Hw3 report			
學號: 109062233	姓名: 蘇裕恆		繳交時間:

Implementation :

1. Which algorithm do you choose in hw3-1?

```
// printf("from %d to  %d \n" , from , to );
for(int k = 0 ;  k < n ; k++){
    for(int i = from ; i < to ; i++){
        for(int j = 0 ; j < n ; j++){
            Dist[i][j] = std::min(Dist[i][j], Dist[i][k] + Dist[k][j]);
        }
    }
    pthread_barrier_wait(&barrier);
}
pthread_exit(NULL);
```

在 3-1 裡面，我使用的是很 naïve 的 FLOYD-WARSHALL algorithm，並使用 pthread 作為相應的解法。因為主要來說都是 floyd – warshall 也比較好判斷。

2. How do you divide your data in hw3-2, hw3-3?
   兩種方法都差不多，基本上就是把切成 64*64 的方塊並對其執行 block-floyd –warshall algorithm.

3. What's your configuration in hw3-2, hw3-3? And why? (e.g. blocking factor, #blocks, #threads)
   我不知道 blocking factor 與 blocks 的差別，我選擇的是 64 作為 blocksize。
   因為 gtx 1080 的配置，在 share memory 裡面總共有 49152 bytes. 我在 phase 3 裡面的實作需要三個 share memory block。同時，考慮到 warp 的性質是以 32 個資料作為單位，因此考慮 block 與 threads 接為 32 之倍數。
   Thread 在 HW3-2 視為 32*32 ，在 HW3-3 為 16*64 (原本 hw3-3 是參考我的第九版 code 所測計的 因此 thread 為 16*64 (若變為 32*32 將有更好的 performance 但是因為已經全過了所以就沒有更改。)

4. How do you implement the communication in hw3-3?
   我在 hw3-3 裡面，使用的是 openmp 作為 base (我不是很清楚這題想要問什麼 但如果是問 communication function 的話我使用的是 cudaMemcpy

```
if(i >= y_offset && i < y_offset + round ){
    cudaMemcpy(deviceDist[!cpu_thread_id] + i * BlockSize * new_n , deviceDist[cpu_thread_id] + i * BlockSize * new_n , BlockSize * new_n * sizeof(int) , cudaMemcpyDeviceToHost);
    // printf("in round %d there's an exchange from %d to %d \n" , i ,cpu_thread_id , !cpu_thread_id);
}
```

5. Briefly describe your implementations in diagrams, figures or sentences.

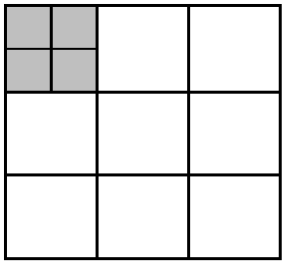
   在 hw3-2 裡面，我總共 implement 了九個版本。因為後面的版本過於優化以至於基本上很難看懂，因此以下為第一版的 code。

```cpp
int main(int argc, char* argv[]) {
    input(argv[1]);
    // int BlockSize = 512;
    int* deviceDist;
    cudaMalloc(&deviceDist, new_n * new_n * sizeof(int));
    cudaMemcpy(deviceDist, Dist, new_n * new_n * sizeof(int), cudaMemcpyHostToDevice);

    int total_block_num = new_n / BlockSize;
    dim3 block_num1(1, 1);
    dim3 block_num2(1, total_block_num);
    dim3 block_num3(total_block_num, total_block_num);
    dim3 thread_each_block(BlockSize,BlockSize);
    // printf("the total block : %d \n" , total_block_num);
    // printf("the n is %d and the new_n is %d \n" ,  n , new_n);

    for(int i = 0 ; i < total_block_num ; i++){
        phase1<<<block_num1 , thread_each_block>>>(deviceDist , i , new_n); // phase 1
        phase2<<<block_num2 , thread_each_block>>>(deviceDist , i , new_n); // phase 2
        phase3<<<block_num3 , thread_each_block>>>(deviceDist , i , new_n); // phase 3
    }
    cudaMemcpy(Dist , deviceDist, new_n * new_n * sizeof(int), cudaMemcpyDeviceToHost);
    output(argv[2]);
    return 0;
}
```

