Parallel Programming

Homework 3: All-Pairs Shortest Path

- 111065524 李懷-

Implementation

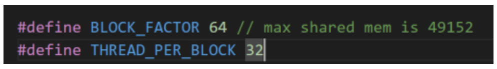
a. Which algorithm do you choose in hw3-1?

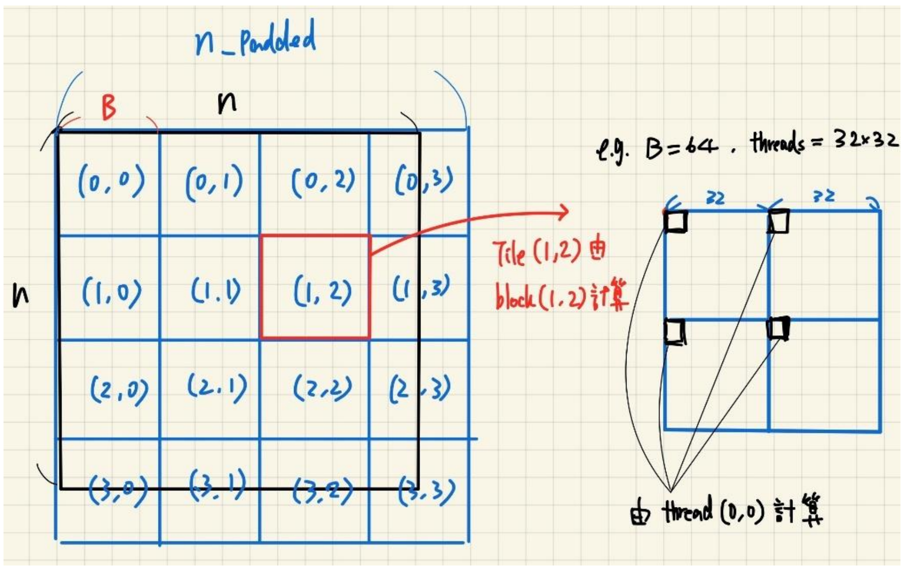
我使用的演算法為 Blocked Floyd-Warshall，針對 CPU 版使用的 threading library 為 openmp，並搭配 vectorization 去處理每個 tile(這裡我用 tile 稱呼避免跟之後 cuda 的 block 搞混)。

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

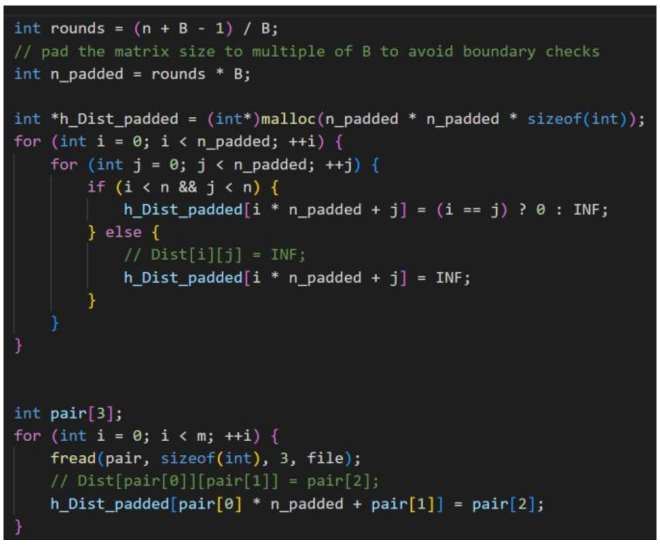
HW3-2(Single GPU)

將整體$n*n$的 vertices 矩陣切分成$rounds*rounds$個 tiles($rounds = \lceil n/BLOCK\_FACTOR \rceil$)，然後將 launch 的 cuda block 一對一 map 到每個 tile，一個 block 設$THREAD_PER_BLOCK * THREAD_PER_BLOCK$個 threads，也就是一個 thread 需負責處理$(BLOCK_FACTOR/THREAD_PER_BLOCK)^2$個 elements，下圖範例為$BLOCK_FACTOR = 64$，$threads = 32 * 32$:


```
#define BLOCK_FACTOR 64 // max shared mem is 49152
#define THREAD_PER_BLOCK 32
```



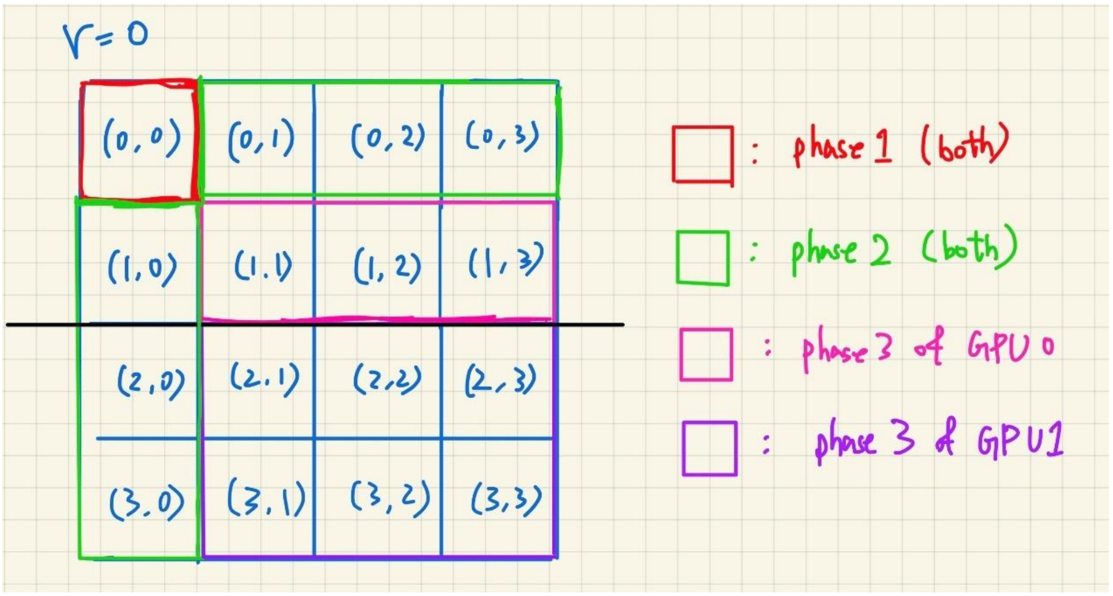
下面是將$n*n$的 data 加 padding，這樣在計算每對 vertices 的 distance 時可以不需顧慮當 n 不整除於 block_factor 時邊緣 tile 的 boundary check，即便處理到超出 boundary 的 data，到時 unpad 回來並不會影響結果:



```
int rounds = (n + B - 1) / B;
// pad the matrix size to multiple of B to avoid boundary checks
int n_padded = rounds * B;

int *h_Dist_padded = (int*)malloc(n_padded * n_padded * sizeof(int));
for (int i = 0; i < n_padded; ++i) {
    for (int j = 0; j < n_padded; ++j) {
        if (i < n && j < n) {
            h_Dist_padded[i * n_padded + j] = (i == j) ? 0 : INF;
        } else {
            // Dist[i][j] = INF;
            h_Dist_padded[i * n_padded + j] = INF;
        }
    }
}


int pair[3];
for (int i = 0; i < m; ++i) {
    fread(pair, sizeof(int), 3, file);
    // Dist[pair[0]][pair[1]] = pair[2];
    h_Dist_padded[pair[0] * n_padded + pair[1]] = pair[2];
}
```

HW3-3(Multi-GPU)

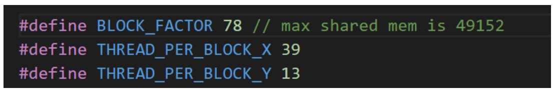
threads 處理 data 的方式跟 single GPU 版的一樣，一樣是一個 block 處理一個 tile，一個 thread 處理 tile 內對應的$(Block_Factor/blockDim.x)$個 datas，tiles 分佈的部分，因為有兩張 GPU，且由於 Blocked Floyd-Warshall 當 vertices 很多時主要的計算都會在 phase3 上，phase1 與 phase2 需要處理的 tiles 相對少，所以與其讓各 GPU 計算自己部份的 phase1 跟 2 後再互傳，增加 communication 成本，直接讓兩張 GPU 都去算當下 round 的 pivot 及 pivot row&column 反而比較快，而在 phase3 時讓 GPU0 負責計算 n*n 上半，GPU1 負責下半的 tiles，這樣的分法各 GPU 只需在 round 剛開始時同步當下的 row，不需每個 phase 算完就同步一次(communication 的細節放在 1-d)。



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

HW3-2(Single GPU)

最一開始我 **block_factor** 是設 64，搭配 32*32 的**#thread**，然後一個 block 一個 tile，所以一個 thread 需處理 4 個 datas，但考量到 GTX1080 的 shared memory per block 有 49152 bytes，**block_factor** 設 64 不能完全利用 shared memory，後來我改設 78 (78*78*2*4=48672 剛好為最大可設的 **block_factor**，*2 是因為 phase2 跟 phase3 計算所需的 data dependent 的 tiles 有兩塊)，相較於 64 更大的 tile 更能利用的 shared memory(詳細比較放在 optimization)，**#threads** 則改設 39*13(為了避免 boundary check，所以須可以整除 78)，代表一個 thread 需處理 12 個 data，這降低了 sm warps 的 occupancy，但同時也提升了 ilp(Instruction-Level Parallelism) (詳細在 Section 2 的 Profiling Results 有說明):



```
#define BLOCK_FACTOR 78 // max shared mem is 49152
#define THREAD_PER_BLOCK_X 39
#define THREAD_PER_BLOCK_Y 13
```

