hw3_report

110590007白宸安

1. Implementation

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

原先是使用了Floyd-Warshall配上OpenMP

```
void floydWarshall() {
    // Floyd-Warshall algorithm
    for (int k = 0; k < n; ++k) {
        #pragma omp parallel for schedule(static)
        for (int i = 0; i < n; ++i) {
            #pragma unroll 4
            for (int j = 0; j < n; ++j) {
                if (Dist[i][k] < INF && Dist[k][j] < INF && Di
                    Dist[i][j] = Dist[i][k] + Dist[k][j];
                }
            }
        }
    }
}
```

後來發現助教有提供Blocked Floyd-Warshall的範例程式，在hw3-2裡，經過測試後發現這個版本的效能叫原本的好，因此改用這個。

主要修改cal函式中，進行Floyd-Warshall的部分，加上OpenMP平行後效果不錯，另外有再加unroll來嘗試減少cache miss。

```
#pragma omp parallel for schedule(static)
for (int i = block_internal_start_x; i < block_internal_end_x;
    // 減少cache miss(int = 8 bytes)
    #pragma unroll 4
    for (int j = block_internal_start_y; j < block_internal_en
        if (Dist[i][k] + Dist[k][j] < Dist[i][j]) {
            Dist[i][j] = Dist[i][k] + Dist[k][j];
```

```
        }
    }
}
```

b. How do you divide your data in hw3-2, hw3-3?

hw3-2和hw3-3都使用同樣的方法，先計算N，根據以下算式。

```
if (n % B) N = n + (B - n % B);
else N = n;

Dist = (int*)malloc(N*N*sizeof(int));
```

如果點的數量n能被B整除，則N=n，否則就多開一塊空間給N，這樣就能避免後續在GPU中計算時還要用if去判斷範圍的問題，能增加效率。這邊是用動態的方式去malloc Dist。

接著在block_FW中，依照以下方式切block。BLOCK_SIZE=32，代表一個block使用32 * 32 = 1024個threads。blocks代表Dist matrix一行的大小，如果有多出來沒辦法被B整除的部分，會再多開一塊來包住。

```
int blocks = (N + B - 1) / B;

dim3 block_dim(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid_dim(blocks, blocks);
```

phase1因為是計算pivot的block，所以只用了1 * 1024 threads。

phase2是計算在pivot的column和row上的所有matrix，所以用了blocks * 1024 threads去計算。

phase3是計算剩下的所有matrix，要開足夠容納整張Dist matrix的大小，所以用blocks * blocks * 1024 threads。

```
phase1<<<1, block_dim>>>(dst, r, N);

phase2_1<<<blocks, block_dim, 0, stream1>>>(dst, r, N);
```

```
phase2_2<<<blocks, block_dim, 0, stream2>>>(dst, r, N);

phase3<<<grid_dim, block_dim>>>(dst, r, N);
```

c. What’s your configuration in hw3-2, hw3-3? And why? (e.g. blocking factor, #blocks, #threads)

Blocking factor B = 64.

blocks = (N + B - 1) / B

BLOCK_SIZE = 32

threads → block_dim(BLOCK_SIZE, BLOCK_SIZE);

一個CUDA block用滿32 * 32 = 1024個threads。

d. How do you implement the communication in hw3-3?

使用了GPU devices之間的memory copy，讓資料直接複製到另一個GPU，以避免還要經過CPU來傳輸資料。hw3-3中有使用OpenMP，共有兩個CPU threads，一個thread控制一個GPU。

把資料從host搬到個別的GPU，並開啟GPU之間的溝通。

```
cudaMemcpy(dst[thread_id] + dist_offset, Dist + dist_offset, t
cudaDeviceEnablePeerAccess(peer_thread_id, 0);
```

接著在後續計算phase的過程中，因為整個程式主要的計算工作都是在phase3發生，所以讓phase1和phase2都做一樣的計算，而phase3時兩個GPU一人各算一半的資料。因為GPU只需要確保自己負責的範圍是對的就好，過程中每次只需要傳送一段B * N（相當於一個row）大小的資料就好，也就是phase3中計算的所有block(64, 64)中，最上面的那一row，因為下一個round時，另一個GPU需要這段row的資料來計算phase1和phase2。

最後兩個GPU分別把自己範圍的資料copy回host上。

```
for (int r = 0; r < round; ++r) {
        unsigned int start_offset_num = r * B * N;
        if (r >= start_offset && r < (start_offset + total_row
            cudaMemcpy(dst[peer_thread_id] + start_offset_num,
        }
        #pragma omp barrier
        phase1<<<1, block_dim>>>(dst[thread_id], r, N);

        phase2_1<<<blocks, block_dim, 0, stream1>>>(dst[thread
            phase2_2<<<blocks, block_dim, 0, stream2>>>(dst[th
            cudaStreamSynchronize(stream1);
            cudaStreamSynchronize(stream2);

        phase3<<<grid_dim, block_dim>>>(dst[thread_id], r, N,
}
cudaMemcpy(Dist + dist_offset, dst[thread_id] + dist_offset, t
```

