficiency** and **gld_throughput increase!** Other performance decreases from single GPU version to multi GPU version.
I reckon it's because in Single-GPU version, I didn't optimize my program enough and in multi-GPU version, the problem would be more serious since there are now more communication overhead.

(4) Running time v.s. Single GPU version.
test cases: v500.in, v1000.in, v5000.in, v10000.in



3. Experience & conclusion

(1) What have you learned from this homework?

Through this assignment, I have learned how to implement multi-GPU programming and how to communicate data between different gpu devices. I also learned that to write an efficient CUDA program is really difficult.
I encountered several problems includes:
cuda-memcheck : invalid resource handle error 400.
Eventually I found it's because when I used cudaEventRecord(&start), I didn't cudaSetDevice() to the correct device where I have used cudaEventCreate(&start).

In the older version of my implementation, I used two many cudaMemcpyPeer in each round, so the communication time dominated the total running time. This is terrible since communication is purely overhead.
In my final version, I reduce the communication time to only phase 1 and reduce the data size being copied to only the pivot row.

Though I still cannot pass test cases c06 to c07 and performance cases, which is really frustrating since I have tried my best, I learned a lot.

(2) Feedback :

Thanks for TA's help and thanks for providing apollo and hades server for us to practice parallel programming!