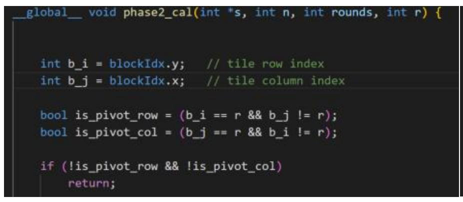
#blocks 則根據 phase 而有所不同:

Phase1 是處理 tile(r, r) (r:第幾個 round)，也就是我們只需要 launch 一個 block；

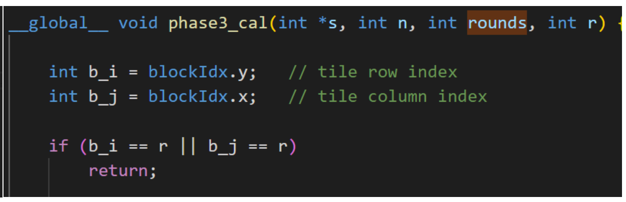
Phase2 是處理以 tile(r, r)為中心的十字上的 tiles，所需的 blocks 數為$2 * (rounds - 1)$；

Phase3 則是處理以除此之外的 tiles，所需的 blocks 數為$(rounds - 1)^2$；

由於我直接讓 block index 與 tile index 對應，所以 phase2 跟 phase3 的 kernel 我是直接 launch $rounds^2$個 blocks，並在 kernel 中 return 掉當下 phase 無需計算的 tile 的 index 即可:

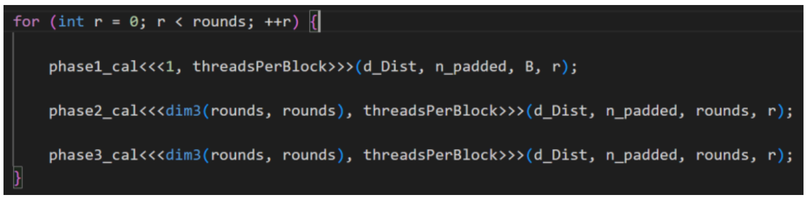


```
__global__ void phase2_cal(int *s, int n, int rounds, int r) {


    int b_i = blockIdx.y;    // tile row index
    int b_j = blockIdx.x;    // tile column index

    bool is_pivot_row = (b_i == r && b_j != r);
    bool is_pivot_col = (b_j == r && b_i != r);

    if (!is_pivot_row && !is_pivot_col)
        return;
```



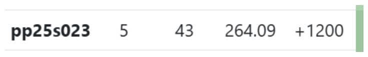
```
__global__ void phase3_cal(int *s, int n, int rounds, int r) {

    int b_i = blockIdx.y;    // tile row index
    int b_j = blockIdx.x;    // tile column index

    if (b_i == r || b_j == r)
        return;
```

Kernel launch 的部分:



```
for (int r = 0; r < rounds; ++r) {

    phase1_cal<<<1, threadsPerBlock>>>(d_Dist, n_padded, B, r);

    phase2_cal<<<dim3(rounds, rounds), threadsPerBlock>>>(d_Dist, n_padded, rounds, r);

    phase3_cal<<<dim3(rounds, rounds), threadsPerBlock>>>(d_Dist, n_padded, rounds, r);
}
```

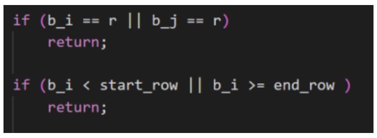
實測後(**block_factor**, **#thread**)設(78, 39*13)能 pass 較多 testcases:



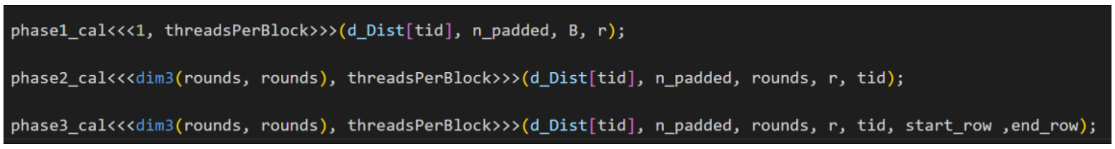
pp25s023	5	43	264.09	+1200

HW3-3(Multi-GPU)

(**block_factor**, **#thread**)跟 hw3-2 一樣設(78, 39*13)，Kernel 一樣是 launch rounds^2 個 blocks，由於兩張 GPU 各只處理一半的 data(我是分成上下兩半)，因此在 phase3 kernel 裡會把在處理範圍外無需計算的 block 擋掉:



```
if (b_i == r || b_j == r)
    return;

if (b_i < start_row || b_i >= end_row )
    return;
```



```
phase1_cal<<<1, threadsPerBlock>>>(d_Dist[tid], n_padded, B, r);

phase2_cal<<<dim3(rounds, rounds), threadsPerBlock>>>(d_Dist[tid], n_padded, rounds, r, tid);

phase3_cal<<<dim3(rounds, rounds), threadsPerBlock>>>(d_Dist[tid], n_padded, rounds, r, tid, start_row ,end_row);
```

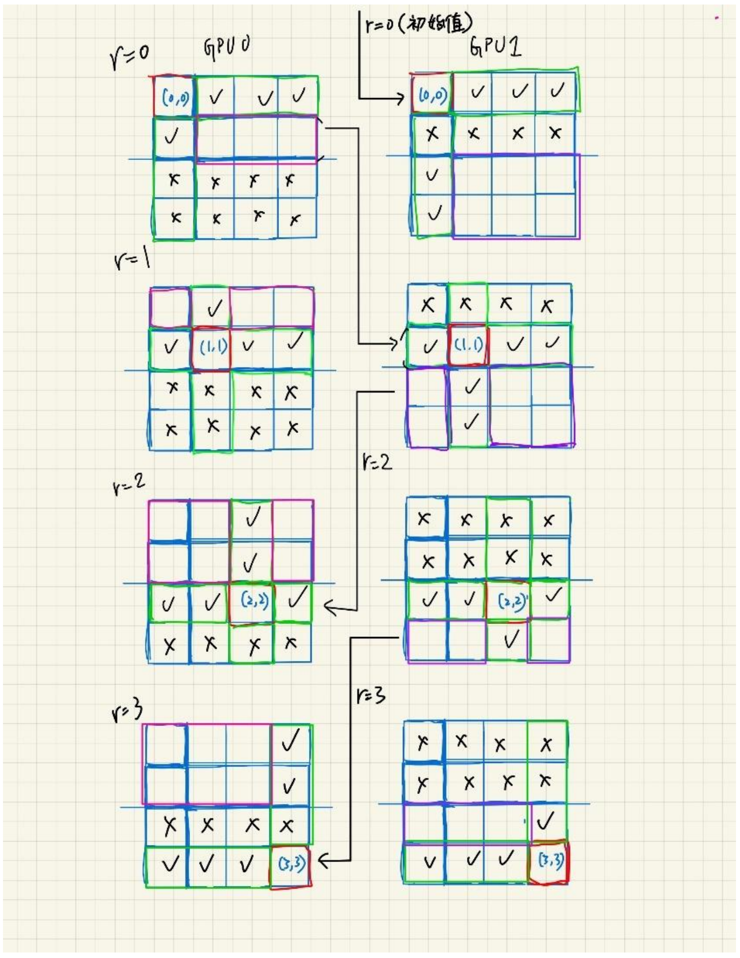
d. How do you implement the communication in hw3-3?

HW3-3(Multi-GPU)

首先是用 openmp create 兩個 threads，一張 GPU map 到一個 thread，data 交換最一開始我是使用 zero-copy 的方式讓多張 GPU 來讀取同段 pinned 的 host memory 來實作，將 phase2 及 phase3 的 tiles 平分給 GPU，各 GPU 在 kernel 中計算完會將自己負責的 tiles 直接寫到 pinned 的 host memory，這樣可以不需要 device 間 cudaMemcpy，且 workload 可以更平均，但由於讀寫 host memory 太慢了，所以效率差，後來改成用將 n*n 的 data 水平切分成兩半，phase3 時 GPU0 處理上半，GPU1 處理下半，並用 cudaMemcpyPeer 的方式在每個 round 開始時同步當下的 pivot row。

各 GPU 在每個 round 計算 phase3 時，只需當下的 pivot row 及其處理部分對應到的 pivot column tiles 即可，例: 在 r=0 時，GPU0 計算上半的 phase3 tiles 需要整條 pivot row，但 pivot column 只需要上半的即可，GPU1 則是只需要整條 pivot row 及下半的 pivot column，且由於每個 round 在計算 pivot 及 pivot row&column 時，需要前一個 round 的 phase3 的結果，所以需在每個 round 的 phase1 前將當下的 row 同步，例如 4*4 個 tiles，共 4 個 rounds，在 r=0,1 時，pivot row 會是在上半，所以 GPU0 會將第 1, 2 列傳給 GPU1 讓 GPU1 可以計算正確的 pivot 及 pivot row&column；當進入到 r=2,3 時，pivot row 會是在下半，所以 GPU1 會將第 3,4 列傳給 GPU0，透過這種方式能確保，每個 round 各 GPU 會用到的 tile 都為最新 update 的，最後 GPU0 將其上半，GPU1 將其下半各別寫回 host memory 即可。