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

hw3-1

CPU版的實作比較沒有太多東西可以說，大致上就是使用了助教提供的範例程式，然後在主要計算的部分加上OpenMP來優化for loop。

```
// 這邊用openMP來平行
#pragma omp parallel for schedule(static)
for (int i = block_internal_start_x; i < block_internal_end_x;
    // 減少cache miss(int = 8 bytes)
    #pragma unroll 4
    for (int j = block_internal_start_y; j < block_internal_en
        if (Dist[i][k] + Dist[k][j] < Dist[i][j]) {
            Dist[i][j] = Dist[i][k] + Dist[k][j];
        }
    }
}
```

hw3-2

分成五個部分來說明，分別有input、phase1、phase2、phase3、output

input :

input的部分額外使用了memory map（mmap）的方法，將檔案file map到memory上，這樣讀取的速度就能夠變快一點。

```
int file = open(infile, O_RDONLY);
int *ft = (int*)mmap(NULL, 2*sizeof(int), PROT_READ, MAP_PRIVA
int *pair = (int*)(mmap(NULL, (3 * m + 2) * sizeof(int), PROT_
Dist = (int*)malloc(N*N*sizeof(int));
```

而為了讓之後GPU內部的計算不需要在if來判斷範圍，如果點的數量有多出來，就會多開一塊空間，空的地方就做空運算。

```
if (n % B) N = n + (B - n % B);
else N = n;
```

phase1 :

phase1是計算pivot matrix，這邊的實作為了善用shared memory，所以一次會去計算四個點的資料。一個GPU block是32*32，但一次要計算的matrix block是64*64，所以一次要計算四個點。

shared memory總共會用到64 * 64 * 4 = 16384 bytes。

一開始先去得x和y的threadId，並計算y_offset和x_offset（offset=32），是用來access (0, 0)以外的點的資料，總共會access到(0, 0), (0, 32), (32, 0), (32, 32)，分別對應到64*64的大block中，左上、右上、左下、右下這四個32*32的小block。

每個GPU thread會分別把自己需要access到的四個點資料都存進shared memory，而當所有的GPU thread都存好點後，就會得到一張完整64*64大block的shared memory，這時在__syncthreads()。

接下來就是讓每個thread去各自做Floyd-Warshall的計算，每個thread算完4個點後要先__syncthreads()，確保所有thread都算完，最後再將算出的結果寫到GPU global memory 的dst上。

```
__global__ void phase1(int *dst, int Round, int N) {
  int y = threadIdx.y; // y軸 = row
    int y_offset = y + offset;

    int x = threadIdx.x; // x軸 = column
    int x_offset = x + offset;

    // y => 0~31
  // x => 0~31
    __shared__ int s[B][B];

    // 因為最多只能用1024(32 * 32)個threads，但要算(64 * 64)大小的b
    // 且又要盡量用shared memory
    // 所以讓一個thread算4個點的資料。
    int top_left = Round * B * (N + 1) + y * N + x;
  s[y][x] = dst[top_left];

    int top_right = Round * B * (N + 1) + y * N + x + offset;
    s[y][x_offset] = dst[top_right];

    int bottom_left = Round * B * (N + 1) + (y + offset) * N +
    s[y_offset][x] = dst[bottom_left];

    int bottom_right = Round * B * (N + 1) + (y + offset) * N
    s[y_offset][x_offset] = dst[bottom_right];

    __syncthreads();

    for (int k = 0; k < B; ++k) {
        s[y][x] = Min(s[y][k] + s[k][x], s[y][x]);
        s[y][x_offset] = Min(s[y][k] + s[k][x_offset], s[y][x_
        s[y_offset][x] = Min(s[y_offset][k] + s[k][x], s[y_off
        s[y_offset][x_offset] = Min(s[y_offset][k] + s[k][x_of
        __syncthreads();
    }
```

```
    dst[top_left] = s[y][x];
    dst[top_right] = s[y][x_offset];
    dst[bottom_left] = s[y_offset][x];
    dst[bottom_right] = s[y_offset][x_offset];
}
```

Phase2 :

phase2是計算在pivot的column和row上的所有matrix。這邊分別用phase2-1來算column，phase2-2來算row。為了讓計算column和row的部分可以變快，所以使用stream來平行處理兩邊的計算，因為他們資料彼此不會交互影響，一個是column上的，一個是row上的。

```
// Create CUDA streams
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

phase2_1<<<blocks, block_dim, 0, stream1>>>(dst, r, N);
phase2_2<<<blocks, block_dim, 0, stream2>>>(dst, r, N);

// Synchronize with both streams
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

以phase2-1為例：

因為是要算整個column的所有matrix block，所以呼叫時要開blocks個block，每個block為32*32。

```
int blocks = (N + B - 1) / B;
phase2_1<<<blocks, block_dim, 0, stream1>>>(dst, r, N);
```

首先如果是空的就不用算（因為input時會多開空間）。

接著將

1. pivot matrix的四個點
2. 自己這個block在算的這個column matirx的四個點

load進shared memory，兩者matrix都有左上、右上、左下、右下四個點。所以共8個點。

一個thread會access到8個shared memory點。

因為shared memory是在一個GPU block中共享，所以一個GPU block（32*32）負責的就是一個column matrix block（64*64）。

之後一樣進行

1. __syncthreads()
2. Floyd-Warshall
3. 寫進global memory dst中

```
__global__ void phase2_1(int *dst, int Round, int N) {
    if (blockIdx.x == Round) return;
    int y = threadIdx.y;
    int y_B = y + offset;

    int x = threadIdx.x;
    int x_B = x + offset;

    __shared__ int s[B][B];
    __shared__ int col[B][B];

  // 要算跟pivot B有row或col相同的所有B
  // 一樣，每個thread要算4個點
  // 算col的matrix B

    int main_top_left = Round * B * (N + 1) + y * N + x;
  s[y][x] = dst[main_top_left];
  int col_top_left = blockIdx.x * B * N + Round * B + y * N +
    col[y][x] = dst[col_top_left];

    int main_top_right = Round * B * (N + 1) + y * N + x + off
  s[y][x_B] = dst[main_top_right];
    int col_top_right = blockIdx.x * B * N + Round * B + y * N
```