總共有三個 phase，而一開始時候，會將 block 設為一個，並執行一開始使的計算



(a) Phase 1

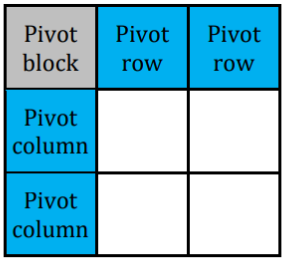
概念上來說，就是將 naïve floyd-warshell 在小範圍內實作

```cpp
__global__ void phase1(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize][BlockSize];
    int x_real = threadIdx.x + round_cnt * BlockSize;
    int y_real = threadIdx.y + round_cnt * BlockSize;
    int x = threadIdx.x;
    int y = threadIdx.y;

    /*if(x_real >= new_n || y_real  >= new_n){
        return;
    }*/
    store[x][y] = Dist_d[x_real * new_n + y_real];

    __syncthreads();
    for(int i = 0 ; i < BlockSize ; i++){
        store[x][y] = min(store[x][i] + store[i][y] , store[x][y]);
        __syncthreads();
    }
    Dist_d[x_real * new_n + y_real] = store[x][y];
}
```

注意在每次 run 完以後就必須像是 hw3-1 的實作一樣將 thread 同步 (因為 data 有 dependency)。



(b) Phase 2

```
__global__ void phase2(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize][BlockSize];
    __shared__ int vertical[BlockSize][BlockSize];
    __shared__ int herizonal[BlockSize][BlockSize];
    int x = threadIdx.x;
    int y = threadIdx.y;
    int block_num = blockIdx.y;
    int x_ver = threadIdx.x + round_cnt * BlockSize;
    int x_her = threadIdx.x + block_num * BlockSize;
    int y_ver = threadIdx.y + block_num * BlockSize;
    int y_her = threadIdx.y + round_cnt * BlockSize;
    // if duplicate with the phase 1
    if(block_num == round_cnt ){
        return;
    }
    // since we calculate both vertical and herizonal at once ,we can't delete both given one doesn't fulfill
    store[x][y] = Dist_d[x_ver * new_n + y_her];
    herizonal[x][y] = Dist_d[ x_her * new_n + y_her];
    vertical[x][y] = Dist_d[ x_ver * new_n + y_ver ];
    __syncthreads();
    #pragma unroll 32
    for(int i = 0 ; i < BlockSize ; i++){
        vertical[x][y] = min( store[x][i] + vertical[i][y] , vertical[x][y]);
        herizonal[x][y] =  min( herizonal[x][i] + store[i][y] , herizonal[x][y]);
    }
    Dist_d[x_her * new_n + y_her] = herizonal[x][y];
    Dist_d[x_ver * new_n + y_ver] = vertical[x][y];
}
```

在 phase 二這邊我們使用一個(1, total_block_num) 作為 block 的形狀。同時，我的實作會同時將 row 跟 column 算完並回傳。基本上來說 vertical 就是 column。

```
__global__ void phase3(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize][BlockSize];
    __shared__ int vertical[BlockSize][BlockSize];
    __shared__ int herizonal[BlockSize][BlockSize];
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;
    int x = threadIdx.x;
    int y = threadIdx.y;
    if(block_x == round_cnt || block_y == round_cnt){
        return;
    }
    int x_real = block_x * BlockSize + x;
    int y_real = block_y * BlockSize + y;
    int x_her = x_real;
    int y_her = round_cnt * BlockSize + y;
    int x_ver = round_cnt * BlockSize + x;
    int y_ver = y_real;
    if(x_real >= new_n || y_real >= new_n){
        return;
    }
    store[x][y] = Dist_d[x_real * new_n + y_real];
    vertical[x][y] = Dist_d[x_ver * new_n + y_ver];
    herizonal[x][y] = Dist_d[x_her * new_n + y_her];
    __syncthreads();
    #pragma unroll 32
    for(int i = 0 ; i < BlockSize ; i++){
        store[x][y] = min(herizonal[x][i] + vertical[i][y] , store[x][y]);
    }
    Dist_d[x_real * new_n + y_real] = store[x][y] ;
}
```

在 phase3 這邊，我們將 row 跟 column 找出來，並透過 spec 給的方式做 reduction。注意: 此階段與上一階段接在每個 block 做同步。基本上就是找 pivot row 跟 column 來作為 base 並計算出 target 最小值。