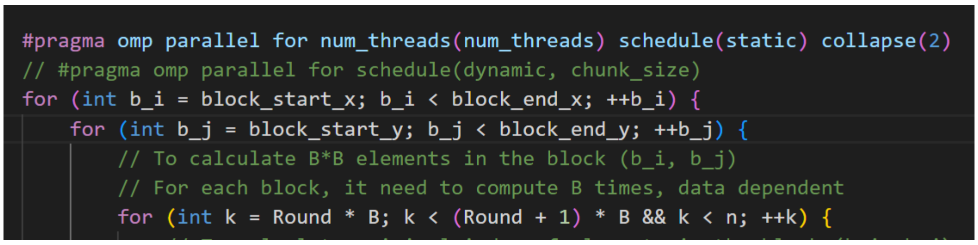
打勾的 tile 代表各 GPU 在 phase3 時 dependent 的 tile，也就是只要確保在每個 round 開始時的 pivot row 都是最新的，就能確保 phase3 是正確的，各 GPU 會用到的 pivot column 的 tiles 由於都位在自己處理的 rows 內，所以一定能確保是最新:



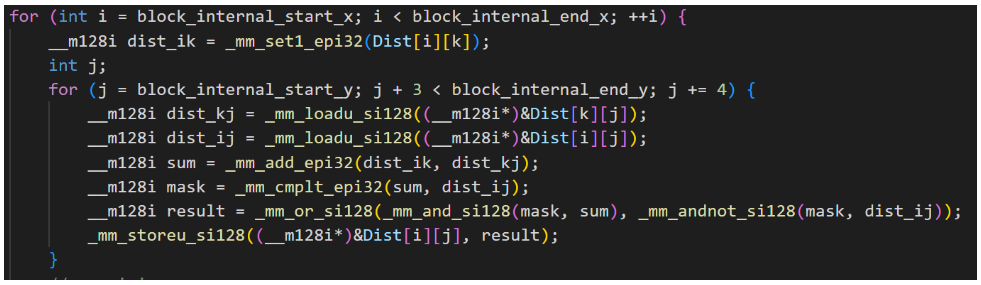
e. Briefly describe your implementations in diagrams, figures or sentences.

HW3-1(CPU)

使用 OpenMP 平行化外層的兩個 loops，也就是分配 tiles 給不同的 threads 去執行，每個 threads 負責處理 tile 內部的運算，並搭配 128bit SIMD register 可一次處理 4 個 INT，在 CPU 版本下，block factor 越小搭配 omp scheduler 有越好的 load balance，因為 threads 處理完一小塊 tile 後便可執行下一個。



```
#pragma omp parallel for num_threads(num_threads) schedule(static) collapse(2)
// #pragma omp parallel for schedule(dynamic, chunk_size)
for (int b_i = block_start_x; b_i < block_end_x; ++b_i) {
    for (int b_j = block_start_y; b_j < block_end_y; ++b_j) {
        // To calculate B*B elements in the block (b_i, b_j)
        // For each block, it need to compute B times, data dependent
        for (int k = Round * B; k < (Round + 1) * B && k < n; ++k) {
```



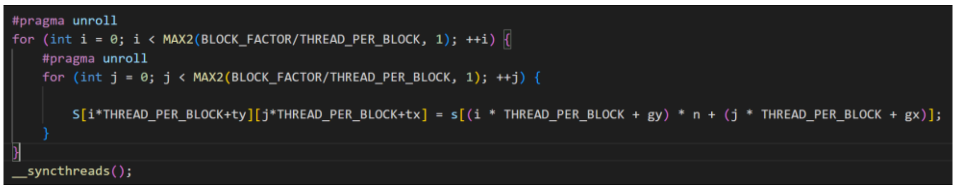
```
for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
    __m128i dist_ik = _mm_set1_epi32(Dist[i][k]);
    int j;
    for (j = block_internal_start_y; j + 3 < block_internal_end_y; j += 4) {
        __m128i dist_kj = _mm_loadu_si128((__m128i*)&Dist[k][j]);
        __m128i dist_ij = _mm_loadu_si128((__m128i*)&Dist[i][j]);
        __m128i sum = _mm_add_epi32(dist_ik, dist_kj);
        __m128i mask = _mm_cmplt_epi32(sum, dist_ij);
        __m128i result = _mm_or_si128(_mm_and_si128(mask, sum), _mm_andnot_si128(mask, dist_ij));
        _mm_storeu_si128((__m128i*)&Dist[i][j], result);
    }
```

HW3-2(Single-GPU)

Shared memory assignment:

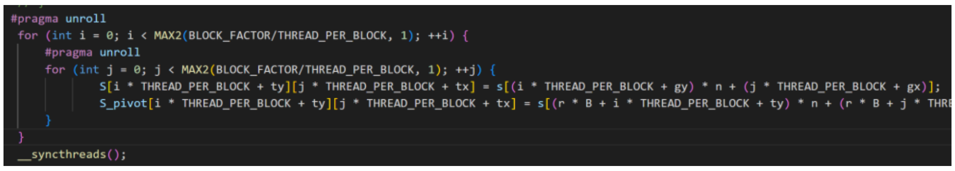
分成 phase1,2,3 共 3 個 kernels，每個 phase 的各 block 會先將其需要用到的 tiles 從 global memory load 進其 block 的 shared memory，而每個 block 的 thread 各負責處理 $(BLOCK_FACTOR/THREAD_PER_BLOCK)^2$個 datas，對應方式請參考 1-b。

Phase 1:



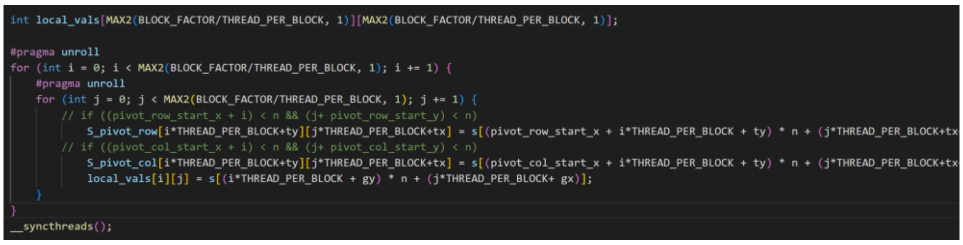
```
#pragma unroll
for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK, 1); ++i) {
    #pragma unroll
    for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK, 1); ++j) {

        S[i*THREAD_PER_BLOCK+ty][j*THREAD_PER_BLOCK+tx] = s[(i * THREAD_PER_BLOCK + gy) * n + (j * THREAD_PER_BLOCK + gx)];
    }
}
__syncthreads();
```

Phase2:



```
#pragma unroll
 for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK, 1); ++i) {
     #pragma unroll
     for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK, 1); ++j) {
            S[i * THREAD_PER_BLOCK + ty][j * THREAD_PER_BLOCK + tx] = s[(i * THREAD_PER_BLOCK + gy) * n + (j * THREAD_PER_BLOCK + gx)];
            S_pivot[i * THREAD_PER_BLOCK + ty][j * THREAD_PER_BLOCK + tx] = s[(r * B + i * THREAD_PER_BLOCK + ty) * n + (r * B + j * THRE
     }
 }
 __syncthreads();
```

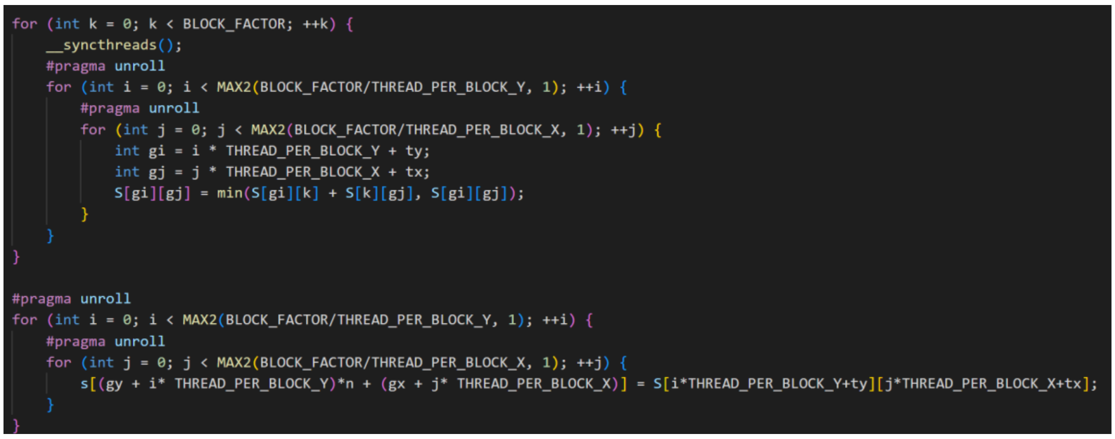
Phase3:



```
int local_vals[MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK, 1)][MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK, 1)];

#pragma unroll
for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK, 1); i += 1) {
    #pragma unroll
    for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK, 1); j += 1) {
        // if ((pivot_row_start_x + i) < n && (j+ pivot_row_start_y) < n)
            S_pivot_row[i*THREAD_PER_BLOCK+ty][j*THREAD_PER_BLOCK+tx] = s[(pivot_row_start_x + i*THREAD_PER_BLOCK + ty) * n + (j*THREAD_PER_BLOCK+tx
        // if ((pivot_col_start_x + i) < n && (j+ pivot_col_start_y) < n)
            S_pivot_col[i*THREAD_PER_BLOCK+ty][j*THREAD_PER_BLOCK+tx] = s[(pivot_col_start_x + i*THREAD_PER_BLOCK + ty) * n + (j*THREAD_PER_BLOCK+tx
            local_vals[i][j] = s[(i*THREAD_PER_BLOCK + gy) * n + (j*THREAD_PER_BLOCK+ gx)];
    }
}
__syncthreads();
```

各 phase 從 shared memory load 資料運算當下的 tile，最後寫回 global memory，(ty,tx)為 thread 的 index，(gy,gx)為各 data 在 n*n matrix 的 global index。