```
    col[y][x_B] = dst[col_top_right];


    int main_bottom_left = Round * B * (N + 1) + (y + offset)
  s[y_B][x] = dst[main_bottom_left];
    int col_bottom_left = blockIdx.x * B * N + Round * B + (y
    col[y_B][x] = dst[col_bottom_left];


    int main_bottom_right = Round * B * (N + 1) + (y + offset)
  s[y_B][x_B] = dst[main_bottom_right];
    int col_bottom_right = blockIdx.x * B * N + Round * B + (y
    col[y_B][x_B] = dst[col_bottom_right];

    __syncthreads();

    for (int k = 0; k < B; ++k) {
        col[y][x] = Min(col[y][x], col[y][k] + s[k][x]);
        col[y][x_B] = Min(col[y][x_B], col[y][k] + s[k][x_B]);
        col[y_B][x] = Min(col[y_B][x], col[y_B][k] + s[k][x]);
        col[y_B][x_B] = Min(col[y_B][x_B], col[y_B][k] + s[k][
        __syncthreads();
    }
    dst[col_top_left] = col[y][x];
    dst[col_top_right] = col[y][x_B];
    dst[col_bottom_left] = col[y_B][x];
    dst[col_bottom_right] = col[y_B][x_B];
}
```

phase2-2依此類推，只是是算row matrix block。

Phase3

phase3是計算剩下的所有matrix。

phase3中需要取用到的點是

1. 自己這個block計算到這個target matrix

2. target matrix對應到的column matrix
3. target matrix對應到的row matrix

一樣每個matrix要取用4個點，所以總共會有12個點，代表12個shared memory access。

此時的shared memory 有 3 * B(=64) * B(=64) * 4(=int size) = 49152，而因為GTX 1080上的shared memory最多只有70000多，所以會選擇B=64正是因為在phase3的shared memory大小會被bound住，所以就讓一個GPU block(32*32)算一個Matrix block(64*64)，也就是一個GPU thread 算4個點。

之後一樣進行

1. __syncthreads()
2. Floyd-Warshall
3. 寫進global memory dst中

```
__global__ void phase3(int *dst, int Round, int N) {
    if (blockIdx.x == Round || blockIdx.y == Round) return;
    int y = threadIdx.y;
    int y_B = y + offset;

    int x = threadIdx.x;
    int x_B = x + offset;

    __shared__ int col[B][B];
    __shared__ int row[B][B];
    __shared__ int target[B][B];

  int target_top_left = blockIdx.y * B * N + blockIdx.x * B +
  target[y][x] = dst[target_top_left];
  int col_top_left = blockIdx.y * B * N + Round * B + y * N +
  col[y][x] = dst[col_top_left];
  int row_top_left = Round * B * N + blockIdx.x * B + y * N +
  row[y][x] = dst[row_top_left];

  int target_top_right = blockIdx.y * B * N + blockIdx.x * B +
  target[y][x_B] = dst[target_top_right];
  int col_top_right = blockIdx.y * B * N + Round * B + y * N +
```

```
  col[y][x_B] = dst[col_top_right];
  int row_top_right = Round * B * N + blockIdx.x * B + y * N +
  row[y][x_B] = dst[row_top_right];

  int target_bottom_left = blockIdx.y * B * N + blockIdx.x * B
  target[y_B][x] = dst[target_bottom_left];
  int col_bottom_left = blockIdx.y * B * N + Round * B + (y +
  col[y_B][x] = dst[col_bottom_left];
  int row_bottom_left = Round * B * N + blockIdx.x * B + (y +
  row[y_B][x] = dst[row_bottom_left];

  int target_bottom_right = blockIdx.y * B * N + blockIdx.x *
  target[y_B][x_B] = dst[target_bottom_right];
    int col_bottom_right = blockIdx.y * B * N + Round * B + (y
  col[y_B][x_B] = dst[col_bottom_right];
  int row_bottom_right = Round * B * N + blockIdx.x * B + (y +
    row[y_B][x_B] = dst[row_bottom_right];

    __syncthreads();

    for (int k = 0; k < B; ++k) {
        target[y][x] = Min(col[y][k] + row[k][x], target[y][x]
        target[y][x_B] = Min(col[y][k] + row[k][x_B], target[y
        target[y_B][x] = Min(col[y_B][k] + row[k][x], target[y
        target[y_B][x_B] = Min(col[y_B][k] + row[k][x_B], targ
    }
    dst[target_top_left] = target[y][x];
    dst[target_top_right] = target[y][x_B];
    dst[target_bottom_left] = target[y_B][x];
    dst[target_bottom_right] = target[y_B][x_B];
}
```

output :

等算完phase1、phase2、phase3的所有round後，GPU中的dst即為最後的答案，接著將它搬回CPU的Dist中。

```
// GPU算完搬回CPU
cudaMemcpy(Dist, dst, size, cudaMemcpyDeviceToHost);

// 清掉dst
cudaFree(dst);
```

並用output function將答案寫出。

```
void output(char* outFileName) {
    FILE* outfile = fopen(outFileName, "w");
    #pragma unroll 32
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (Dist[i*N + j] >= INF) Dist[i*N + j] = INF;
        }
        fwrite(&Dist[i*N], sizeof(int), n, outfile);
    }
    fclose(outfile);
}
```

hw3-3

兩個GPU一樣都會算phase1和phase2，雖然會有重複計算到的部分，但因為整個程式其實最大宗的計算都在phase3（90%以上），所以phase1和phase2就不另外做平行。

2 GPUs版本的實作大致上和1 GPU版本的一樣，只差在

1. 使用OpenMP，每個thread管理一個GPU。
2. 分配memory時會分給兩個GPU
3. 要開啟memory access給對方GPU
4. 每個round算完後兩個GPU要交換一部份資料

另外，也要確保動作間要synchronous，避免某一個GPU提早完成而搶先做了下一步，可能導致錯誤。

1. 使用OpenMP，每個thread管理一個GPU。