至於 hw3-3，基本上作法與 hw3-2 類似，唯一的差別在於我將 phase 3 做了優化 (因為 profile 以後發現 phase3 占了 8.90%以上的運行時間，所以基本上優化都是由那邊進行的) 也有一部份是因為不需要過多的優化就可以將 hw3-3 全部通過，因此並沒有特別在做優化。優化方法如下 : round 為每個 gpu 需要計算的點數 (行數) 並且會有一個叫做 y_offset 來說從哪裡開始計算。最後，為了確保正確性，在計算之前都必須將 device data 做同步化來確保計算實的正確性。

```
dim3 thread_each_block(16 , BlockSize);
#pragma omp parallel
{
    int cpu_thread_id = omp_get_thread_num();
    cudaSetDevice(cpu_thread_id);
    cudaMalloc(&deviceDist[cpu_thread_id], size_);
    cudaMemcpy(deviceDist[cpu_thread_id], Dist, size_, cudaMemcpyHostToDevice);
    int y_offset = (cpu_thread_id == 1) ? total_block_num / 2 : 0 ;
    int round = (cpu_thread_id == 1) ? total_block_num - y_offset : total_block_num / 2;
    // handling the remainder
    dim3 block_num3(total_block_num, round );
    for(int i = 0 ; i < total_block_num ; i++){
        // printf("it's %d's turn \n",cpu_thread_id);
        // make sure that other thread proform correctly
        if(i >= y_offset && i < y_offset + round ){
            cudaMemcpy(deviceDist[!cpu_thread_id] + i * BlockSize * new_n , deviceDist[cpu_thread_id] + i * BlockSize * new_n , BlockSize * new_n * sizeof(int) , cu
            // printf("in round %d there's an exchange from %d to %d \n" , i ,cpu_thread_id , !cpu_thread_id);
        }
        #pragma omp barrier

        phase1<<<block_num1 , thread_each_block>>>(deviceDist[cpu_thread_id] , i , new_n); // phase 1
        phase2<<<block_num2 , thread_each_block>>>(deviceDist[cpu_thread_id] , i , new_n); // phase 2
        phase3<<<block_num3 , thread_each_block>>>(deviceDist[cpu_thread_id] , i , new_n , y_offset); // phase 3

    }
    #pragma omp barrier
    cudaMemcpy(Dist +  y_offset * BlockSize * new_n , deviceDist[cpu_thread_id] + y_offset  * BlockSize* new_n , round * BlockSize * new_n * sizeof(int) , cudaMemcpy
    // printf("the thread %d has offset %d and have to handle %d raw\n" , cpu_thread_id , y_offset , round);
}
```

我有不少優化，我將在之後一一闡述。

Profiling Results (hw3-2)

Provide the profiling results of following metrics on the biggest kernel of your program using NVIDIA profiling tools. NVIDIA Profiler Guide.

我使用 p22k1 作為 profiling result 的 baseline。

	Min	Max	Average
occupancy	0.924393	0.925843	0.94892
sm efficiency	99.95%	99.97%	99.96%
shared memory load throughput	3232.6GB/s	3307.6GB/s	3272.7GB/s
shared memory store throughput	263.89GB/s	270.01GB/s	267.16GB/s
global load throughput	197.92GB/s	202.50GB/s	200.37GB/s
global store throughput	65.972GB/s	67.502GB/s	66.789GB/s

Experiment & Analysis & Discussion:

Explain how and why you do these experiments? Explain how you collect those measurements? Show the result of your experiments in plots, and explain your observations.