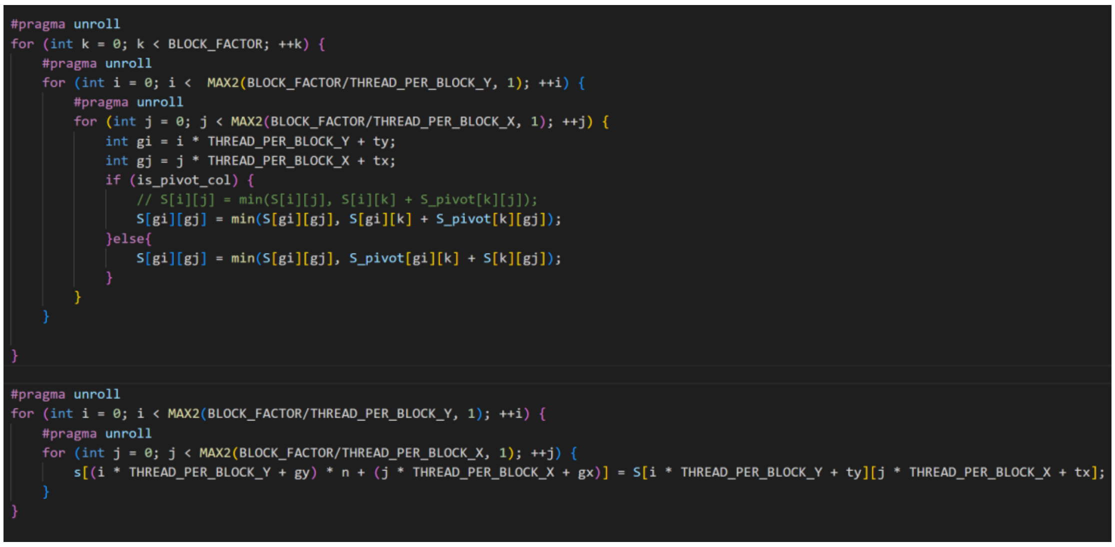
Phase1:



```
for (int k = 0; k < BLOCK_FACTOR; ++k) {
    __syncthreads();
    #pragma unroll
    for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_Y, 1); ++i) {
        #pragma unroll
        for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_X, 1); ++j) {
            int gi = i * THREAD_PER_BLOCK_Y + ty;
            int gj = j * THREAD_PER_BLOCK_X + tx;
            S[gi][gj] = min(S[gi][k] + S[k][gj], S[gi][gj]);
        }
    }
}

#pragma unroll
for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_Y, 1); ++i) {
    #pragma unroll
    for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_X, 1); ++j) {
        s[(gy + i* THREAD_PER_BLOCK_Y)*n + (gx + j* THREAD_PER_BLOCK_X)] = S[i*THREAD_PER_BLOCK_Y+ty][j*THREAD_PER_BLOCK_X+tx];
    }
}
```

由於 phase1 是 self-dependent，且 k 的 iterations 是 dependent 的，所以要確保整個 tile 裡的 data 在每個 k 時都是用到最新 update 的值，也因為我們 launch 的 block threads 數大於 32，一個 block 有多個 warps，所以未必免有 warps 先進到下個 iterations 用到尚未 update 的資料，所以在每個 k 都用 syncthreads()確保 block 裡的 threads 在每個 k 時是 sync 的。

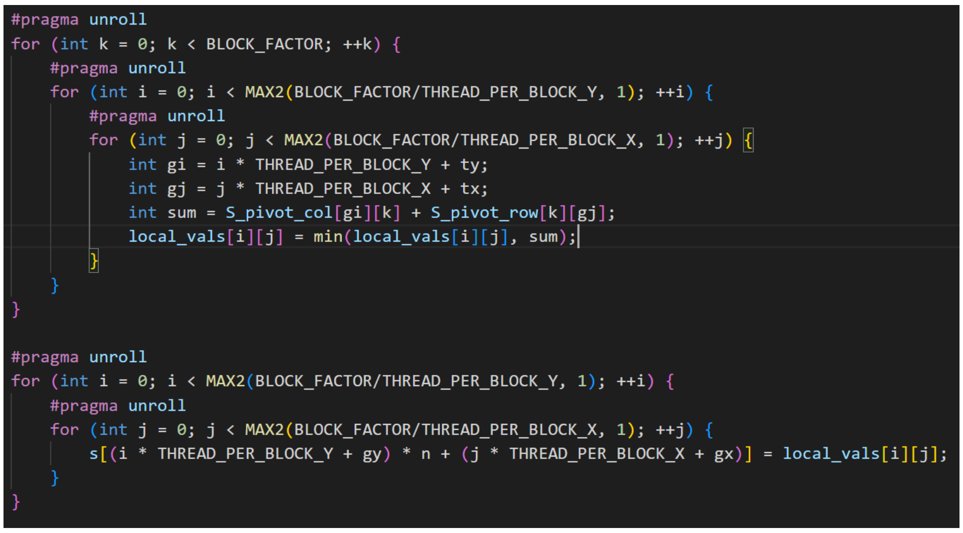
Phase2:



```
#pragma unroll
for (int k = 0; k < BLOCK_FACTOR; ++k) {
    #pragma unroll
    for (int i = 0; i <  MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_Y, 1); ++i) {
        #pragma unroll
        for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_X, 1); ++j) {
            int gi = i * THREAD_PER_BLOCK_Y + ty;
            int gj = j * THREAD_PER_BLOCK_X + tx;
            if (is_pivot_col) {
                // S[i][j] = min(S[i][j], S[i][k] + S_pivot[k][j]);
                S[gi][gj] = min(S[gi][gj], S[gi][k] + S_pivot[k][gj]);
            }else{
                S[gi][gj] = min(S[gi][gj], S_pivot[gi][k] + S[k][gj]);
            }
        }
    }

}

#pragma unroll
for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_Y, 1); ++i) {
    #pragma unroll
    for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_X, 1); ++j) {
        s[(i * THREAD_PER_BLOCK_Y + gy) * n + (j * THREAD_PER_BLOCK_X + gx)] = S[i * THREAD_PER_BLOCK_Y + ty][j * THREAD_PER_BLOCK_X + tx];
    }
}
```

Phase2 在計算時，會根據是 pivot row 或是 col 計算方式會有不同，當 data 在 pivot row 時，其會用到對應到的 pivot tile 的 row(S_pivot[gi][k])以及 self tile 的 column；當 data 在 pivot column 時，則是反過來。

Phase3:

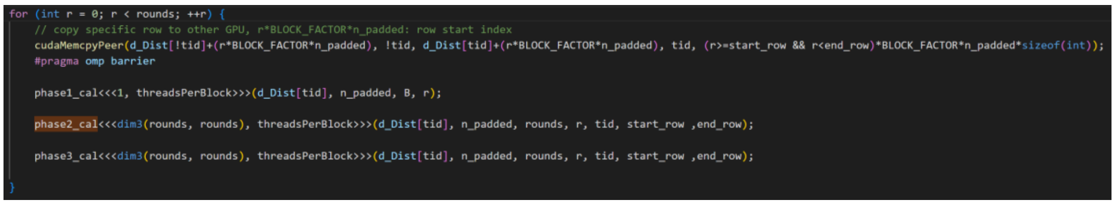


```
#pragma unroll
for (int k = 0; k < BLOCK_FACTOR; ++k) {
    #pragma unroll
    for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_Y, 1); ++i) {
        #pragma unroll
        for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_X, 1); ++j) {
            int gi = i * THREAD_PER_BLOCK_Y + ty;
            int gj = j * THREAD_PER_BLOCK_X + tx;
            int sum = S_pivot_col[gi][k] + S_pivot_row[k][gj];
            local_vals[i][j] = min(local_vals[i][j], sum);
        }
    }
}

#pragma unroll
for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_Y, 1); ++i) {
    #pragma unroll
    for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_X, 1); ++j) {
        s[(i * THREAD_PER_BLOCK_Y + gy) * n + (j * THREAD_PER_BLOCK_X + gx)] = local_vals[i][j];
    }
}
```

Phase3 在計算時，會使用到對應到的 pivot row 跟 column 的 tile，local_vals 為暫存當下的 distance，最後算完後再 load 回 global memory，這樣可以避免頻繁寫入到 global memory。

HW3-3(Multi-GPU)

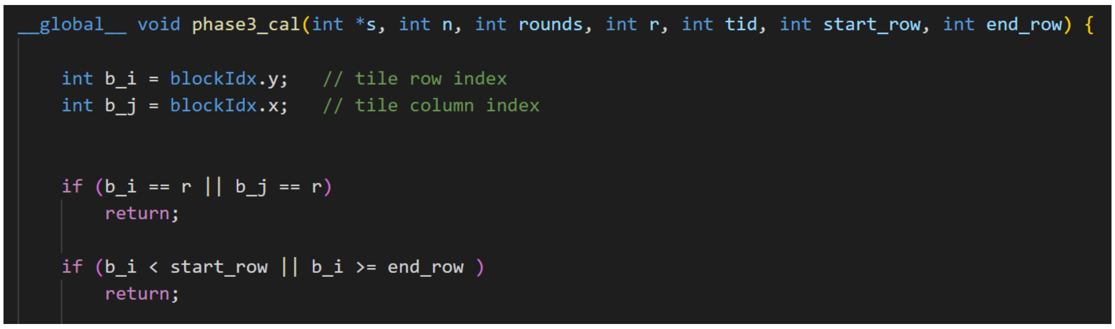
Data Exchange between GPUs:



```
for (int r = 0; r < rounds; ++r) {
    // copy specific row to other GPU, r*BLOCK_FACTOR*n_padded: row start index
    cudaMemcpyPeer(d_Dist[!tid]+(r*BLOCK_FACTOR*n_padded), !tid, d_Dist[tid]+(r*BLOCK_FACTOR*n_padded), tid, (r>=start_row && r<end_row)*BLOCK_FACTOR*n_padded*sizeof(int));
    #pragma omp barrier

    phase1_cal<<<1, threadsPerBlock>>>(d_Dist[tid], n_padded, B, r);

    phase2_cal<<<dim3(rounds, rounds), threadsPerBlock>>>(d_Dist[tid], n_padded, rounds, r, tid, start_row ,end_row);

    phase3_cal<<<dim3(rounds, rounds), threadsPerBlock>>>(d_Dist[tid], n_padded, rounds, r, tid, start_row ,end_row);

}
```