```
#pragma omp parallel num_threads(2)
```

2. 分配memory時會分給兩個GPU

使用start_offset來計算兩個GPU的起始位置。

total_row是一半的資料量。

```
unsigned int thread_id = omp_get_thread_num();
unsigned int peer_thread_id = !thread_id;

unsigned int start_offset = (thread_id == 1) ? round / 2 : 0;
unsigned int total_row = round / 2;
if (round % 2 == 1 && thread_id == 1) total_row += 1;

dim3 grid_dim(blocks, total_row);

unsigned int dist_offset = start_offset * N * B;
unsigned int total_byte_num = total_row * N * B * sizeof(int);
unsigned int one_row_byte_num = B * N * sizeof(int);

cudaSetDevice(thread_id);
cudaMalloc(&dst[thread_id], N*N*sizeof(int));
cudaMemcpy(dst[thread_id] + dist_offset, Dist + dist_offset,
    total_byte_num, cudaMemcpyHostToDevice);
```

另外在phase3一開始確定資料範圍的部分，加上判斷start_offset的部分，確保自己只有計算自己那一半的資料。

```
__global__ void phase3(int *dst, int Round, int N, int row_off
    if (blockIdx.x == Round || blockIdx.y + row_offset == Roun
```

3. 要開啟memory access給對方GPU

```
cudaDeviceEnablePeerAccess(peer_thread_id, 0);

cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

#pragma omp barrier
```

4. 每個round算完後兩個GPU要交換一部份資料

```
if (r >= start_offset && r < (start_offset + total_row)) {
    cudaMemcpy(dst[peer_thread_id] + start_offset_num, dst[thr
        start_offset_num, one_row_byte_num, cudaMemcpyDefault)
}
#pragma omp barrier
```

2. Profiling Results (hw3-2)

使用p15k1這筆測資，在hades上使用nvprof來profile phase3這個kernel function。

指令如下：

srun -p prof -N1 -n1 --gres=gpu:1 nvprof --metrics <metric name> ./hw3-2 ~/hw3-2/cases/p15k1 p15k1.out

Metrics	Min	Max	Avg
Occupancy	0.934671	0.937816	0.937300
sm_efficiency	99.93%	99.96%	99.94%
Shared Memory Load Throughput	2877.7GB/s	3256.8GB/s	3231.7GB/s
Shared Memory Store Throughput	235.72GB/s	265.02GB/s	262.64GB/s
Global Load Throughput	16.971GB/s	19.060GB/s	18.860GB/s
Global Store Throughput	65.504GB/s	66.331GB/s	65.785GB/s

3. Experiment & Analysis

a. System Spec

使用上課提供的hades server。

b. Blocking Factor (hw3-2)

實驗要測試phase3下，不同blocking factor對integer GOPS、global memory bandwidth、shared memory bandwidth造成的影響。

一樣使用p15k1的測資進行實驗。

為了讓程式能適應不同的blocking factor，實驗時除了改動B的大小，還會去改對應的BLOCK_SIZE和offset值，以避免程式error。

分別調整了以下參數：

B = 64, BLOCK_SIZE = 32, offset = 32