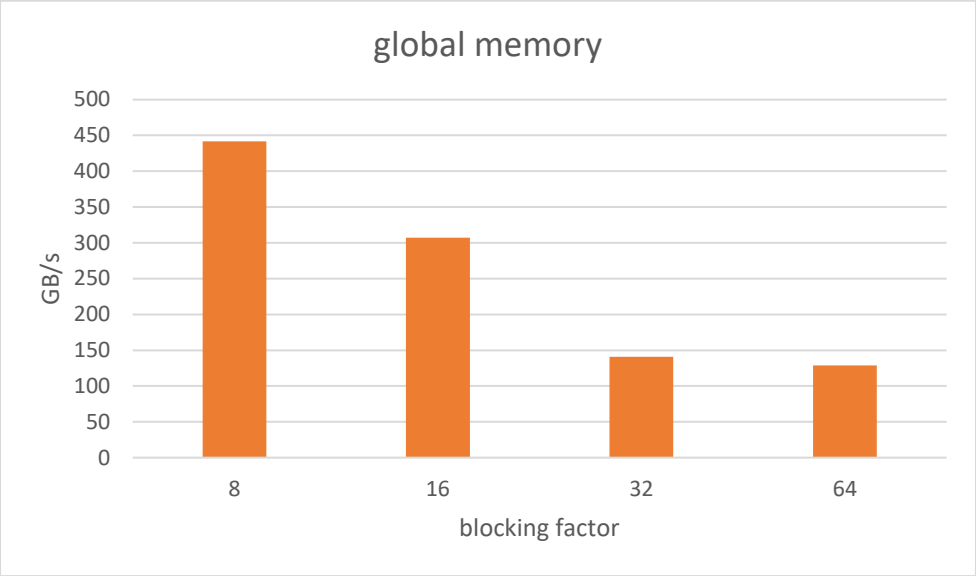
Methodology

1. System Spec (If you run your experiments on your own machine)
   Please specify the system spec by providing the CPU, RAM, storage and network (Ethernet / InfiniBand) information of the system.
   沒有，我使用的是 hades
2. Blocking Factor (hw3-2)

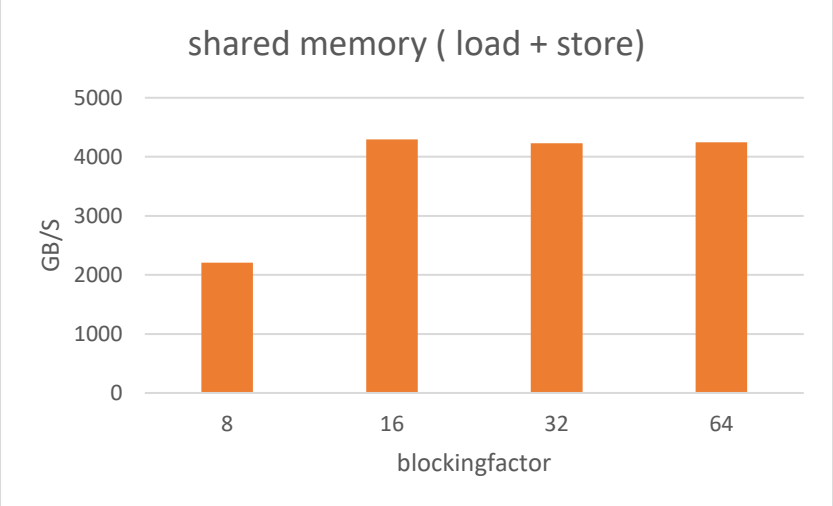
Observe what happened with different blocking factors, and plot the trend in terms of Integer GOPS and global/shared memory bandwidth. (You can get the information from profiling tools or manual) (You might want to check nvprof and Metrics Reference)

以下的圖表我將不會使用我的final version (因為他已經過度優化，無法隨意調整blocksize)
因此 我會將整個結果建立在 version 7 (可以過到p23k1) 而不是最終版本。
而已下的資訊是建立在c19.01

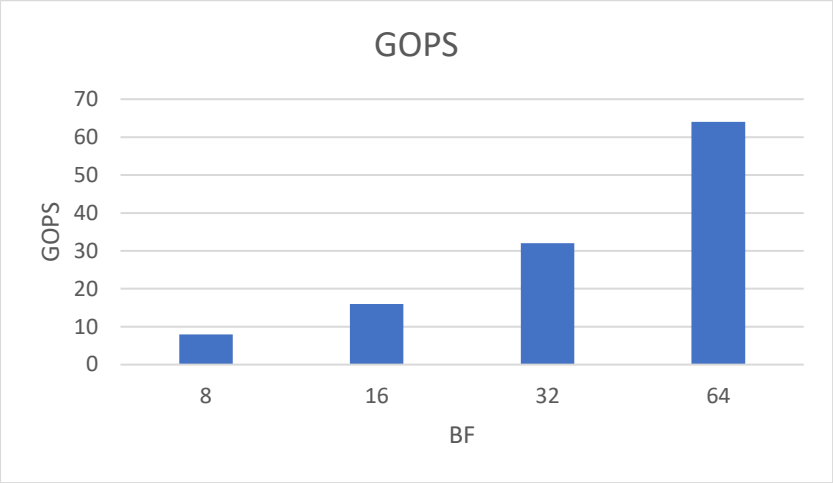
測量方法為先將 inst_int 數量除以 total time。Memory bandwidth 的部分則是將 load/store 的加總在除以時間。