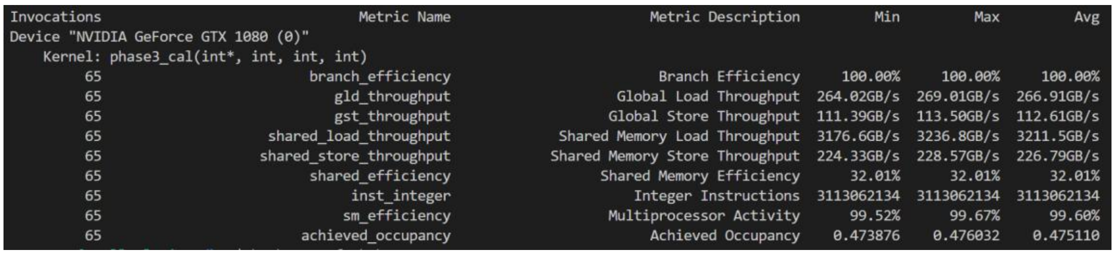
在每個 round 開始時由負責處理當 round 的 pivot row 的 GPU 將 row 用 cudaMemcpyPeer 傳給另一張 GPU。

Phase3 各 GPU 只會處理自己負責的 rows:



```
__global__ void phase3_cal(int *s, int n, int rounds, int r, int tid, int start_row, int end_row) {

    int b_i = blockIdx.y;    // tile row index
    int b_j = blockIdx.x;    // tile column index


    if (b_i == r || b_j == r)
        return;

    if (b_i < start_row || b_i >= end_row )
        return;
```

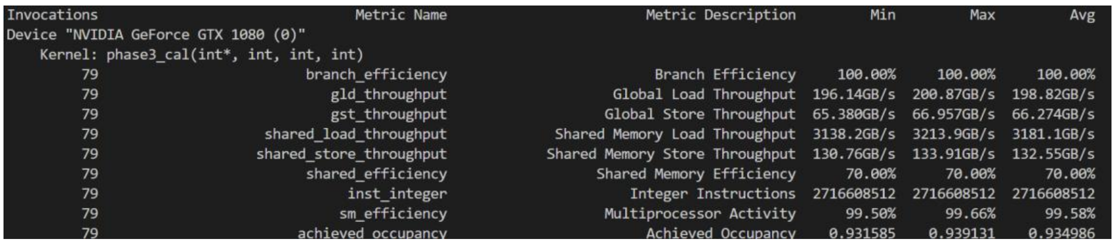
Profiling Results (hw3-2)



```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
         65                         branch_efficiency                         Branch Efficiency     100.00%     100.00%     100.00%
         65                            gld_throughput                    Global Load Throughput  264.02GB/s  269.01GB/s  266.91GB/s
         65                            gst_throughput                   Global Store Throughput  111.39GB/s  113.50GB/s  112.61GB/s
         65                    shared_load_throughput             Shared Memory Load Throughput  3176.6GB/s  3236.8GB/s  3211.5GB/s
         65                   shared_store_throughput            Shared Memory Store Throughput  224.33GB/s  228.57GB/s  226.79GB/s
         65                         shared_efficiency                  Shared Memory Efficiency      32.01%      32.01%      32.01%
         65                              inst_integer                      Integer Instructions  3113062134  3113062134  3113062134
         65                             sm_efficiency                   Multiprocessor Activity      99.52%      99.67%      99.60%
         65                        achieved_occupancy                        Achieved Occupancy    0.473876    0.476032    0.475110
```

Fig. 2-1 Block_factor=78, threads=39*13, testcase:c20.1

這裡我用 nvprof 去 profile phase3_cal 這個最大的 kernel，testcase 選擇 c20.1 主要是 profiling 很花時間，太大的 testcase 會跑太久被 slurm terminate 掉。可看到當一個 block 的 thread 設 39*13 時，occupancy 會下降，這是因為一個 block 需要 16 個 warps(ceil(39*13/32)=16)，但又因為我們一個 block 用滿了 GTX 1080 的 per block shared memory limit(48KB)，然後 max shared memory per SM 是 96KB，代表在 block_factor =78，threads 數為 39*13 的情況下，一個 SM 只能有兩個 blocks，2*16/64=0.5，導致較低的 warps occupancy，當我將 threads 數為 32*32 時，兩個 blocks 共 64 warps，所以可以佔滿 occupancy:



```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
         79                         branch_efficiency                         Branch Efficiency     100.00%     100.00%     100.00%
         79                            gld_throughput                    Global Load Throughput  196.14GB/s  200.87GB/s  198.82GB/s
         79                            gst_throughput                   Global Store Throughput  65.380GB/s  66.957GB/s  66.274GB/s
         79                    shared_load_throughput             Shared Memory Load Throughput  3138.2GB/s  3213.9GB/s  3181.1GB/s
         79                   shared_store_throughput            Shared Memory Store Throughput  130.76GB/s  133.91GB/s  132.55GB/s
         79                         shared_efficiency                  Shared Memory Efficiency      70.00%      70.00%      70.00%
         79                              inst_integer                      Integer Instructions  2716608512  2716608512  2716608512
         79                             sm_efficiency                   Multiprocessor Activity      99.50%      99.66%      99.58%
         79                        achieved_occupancy                        Achieved Occupancy    0.931585    0.939131    0.934986
```

Fig. 2-2 Block_factor=64, threads=32*32, testcase:c20.1

但較低的 occupancy 不一定代表較差的 optimization，Nvidia 官方 CUDA document 中有提到「Higher occupancy does not always equate to higher performance-there is a point above which additional occupancy does not improve performance.」，occupancy 只要足夠可以掩蓋掉 memory stall 的 latency 即可，而在使用 shared memory latency 很低的情況下，過高的 occupancy 反而沒有必要，因此我們可以降低 threads 數犧牲一點 occupancy，讓每個 thread 處理更多 data 提高 ILP(Instruction-Level Parallelism)，這樣可以更好地利用每個 thread。另外，Fig. 2-1 的 shared store/global load/store 皆明顯上升，主要也是因為一個 thread 一次處理處理更多資料，但可以看到 shared load 跟 shared efficiency 有下降，主要是因為 bank conflicts。

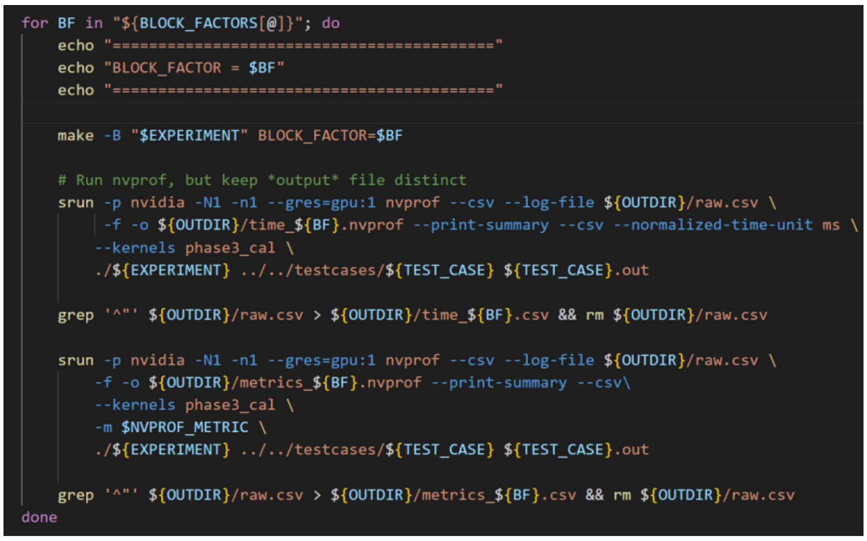
Experiment & Analysis (Nvidia GPU)

a. System Spec:

使用的機器為這門課提供的 Apollo Server。

b. Blocking Factor (hw3-2)

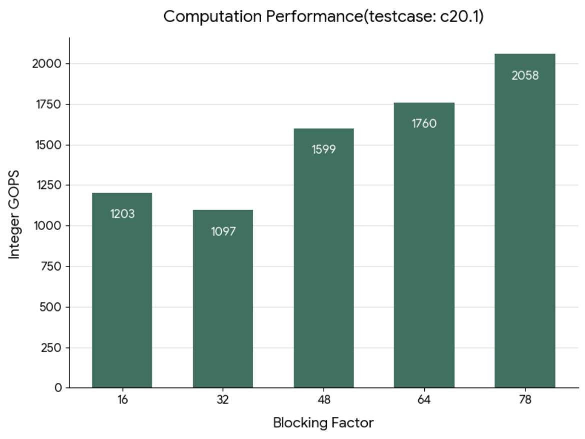
由於 GTX 1080 的 shared memory per block 的上限，針對 shared memory 的實驗上限只能設 78，總共試了以下幾種 BF:{16, 32, 48, 64, 78}，針對 global memory 的實驗則是 BF={16, 32, 64, 128, 256}，testcase 為 c20.1:

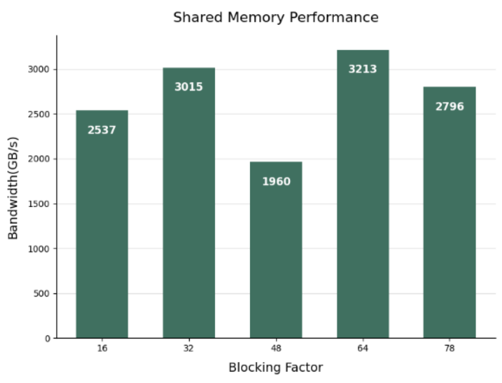


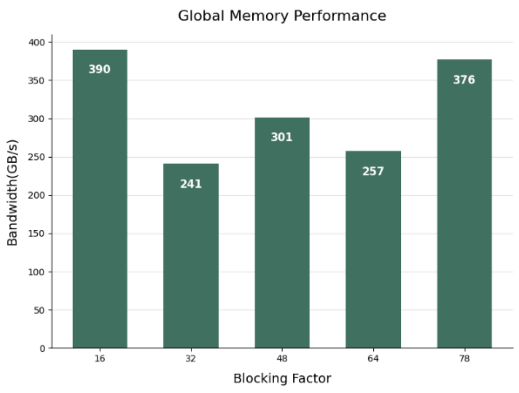
```
for BF in "${BLOCK_FACTORS[@]}"; do
    echo "========================================="
    echo "BLOCK_FACTOR = $BF"
    echo "========================================="

    make -B "$EXPERIMENT" BLOCK_FACTOR=$BF

    # Run nvprof, but keep *output* file distinct
    srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --csv --log-file ${OUTDIR}/raw.csv \
         -f -o ${OUTDIR}/time_${BF}.nvprof --print-summary --csv --normalized-time-unit ms \
        --kernels phase3_cal \
        ./${EXPERIMENT} ../../testcases/${TEST_CASE} ${TEST_CASE}.out

    grep '^"' ${OUTDIR}/raw.csv > ${OUTDIR}/time_${BF}.csv && rm ${OUTDIR}/raw.csv

    srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --csv --log-file ${OUTDIR}/raw.csv \
        -f -o ${OUTDIR}/metrics_${BF}.nvprof --print-summary --csv\
        --kernels phase3_cal \
        -m $NVPROF_METRIC \
        ./${EXPERIMENT} ../../testcases/${TEST_CASE} ${TEST_CASE}.out

    grep '^"' ${OUTDIR}/raw.csv > ${OUTDIR}/metrics_${BF}.csv && rm ${OUTDIR}/raw.csv
done
```

Integer GOPS 的算法我是用 nvprof 去紀錄各 kernel 一個 invocation 平均的時間 t_{avg}(ms)，再用 inst_integer metric 得出的一個 invocation 平均的 int instructions I_{avg}，用以下算式得出: $Int\ GOPS = \frac{I_{avg}}{(t_{avg}(ms)*10^{-3})} * 10^{-9}$

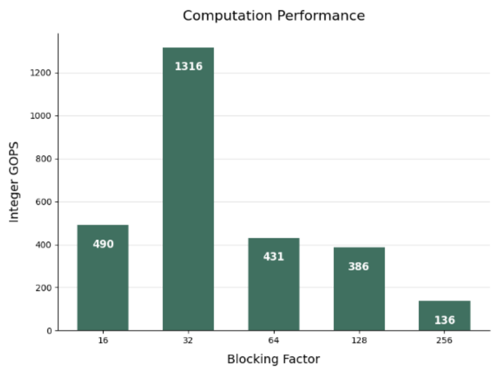
Using shared mem:

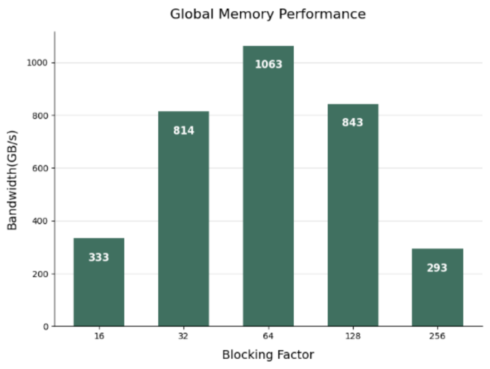






W/O shared mem(coalesced global memory access):





可以看到有用 shared memory 的情況下，int GOPS 基本上是隨著 BF 上升的，但因為 shared memory per block 的上限，最高只能設 78，再來針對 BF 78，因為 threads 數沒辦法設 39*39(維持一個 thread 處理 4 個 datas)，所以改設 39*13，occupancy 下降，所以 shared memory 得 bandwidth 有下降，但 ILP 上升，所以仍可以提升整體的 computation performance；沒有使用 shared memory 的情況下可以將 BF 設更高，但過大的 BF 會導致整體效率下降，主要是因為高 ILP 在 global memory 的 bottleneck 下，使 stall latency 更高，即便高 occupancy(threads 數皆為 32*32)也無法掩蓋掉這些 latency。

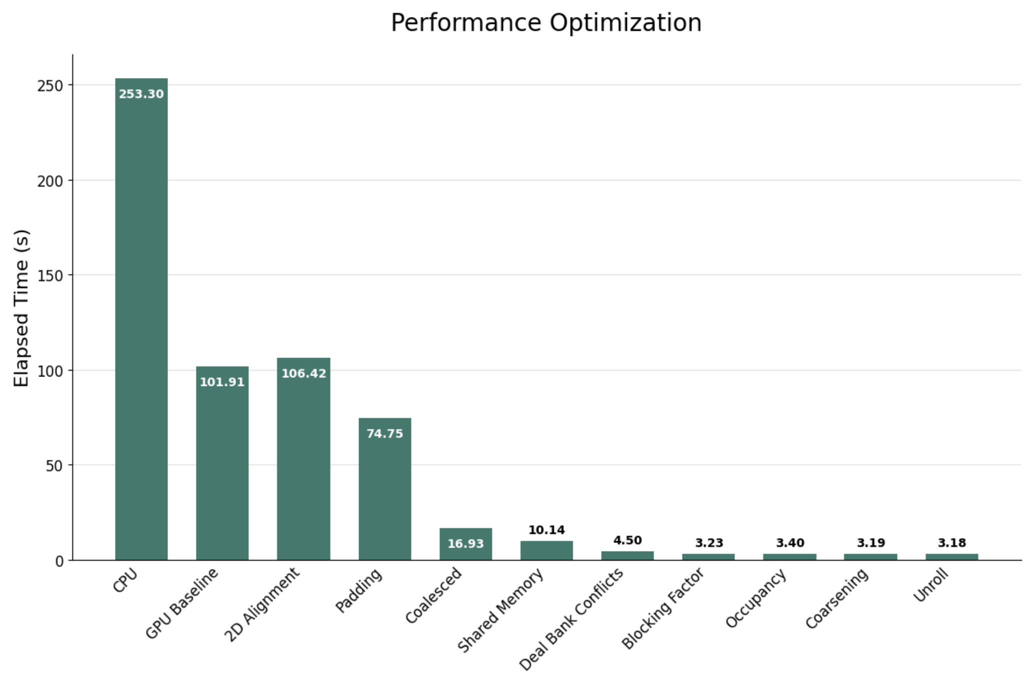
c. Optimization(hw3-2):

由於太大的 testcase 在跑 baseline 時會超過 slurm 每個 task 5 分鐘限制，所以我選 p12k1 作為實驗的 testcase。

我依序試了以下幾種 optimizations，optimizations 是 cumulative 的，所以每一步都是在前一項優化的基礎上繼續優化:

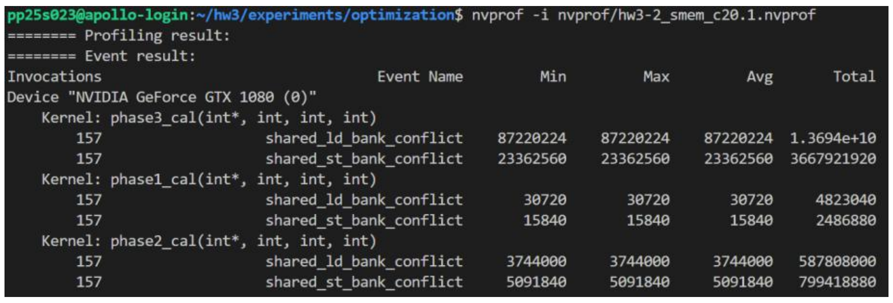
1. 2D alignment: grid 跟 block 由原本 baseline 的 1D 改成 2D;
2. Padding: 原本 n*n 的 data 改成 n_padded*n_padded 可以整除 BLOCK_FACTOR，移除 index 的 boundary check；
3. Coalesced memory access: 將讀取 global memory 的 index 由 col-major 改為 row-major，確保一個 warp 是讀取連續區塊的 memory;