B = 32, BLOCK_SIZE = 16, offset = 16

B = 16, BLOCK_SIZE = 8, offset = 8

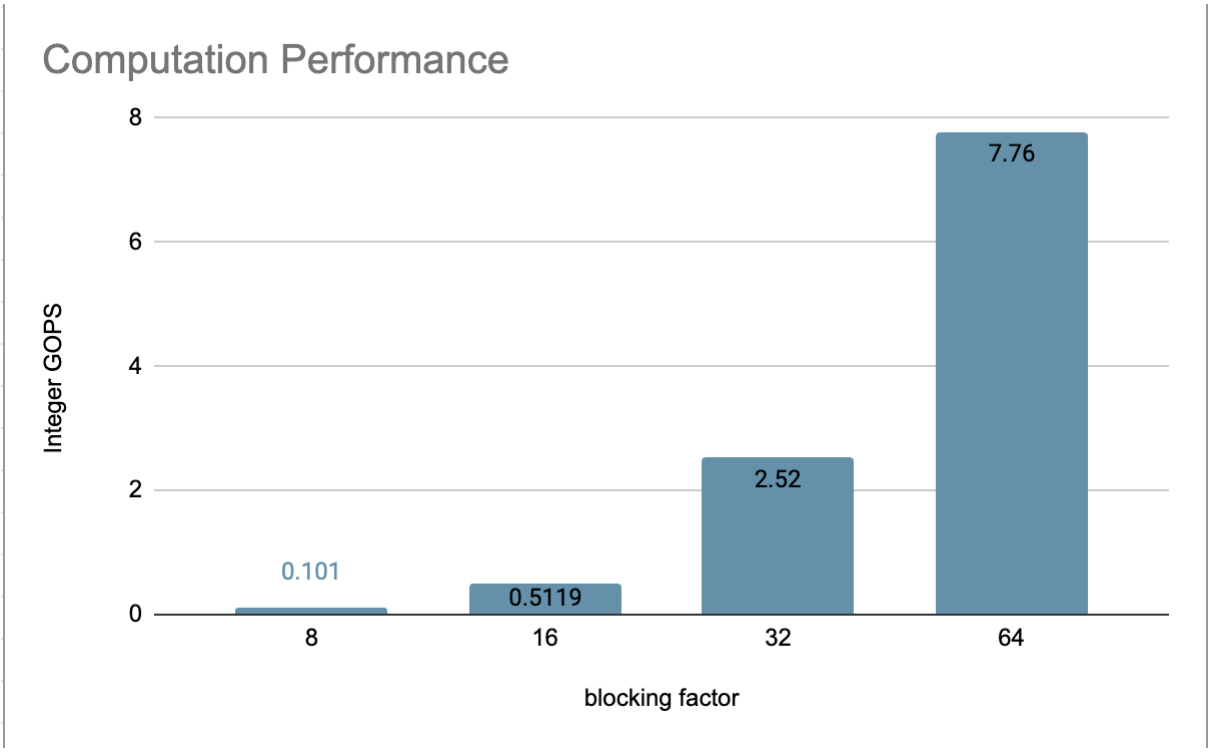
B = 8, BLOCK_SIZE = 4 offset = 4

Integer GOPS :

從圖中可以看出，在Blocking factor設為64時有最好的performance，但無法再往上增加，因為如果加到128時phase3的shared memory會爆掉。

Integer GOPS計算方式是拿 inst_integer總數 / phase3執行時間。

	8	16	32	64
GOPS	0.0101e+10	0.05119e+10	0.252e+10	0.776e+10

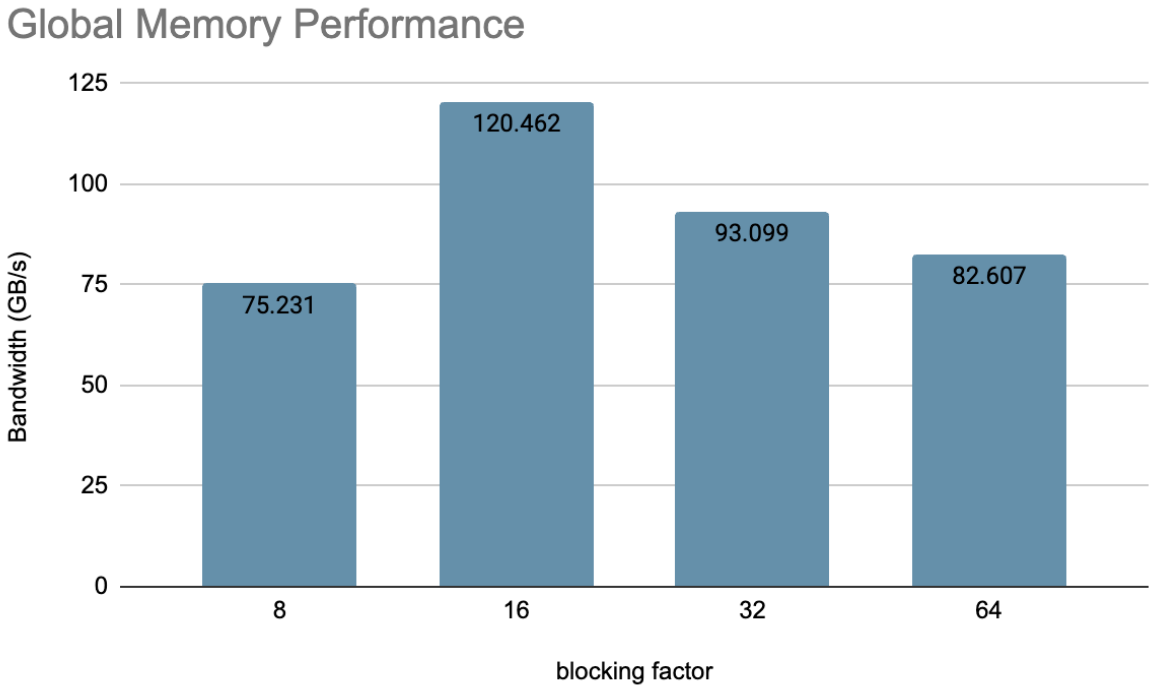


Global Memory Bandwidth :

Global bandwidth的計算方式是把global load和global store的值相加。

從圖中可以看到，不同blocking factor造成的global bandwidth大致上還算相近。我認為原因在於phase3的實作上，無論一個GPU block中使用多shared memory還是少shared memory，所有GPU block加總起來要對global memory的access次數都是接近的，因為會去access到global memory的行為只有將它讀進shared memory還有最後把資料寫回global memory dst的行為，而這個行為是取決於資料點的數量，跟blocking factor較沒有相關。

	8	16	32	64
Global Bandwidth(GB/s)	75.231	120.462	93.099	82.607



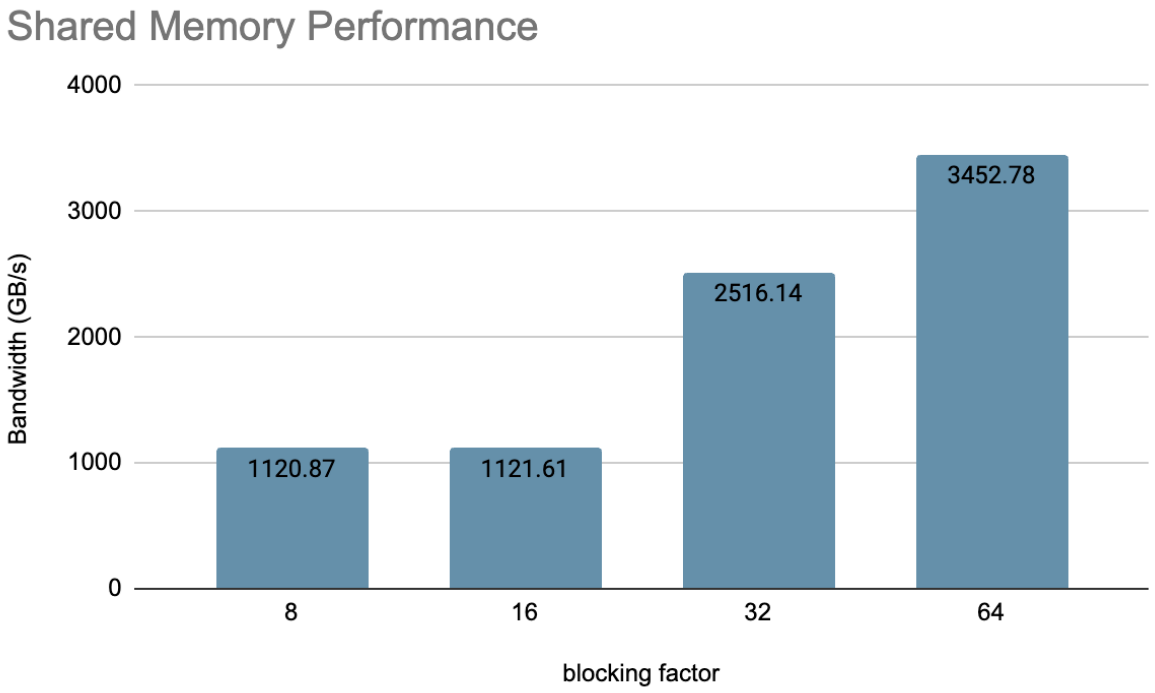
Shared memory Bandwidth

Shared bandwidth的計算方式是把shared load和shared store的值相加。

從圖中可以看出，大致上使用月大的blocking factor，shared memory的bandwidth越好，這是因為shared memory的使用率越高，64時會用到40000多的shared memory，而8時只會用到幾千的shared memory。

另外也能發現，shared memory的bandwidth比global memory的還要快上接近100倍，由此可見善用shared memory確實能對GPU performance做很大的優化。

	8	16	32	64