可以看到global memory因為blocking factor增大而遞減。推測是因為主要運算都是在shared memory，而我們在phase3主要只有將global的東西寫入與寫出，而blocking size變大會造成需要讀寫的次數變少進而造就這個結果。



可以看到並沒有很大的差別，這邊想不太太好的解釋方式。可以看到在雖然有增加，但之後就沒有太大的變化，推測是因為這個版本寫得沒有很好造成如此的現象。



但我們還是可以看到GOPS會因為blocking factor增加而增加。主要是因為shared memory的速度真的很快，因此在運算上如果一次般較大的資料過來，並不會成為bottleneck。因此可以看到隨著blocking factor增加而增加。

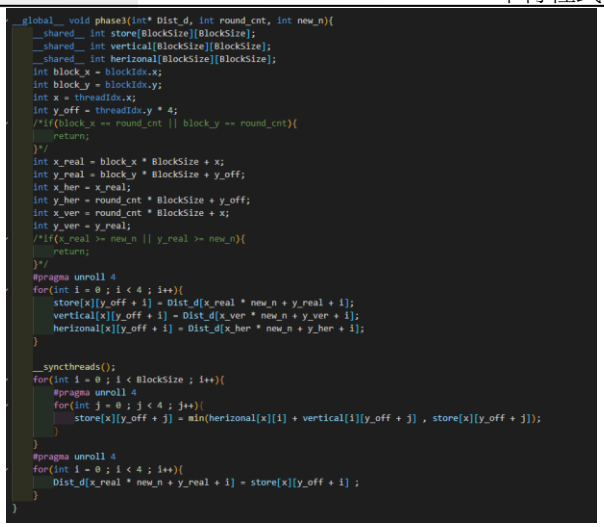
C. Optimization (hw3-2)

1. Shared memory
2. Coalesced memory access
3. No parathesis
4. Large blocking factor

其實主要來說，最好的方式就是shared memory好好調教。這邊就以我自己的版本為例，因為我無法像speculation那樣的寫 (我認為這不是很好劃分 像是shared memory的access pattern算不算一種coalesced memory access ? (bank conflict)) 因此主要以我以下的為主。接下來主要講的phase 3

```
__global__ void phase3(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize][BlockSize];
    __shared__ int vertical[BlockSize][BlockSize];
    __shared__ int herizonal[BlockSize][BlockSize];
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;
    int x = threadIdx.x;
    int y = threadIdx.y;
    if(block_x == round_cnt || block_y == round_cnt){
        return;
    }
    int x_real = block_x * BlockSize + x;
    int y_real = block_y * BlockSize + y;
    int x_her = x_real;
    int y_her = round_cnt * BlockSize + y;
    int x_ver = round_cnt * BlockSize + x;
    int y_ver = y_real;
    if(x_real >= new_n || y_real >= new_n){
        return;
    }
    store[x][y] = Dist_d[x_real * new_n + y_real];
    vertical[x][y] = Dist_d[x_ver * new_n + y_ver];
    herizonal[x][y] = Dist_d[x_her * new_n + y_her];
    __syncthreads();
    #pragma unroll 32
    for(int i = 0 ; i < BlockSize ; i++){
        store[x][y] = min(herizonal[x][i] + vertical[i][y] , store[x][y]);
    }
    Dist_d[x_real * new_n + y_real] = store[x][y] ;
}
```