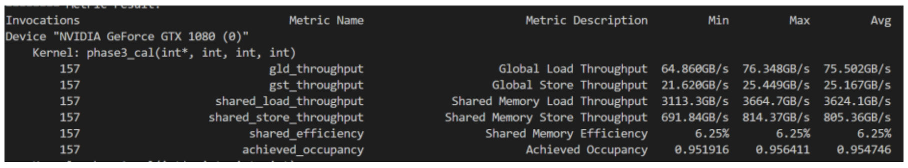
4. Shared memory(w/ bank conflicts): 因為演算法會一直 reuse dependent tile 的 data，所以將每個 phase 會用到的 tile load 到 shared memory 上再進行計算，但用 col major 去 load/store shared memory，所以有嚴重 bank conflict;
5. Shared memory(deal with bank conflicts): 改用 row-major 去 load/store shared memory，降低 bank conflict;
6. Blocking factor tuning: 將 BF 由 32 提高至 64；
7. Occupancy optimization: threads 數由 16*16 提高至 32*32，由於提高一個 block 的 shared memory usage 後，sm 上最多能同時有的 blocks 數下降，此時若 threads 數仍為 16*16 的話，occupancy 低，提升 threads 數可提高 sm 上的 warps 數以提高 occupancy；
8. Thread coarsening: 將一個 block 的 threads 數由 32*32 降為 32*8，此方法降低了 sm 上 warps 的 occupancy，但讓一個 thread 一次計算更多 data，可提高 ILP;
9. Loop unroll: 將 for (int k = 0; k < BLOCK_FACTOR; ++k) 此 for loop 用 openacc directive 去 unroll，減少 iterations 數；



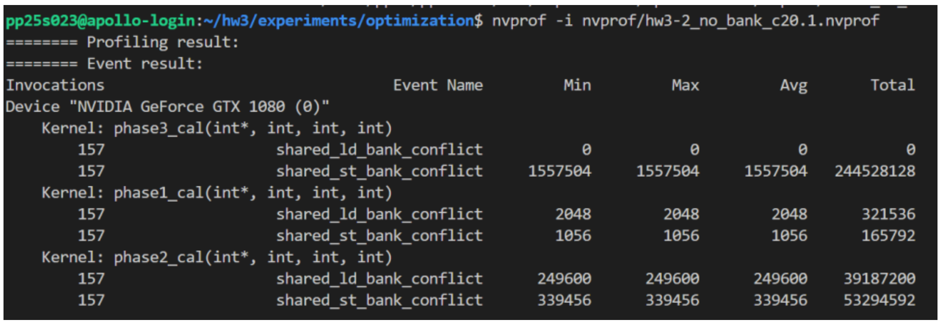
接著我用 nvprof events 去查看在使用 shared memory 下是否有 bank conflict 發生:

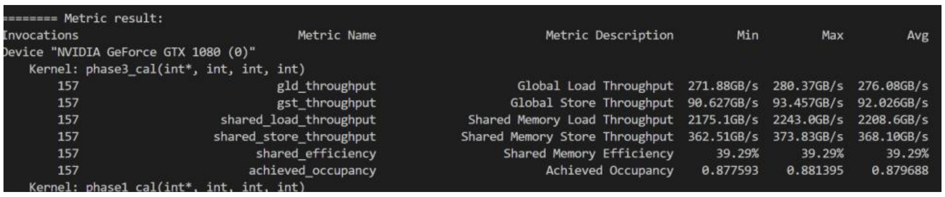
w/ bank conflicts(BF=32, threads=16*16):


```
pp25s023@apollo-login:~/hw3/experiments/optimization$ nvprof -i nvprof/hw3-2_smem_c20.1.nvprof
======== Profiling result:
======== Event result:
Invocations                                Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
        157                  shared_ld_bank_conflict    87220224    87220224    87220224  1.3694e+10
        157                  shared_st_bank_conflict    23362560    23362560    23362560  3667921920
    Kernel: phase1_cal(int*, int, int, int)
        157                  shared_ld_bank_conflict       30720       30720       30720     4823040
        157                  shared_st_bank_conflict       15840       15840       15840     2486880
    Kernel: phase2_cal(int*, int, int, int)
        157                  shared_ld_bank_conflict     3744000     3744000     3744000   587808000
        157                  shared_st_bank_conflict     5091840     5091840     5091840   799418880
```


```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
        157                            gld_throughput                  Global Load Throughput  64.860GB/s  76.348GB/s  75.502GB/s
        157                            gst_throughput                 Global Store Throughput  21.620GB/s  25.449GB/s  25.167GB/s
        157                      shared_load_throughput           Shared Memory Load Throughput  3113.3GB/s  3664.7GB/s  3624.1GB/s
        157                     shared_store_throughput          Shared Memory Store Throughput  691.84GB/s  814.37GB/s  805.36GB/s
        157                           shared_efficiency                Shared Memory Efficiency       6.25%       6.25%       6.25%
        157                          achieved_occupancy                      Achieved Occupancy    0.951916    0.956411    0.954746
```

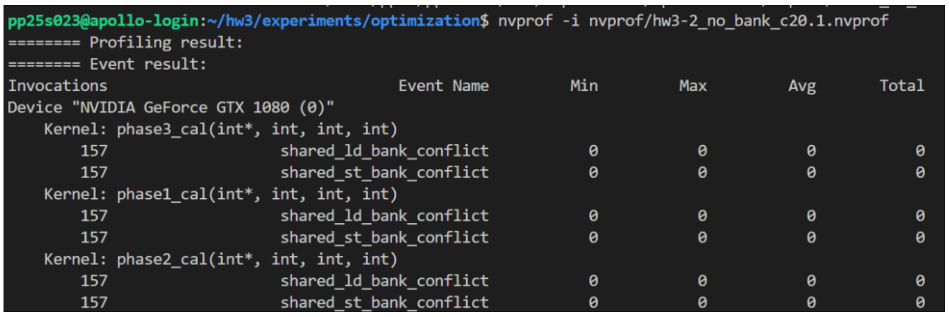
deal with bank conflicts(BF=32, threads=16*16):

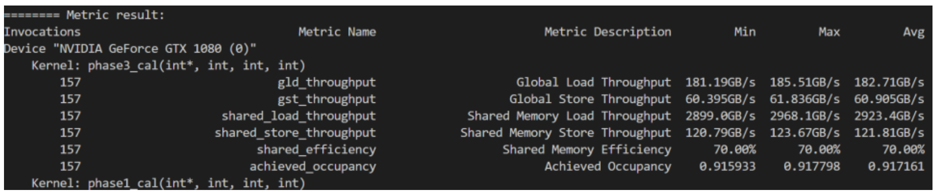

```
pp25s023@apollo-login:~/hw3/experiments/optimization$ nvprof -i nvprof/hw3-2_no_bank_c20.1.nvprof
======== Profiling result:
======== Event result:
Invocations                                Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
        157                   shared_ld_bank_conflict           0           0           0           0
        157                   shared_st_bank_conflict     1557504     1557504     1557504   244528128
    Kernel: phase1_cal(int*, int, int, int)
        157                   shared_ld_bank_conflict        2048        2048        2048      321536
        157                   shared_st_bank_conflict        1056        1056        1056      165792
    Kernel: phase2_cal(int*, int, int, int)
        157                   shared_ld_bank_conflict      249600      249600      249600    39187200
        157                   shared_st_bank_conflict      339456      339456      339456    53294592
```


```
======== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
        157                            gld_throughput                  Global Load Throughput  271.88GB/s  280.37GB/s  276.08GB/s
        157                            gst_throughput                 Global Store Throughput  90.627GB/s  93.457GB/s  92.026GB/s
        157                      shared_load_throughput           Shared Memory Load Throughput  2175.1GB/s  2243.0GB/s  2208.6GB/s
        157                     shared_store_throughput          Shared Memory Store Throughput  362.51GB/s  373.83GB/s  368.10GB/s
        157                           shared_efficiency                Shared Memory Efficiency      39.29%      39.29%      39.29%
        157                          achieved_occupancy                      Achieved Occupancy    0.877593    0.881395    0.879688
    Kernel: phase1_cal(int*, int, int, int)
```

可以看到用 row-major 去 access shared memory 可以大幅降低 bank conflicts(由於 threads 是 16*16 所以仍有 2-way bank conflict)，shared memory efficiency 顯著提升。

No bank conflicts(BF=32, threads=32*32):


```
pp25s023@apollo-login:~/hw3/experiments/optimization$ nvprof -i nvprof/hw3-2_no_bank_c20.1.nvprof
======== Profiling result:
======== Event result:
Invocations                                Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
        157                   shared_ld_bank_conflict           0           0           0           0
        157                   shared_st_bank_conflict           0           0           0           0
    Kernel: phase1_cal(int*, int, int, int)
        157                   shared_ld_bank_conflict           0           0           0           0
        157                   shared_st_bank_conflict           0           0           0           0
    Kernel: phase2_cal(int*, int, int, int)
        157                   shared_ld_bank_conflict           0           0           0           0
        157                   shared_st_bank_conflict           0           0           0           0
```

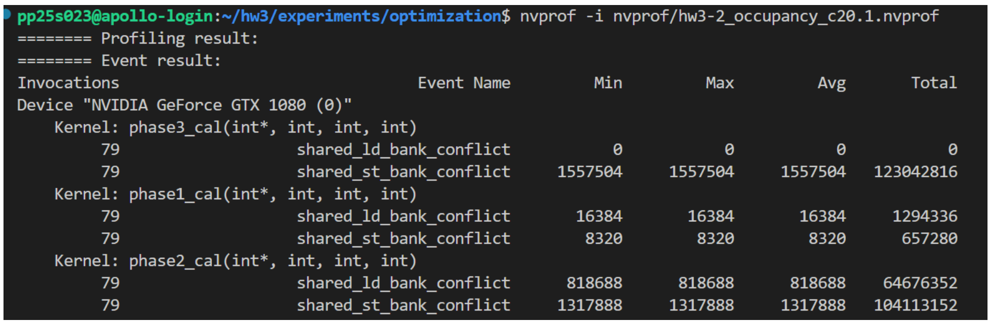

```
======== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
        157                            gld_throughput                  Global Load Throughput  181.19GB/s  185.51GB/s  182.71GB/s
        157                            gst_throughput                 Global Store Throughput  60.395GB/s  61.836GB/s  60.905GB/s
        157                      shared_load_throughput           Shared Memory Load Throughput  2899.0GB/s  2968.1GB/s  2923.4GB/s
        157                     shared_store_throughput          Shared Memory Store Throughput  120.79GB/s  123.67GB/s  121.81GB/s
        157                           shared_efficiency                Shared Memory Efficiency      70.00%      70.00%      70.00%
        157                          achieved_occupancy                      Achieved Occupancy    0.915933    0.917798    0.917161
    Kernel: phase1_cal(int*, int, int, int)
```