Shared Bandwidth(GB/s)	1120.87	1121.61	2516.14	3452.78



c. Optimization (hw3-2)

1. coalesced memory : 20.712s

把4個點的memory access排列好，排成可以連續存取的樣子，這樣就不會每次都需要去去global memory搬資料，可以善用cache。一開始實作時就有弄成coalesced memory，所以直接拿這個當作GPU baseline。

2. shared memory : 9.397s

把thread會用到的資料都先從global memory搬到shared memory上，因為shared memory的access速度比global memory快很多，這樣就能省下很多時間，提升performance。

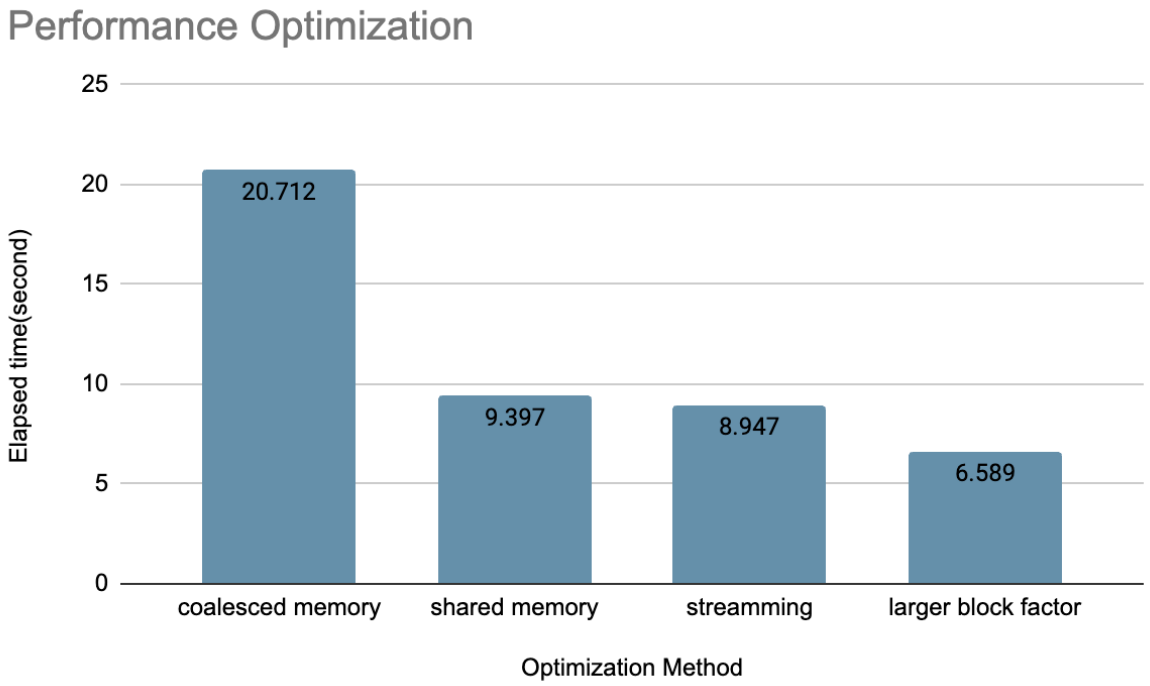
3. 2 stream : 8.947s

原先的column和row matrix是分開計算，要兩次kernel呼叫。後來發現兩者之間計算沒有任何data dependency，所以使用stream去將兩個平行，算完在呼叫stream synchronize。

4. larger block factor : 6.589s

使用B=64時能最大化shared memory的使用量。

注：原先CPU版本在跑實驗測資p15k1時，會超出hades slurm的時間上限（5分鐘），無法正確知道花了多久，因此圖表中就不放原CPU的時間，只能知道GPU的優化performance是高出CPU版非常多的。

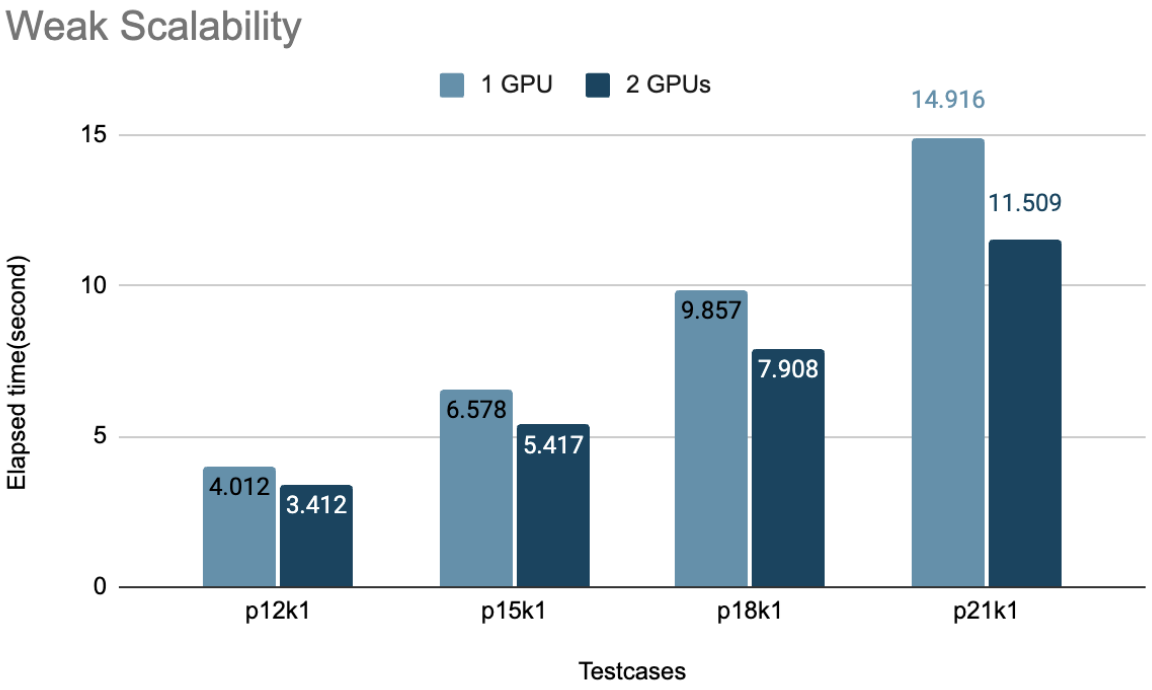


d. Weak scalability (hw3-3)

選用p12k1, p15k1, p18k1, p21k1這四筆測資進行scalability實驗，觀察在使用1 GPU和2 GPUs時效能上的差異。

從圖中可以發現，使用2 GPUs時確實能加快一些計算時間，但效果並沒有到完全砍半，原因在於使用2 GPUs時還需要進行GPU間的溝通，我認為這會是主要的bottleneck。另外，也因為實作上會讓phase1和phase2被兩GPU重複計算，雖然這兩phase的計算比例很低（總時間比例的10%以下)，但或許也會有些影響。

	p12k1	p15k1	p18k1	p21k1
1 GPU	4.012s	6.578s	9.857s	14.916s
2 GPU	3.412s	5.417s	7.908s	11.509s



e. Time Distribution (hw3-2)

選擇不同input size的testcase去進行實驗，分別計算computing、communication（同memcpy）、I/O的時間。