在一版中，我使用的技巧是沒有很嫻熟的share memory與判斷式。同時在input與output 有使用coalesced memory access (這邊快了1.5倍左右)



```
__global__ void phase3(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize][BlockSize];
    __shared__ int vertical[BlockSize][BlockSize];
    __shared__ int herizonal[BlockSize][BlockSize];
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;
    int x = threadIdx.x;
    int y_off = threadIdx.y * 4;
    /*if(block_x == round_cnt || block_y == round_cnt){
        return;
    }*/
    int x_real = block_x * BlockSize + x;
    int y_real = block_y * BlockSize + y_off;
    int x_her = x_real;
    int y_her = round_cnt * BlockSize + y_off;
    int x_ver = round_cnt * BlockSize + x;
    int y_ver = y_real;
    /*if(x_real >= new_n || y_real >= new_n){
        return;
    }*/
    #pragma unroll 4
    for(int i = 0 ; i < 4 ; i++){
        store[x][y_off + i] = Dist_d[x_real * new_n + y_real + i];
        vertical[x][y_off + i] = Dist_d[x_ver * new_n + y_ver + i];
        herizonal[x][y_off + i] = Dist_d[x_her * new_n + y_her + i];
    }

    __syncthreads();
    for(int i = 0 ; i < BlockSize ; i++){
        #pragma unroll 4
        for(int j = 0 ; j < 4 ; j++){
            store[x][y_off + j] = min(herizonal[x][i] + vertical[i][y_off + j] , store[x][y_off + j]);
        }
    }
    #pragma unroll 4
    for(int i = 0 ; i < 4 ; i++){
        Dist_d[x_real * new_n + y_real + i] = store[x][y_off + i] ;
    }
}
```

在第二版中，我將一個thread需要運算的東西調整為原來的四倍，並同時將if else的statement拿掉 (因為不想要發生branch) 這個版本變快了兩倍。同時為了更好的access memory因此將原本的(32,32) 改為 (16,64) 這樣的話就可以一次讀取 x , x+1,x+2 , x+3 而不是 x , x + 32 , (y + 32) * n + x ...

```
__global__ void phase3(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize][BlockSize];
    __shared__ int vertical[BlockSize][BlockSize];
    __shared__ int herizonal[BlockSize][BlockSize];
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;
    int x = threadIdx.x;
    int y_off = threadIdx.y * 4;
    /*if(block_x == round_cnt || block_y == round_cnt){
        return;
    }*/
    int x_real = block_x * BlockSize + x;
    int y_real = block_y * BlockSize + y_off;
    int x_her = x_real;
    int y_her = round_cnt * BlockSize + y_off;
    int x_ver = round_cnt * BlockSize + x;
    int y_ver = y_real;
    int store_place = x_real * new_n + y_real;

    /*if(x_real >= new_n || y_real >= new_n){
        return;
    }*/
    #pragma unroll 4
    for(int i = 0 ; i < 4 ; i++){
        store[y_off + i][x] = Dist_d[store_place + i];
        vertical[y_off + i][x] = Dist_d[x_ver * new_n + y_ver + i];
        herizonal[y_off + i][x] = Dist_d[x_her * new_n + y_her + i];
    }
    __syncthreads();
    int ans[4] = {store[y_off + 0][x] , store[y_off + 1][x] , store[y_off + 2][x] , store[y_off + 3][x]};
    #pragma unroll 32
    for(int i = 0 ; i < BlockSize ; i++){
        #pragma unroll 4
        for(int j = 0 ; j < 4 ; j++){
            ans[j] = min(herizonal[i][x] + vertical[y_off + j][i] , ans[j]);
        }
    }
    #pragma unroll 4
    for(int i = 0 ; i < 4 ; i++){
        Dist_d[store_place + i] = ans[i];
    }
}
```

這是第五版，因為3.4.5版本中最後總共improved了2~3倍左右，因此這邊統一說明。主要的猜測shared memory access產生了bank conflict。所以就索性對調了一下access pattern(因此我將單純store的行列對調)發現他整體會快了兩倍左右。

```
__global__ void phase3(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize * BlockSize];
    __shared__ int vertical[BlockSize * BlockSize];
    __shared__ int herizonal[BlockSize * BlockSize];
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;
    int x = threadIdx.x;
    int y = threadIdx.y;
    /*if(block_x == round_cnt || block_y == round_cnt){
        return;
    }*/
    int x_real = block_x * BlockSize + x;
    int y_real = block_y * BlockSize + y;
    int x_her = x_real;
    int y_her = round_cnt * BlockSize + y;
    int x_ver = round_cnt * BlockSize + x;
    int y_ver = y_real;
    /*if(x_real >= new_n || y_real >= new_n){
        return;
    }*/
    autogenerate(Dist_d , store , x , y , x_real , y_real , new_n);
    autogenerate(Dist_d , herizonal ,x , y, x_her , y_her , new_n);
    autogenerate(Dist_d , vertical , x , y ,x_ver , y_ver , new_n);

    __syncthreads();
    #pragma unroll 32
    for(int i = 0 ; i < BlockSize ; i++){
        store[y * BlockSize + x] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + x] , store[y * BlockSize + x]);
        store[(y + Half) * BlockSize + x] = min(vertical[(y + Half) * BlockSize + i] + herizonal[i * BlockSize + x] , store[(y + Half)* BlockSize + x]);
        store[y * BlockSize + (x + Half)] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + (x + Half)] , store[y * BlockSize + (x + Half)]);
        store[(y + Half) * BlockSize + (x + Half)] = min(vertical[(y + Half) * BlockSize + i] + herizonal[i * BlockSize + (x + Half)] , store[(y + Half)* BlockSize + (x +
    }

    write_back(Dist_d , store , x , y , x_real , y_real , new_n);
}
```

Version 7這邊是將blockSize改為原本的 (32,32) 來測試效果。發現access解決bank conflict又可以增加兩倍的時間。

```
__global__ void phase3(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize * BlockSize];
    __shared__ int vertical[BlockSize * BlockSize];
    __shared__ int herizonal[BlockSize * BlockSize];
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;
    int x = threadIdx.x * 2;
    int y = threadIdx.y;
    /*if(block_x == round_cnt || block_y == round_cnt){
        return;
    }*/
    int x_real = block_x * BlockSize + x;
    int y_real = block_y * BlockSize + y;
    int x_her = x_real;
    int y_her = round_cnt * BlockSize + y;
    int x_ver = round_cnt * BlockSize + x;
    int y_ver = y_real;
    /*if(x_real >= new_n || y_real >= new_n){
        return;
    }*/
    autogenerate(Dist_d , store , x , y , x_real , y_real , new_n);
    autogenerate(Dist_d , herizonal ,x , y, x_her , y_her , new_n);
    autogenerate(Dist_d , vertical , x , y ,x_ver , y_ver , new_n);

    __syncthreads();
    #pragma unroll 32
    for(int i = 0 ; i < BlockSize ; i++){
        store[y * BlockSize + x] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + x] , store[y * BlockSize + x]);
        store[y * BlockSize + x + 1] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + x + 1] , store[y * BlockSize + x + 1]);
        store[y * BlockSize + x + 32] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + x + 32] , store[y * BlockSize + x + 32]);
        store[y * BlockSize + x + 33] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + x + 33] , store[y * BlockSize + x + 33]);
    }

    write_back(Dist_d , store , x , y , x_real , y_real , new_n);
}
```

Version 9 (差了一筆) 是一直測試access pattern。並發現用(16,64) 可以加速不少 (兩倍) 因此就在這個情況下一直在做input/output的speed up。

```
__global__ void phase3(int* Dist_d, int round_cnt, int new_n){
    __shared__ int store[BlockSize * BlockSize];
    __shared__ int vertical[BlockSize * BlockSize];
    __shared__ int herizonal[BlockSize * BlockSize];
    int x = threadIdx.x ;
    int y = threadIdx.y;
    /*if(block_x == round_cnt || block_y == round_cnt){
        return;
    }*/
    int x_her = blockIdx.x * BlockSize + x;
    int y_her = round_cnt * BlockSize + y;
    int x_ver = round_cnt * BlockSize + x;
    int y_ver =  blockIdx.y * BlockSize + y;
    /*if(x_real >= new_n || y_real >= new_n){
        return;
    }*/
    autogenerate(Dist_d , store , x , y , x_her , y_ver ,new_n  , new_n * 32);
    autogenerate(Dist_d , herizonal ,x , y, x_her , y_her ,new_n  , new_n * 32);
    autogenerate(Dist_d , vertical , x , y ,x_ver , y_ver ,new_n  , new_n * 32);

    __syncthreads();
    #pragma unroll 32
    for(int i = 0 ; i < BlockSize ; i++){
        /*store[y * BlockSize + x] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + x] , store[y * BlockSize + x]);
        store[(y + Half) * BlockSize + x] = min(vertical[(y + Half) * BlockSize + i] + herizonal[i * BlockSize + x] , store[(y + Half)* BlockSize + x]);
        store[y * BlockSize + (x + Half)] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + (x + Half)] , store[y * BlockSize + (x + Half)]);
        store[(y + Half) * BlockSize + (x + Half)] = min(vertical[(y + Half) * BlockSize + i] + herizonal[i * BlockSize + (x + Half)] , store[(y + Half)* B
        store[y * BlockSize + x] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + x] , store[y * BlockSize + x]);
        store[y * BlockSize + x + 32] = min(vertical[y * BlockSize + i] + herizonal[i * BlockSize + x + 32] , store[y * BlockSize + x + 32]);
        store[y * BlockSize + x + 2048] = min(vertical[y * BlockSize + i + 2048] + herizonal[i * BlockSize + x ] , store[y * BlockSize + x + 2048]);
        store[y * BlockSize + x + 2080] = min(vertical[y * BlockSize + i + 2048] + herizonal[i * BlockSize + x + 32] , store[y * BlockSize + x + 2080]);
    }

    write_back(Dist_d , store , x , y , x_her , y_ver ,new_n  , new_n * 32);
}
```