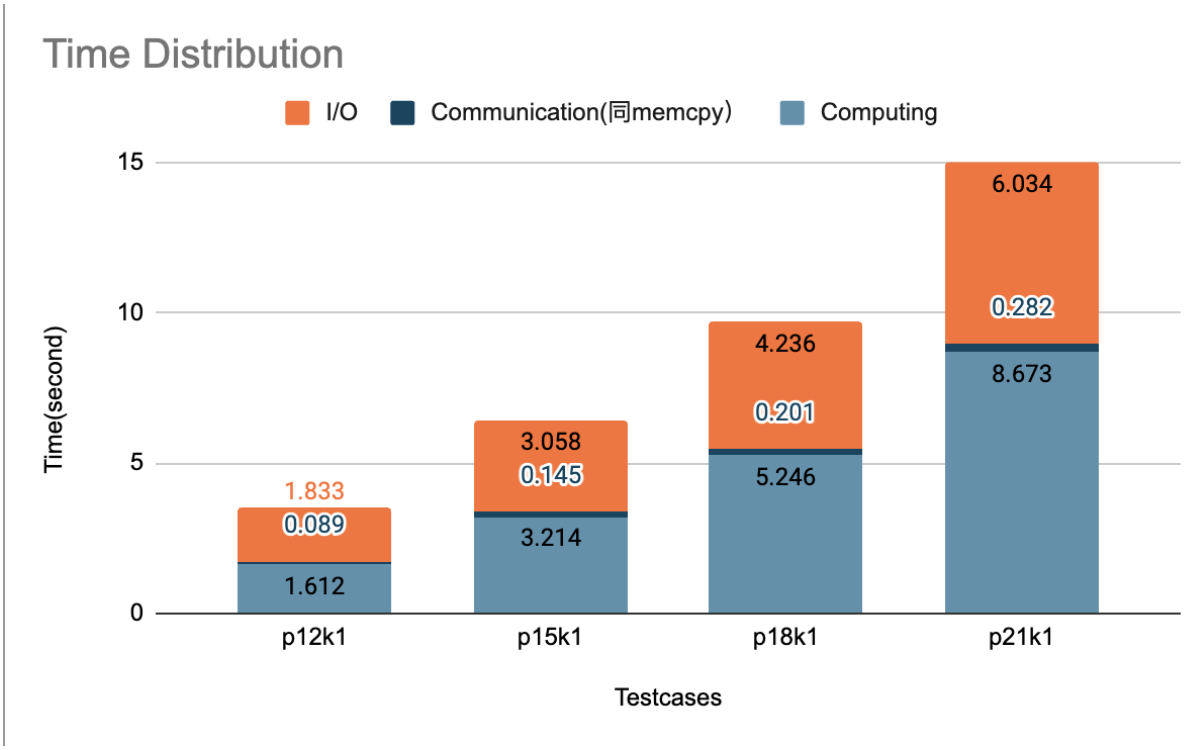
computing和communication是使用nvprof測量，而I/O是用clock_gettime的方式測量。

從圖中可以看到三種行為所花費的時間，大致上的組成分佈相同。communication的時間佔比很少，能看出這支程式並不是communication bound。

值得一提的是I/O time主要都是花在output上面，比如1.883s的I/O中，input可能只佔了0.2s，而剩下的都是output。

基本上可以看出，這支程式的bottleneck會是在output檔案和phase3的計算上，因此如果要再優化這支程式，從output或phase3下手會是個好選擇。

	p12k1	p15k1	p18k1	p21k1
Computing	1.612	3.214	5.246	8.673
Communication(和memcpy一樣)	0.089	0.145	0.201	0.282
I/O	1.833	3.058	4.236	6.034



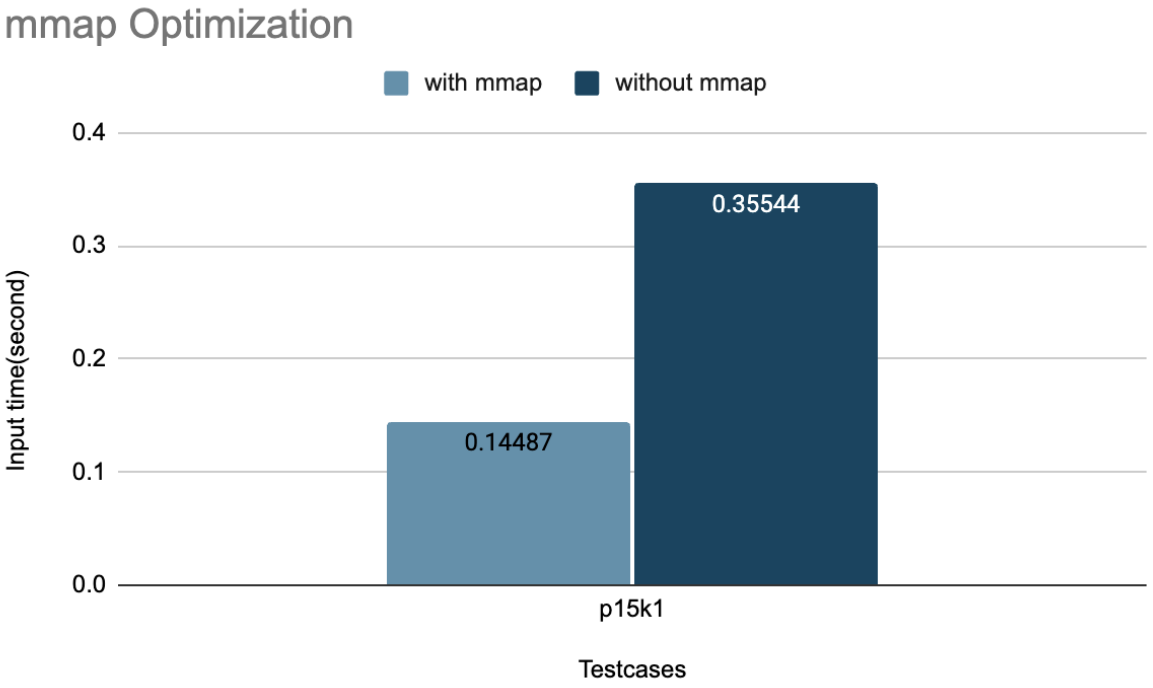
f. Others

mmap optimization

在Input的時候有使用了memory map的方式來進行優化，發現有稍微加快一點讀取的速度。

因為會將file直接map到memory上，這樣access file時就不用再透過system call去access disk。

	p15k1
with mmap	0.144870s
without mmap	0.355440s



4. Experience & conclusion

a. What have you learned from this homework?

這次作業學到了很多，讓我對CUDA programming和GPU架構等有了很多的理解，也讓我學到很多在寫GPU平行程式時的技巧。也透過這次作業了解GPU的強大，發現平行效率竟然能增加這麼多覺得很驚訝。雖然GPU加速很多，但也有代價那便是GPU的平行程式其實很難開發，不僅要先了解GPU的架構，熟悉CUDA語法，還要先知道一些優化技巧比如shared memory、coalesced memory等，要達成這些才能有很好的效率，此外在debug時也非常困難，因為平行開出來的thread數量非常多，很難一個個去trace，有時候甚至不知道程式出錯在哪裡。這次作業，讓我學到很多GPU和CUDA的知識，我認為這些對我以後的發展非常有幫助。

b. Feedback (optional)

感謝助教們在hades機器當機時緊急救援處理，讓我們能順利完成作業。