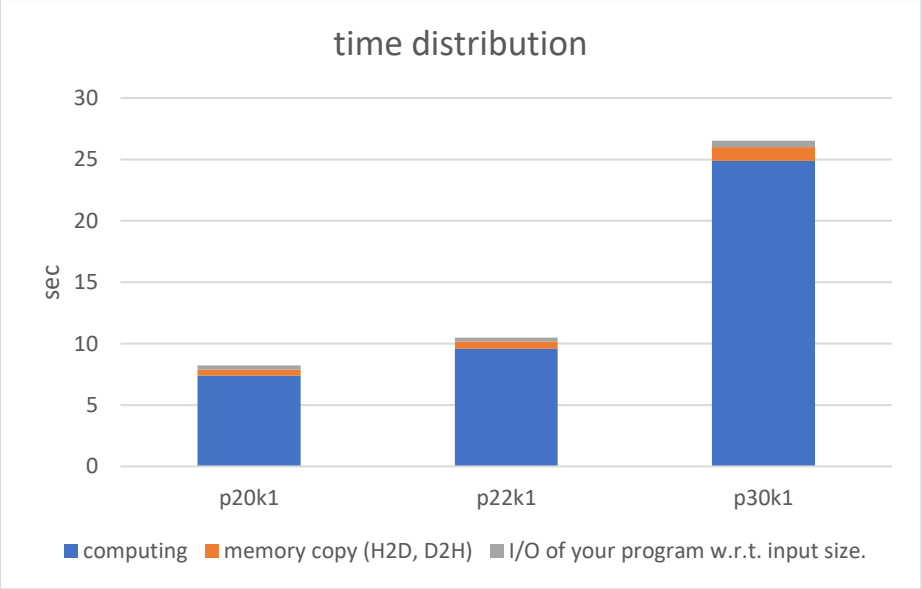
最後一個version最特別，因為我在試的時候發現其實 (x << 6) 會比 x * 64 還要慢。將原本 y * blockSize + x 變成 temp並將全部改成temp不會加速多少。因此嚴重懷疑parathesis在gpu computing裡面造成的速度下降。 因此，就索性將原本很慢的 (32,32) 來使用並直接使用 數字

來代表access block，最終又增快了 1.2 倍。((hurray))

因此我不會對optimization作圖 (因為太難界定甚麼東西會快幾倍了) 。而且在programming中都是可能一次改一小部分，並沒有所謂一次將整個shared memory都改道到完善。

Weak scalability :
我的作法只有將phase 3 分段，因為floyd-warshell是一個dependency很高的算法，因此每次都會需要將資料傳給對方以確保答案的重要性。因此，大概估了一下我的scalability是1.7倍左右。

Time Distribution:



Experience & Conclusion

1. What have you learned from this homework?
   Cuda programming 好像在通靈。有時候改一下就發現突然變快了。有時候改一典認為會變快的東西結果竟然變慢了。同時，要考慮蠻 gpu 硬體架構，這是之前軟體課沒有教過的東西，其實還蠻新奇的。
2. FeedBack : spec 其實用字有點奇怪 像是 block factor 與 block 差異點在哪裡? 還有一些 communication ? 其實不太懂