使 blockDim.x=32 可以完全移除 bank conflicts。

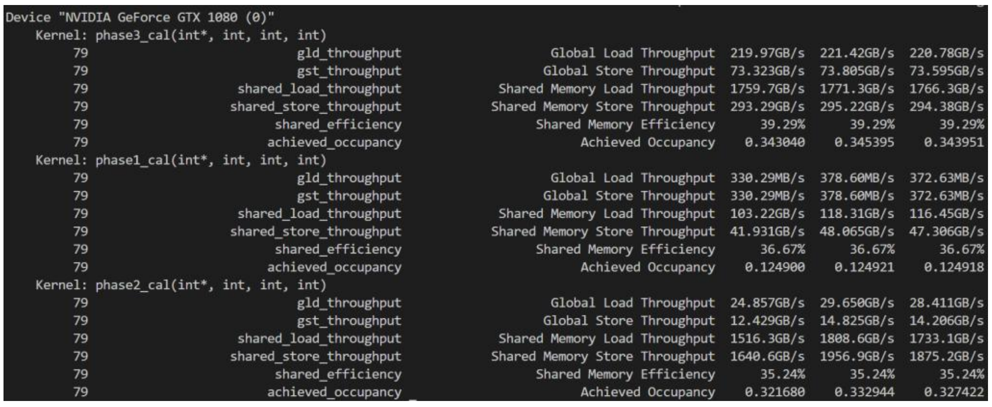
另外在做 Thread coarsening 時，例如將 block 由 32*32 降為 16*16 threads，以目前讀取 shared memory 的方式，一個 warp 會跨 2 個 rows，此時 warp 中可能會有些 threads 讀到同一個 bank，造成 2-way bank conflict，所以我將 16*16 調整為 32*8，threads 數一

樣，一個 thread 都是處理 16 個 datas，ILP 一樣，但一個 warp 就是一個 row，不會有 bank conflict:

BF=64, Threads=16*16:

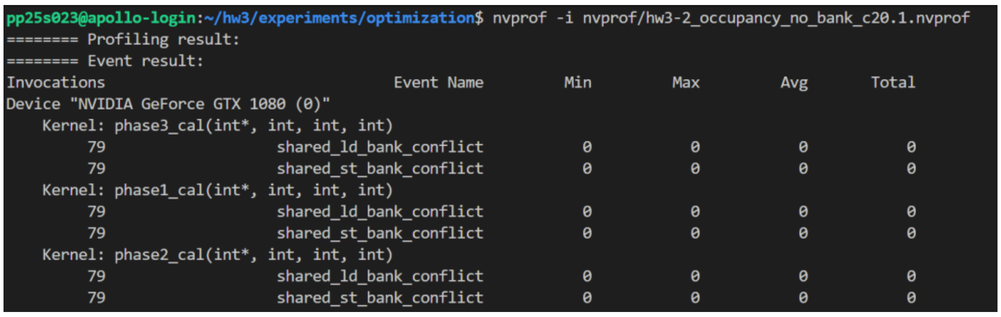


```
pp25s023@apollo-login:~/hw3/experiments/optimization$ nvprof -i nvprof/hw3-2_occupancy_c20.1.nvprof
======== Profiling result:
======== Event result:
Invocations                                Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
         79                      shared_ld_bank_conflict           0           0           0           0
         79                      shared_st_bank_conflict     1557504     1557504     1557504   123042816
    Kernel: phase1_cal(int*, int, int, int)
         79                      shared_ld_bank_conflict       16384       16384       16384     1294336
         79                      shared_st_bank_conflict        8320        8320        8320      657280
    Kernel: phase2_cal(int*, int, int, int)
         79                      shared_ld_bank_conflict      818688      818688      818688    64676352
         79                      shared_st_bank_conflict     1317888     1317888     1317888   104113152
```

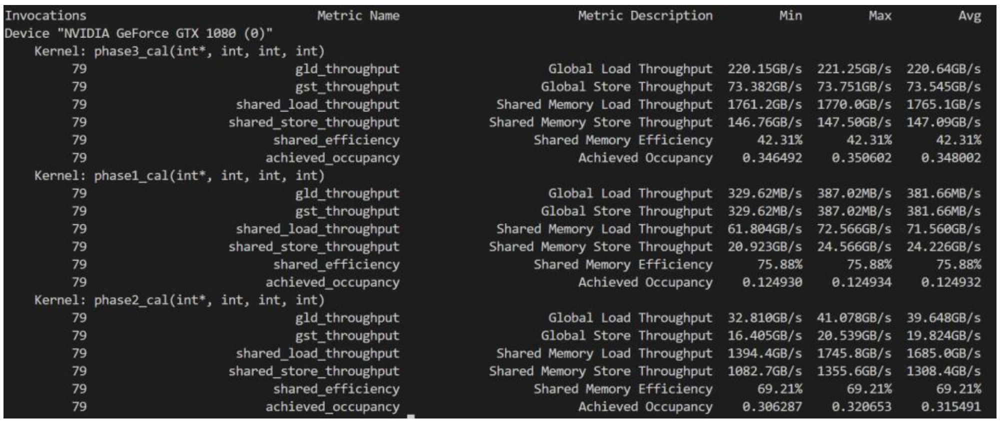


```
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
         79                            gld_throughput                    Global Load Throughput  219.97GB/s  221.42GB/s  220.78GB/s
         79                            gst_throughput                   Global Store Throughput  73.323GB/s  73.805GB/s  73.595GB/s
         79                     shared_load_throughput             Shared Memory Load Throughput  1759.7GB/s  1771.3GB/s  1766.3GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  293.29GB/s  295.22GB/s  294.38GB/s
         79                         shared_efficiency                  Shared Memory Efficiency      39.29%      39.29%      39.29%
         79                        achieved_occupancy                        Achieved Occupancy    0.343040    0.345395    0.343951
    Kernel: phase1_cal(int*, int, int, int)
         79                            gld_throughput                    Global Load Throughput  330.29MB/s  378.60MB/s  372.63MB/s
         79                            gst_throughput                   Global Store Throughput  330.29MB/s  378.60MB/s  372.63MB/s
         79                     shared_load_throughput             Shared Memory Load Throughput  103.22GB/s  118.31GB/s  116.45GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  41.931GB/s  48.065GB/s  47.306GB/s
         79                         shared_efficiency                  Shared Memory Efficiency      36.67%      36.67%      36.67%
         79                        achieved_occupancy                        Achieved Occupancy    0.124900    0.124921    0.124918
    Kernel: phase2_cal(int*, int, int, int)
         79                            gld_throughput                    Global Load Throughput  24.857GB/s  29.650GB/s  28.411GB/s
         79                            gst_throughput                   Global Store Throughput  12.429GB/s  14.825GB/s  14.206GB/s
         79                     shared_load_throughput             Shared Memory Load Throughput  1516.3GB/s  1808.6GB/s  1733.1GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  1640.6GB/s  1956.9GB/s  1875.2GB/s
         79                         shared_efficiency                  Shared Memory Efficiency      35.24%      35.24%      35.24%
         79                        achieved_occupancy                        Achieved Occupancy    0.321680    0.332944    0.327422
```

BF=64, Threads=32*8:



```
pp25s023@apollo-login:~/hw3/experiments/optimization$ nvprof -i nvprof/hw3-2_occupancy_no_bank_c20.1.nvprof
======== Profiling result:
======== Event result:
Invocations                                Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
         79                      shared_ld_bank_conflict           0           0           0           0
         79                      shared_st_bank_conflict           0           0           0           0
    Kernel: phase1_cal(int*, int, int, int)
         79                      shared_ld_bank_conflict           0           0           0           0
         79                      shared_st_bank_conflict           0           0           0           0
    Kernel: phase2_cal(int*, int, int, int)
         79                      shared_ld_bank_conflict           0           0           0           0
         79                      shared_st_bank_conflict           0           0           0           0
```



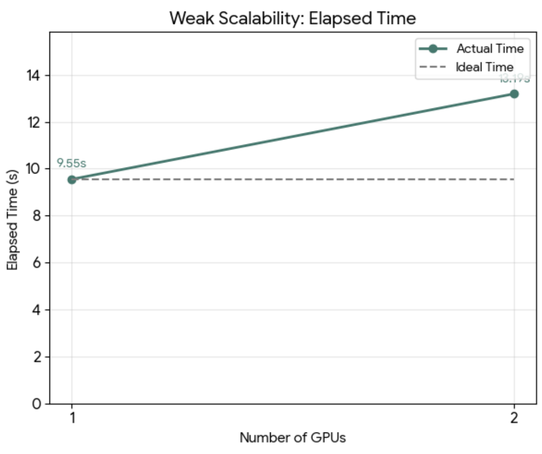
```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3_cal(int*, int, int, int)
         79                            gld_throughput                    Global Load Throughput  220.15GB/s  221.25GB/s  220.64GB/s
         79                            gst_throughput                   Global Store Throughput  73.382GB/s  73.751GB/s  73.545GB/s
         79                     shared_load_throughput             Shared Memory Load Throughput  1761.2GB/s  1770.0GB/s  1765.1GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  146.76GB/s  147.50GB/s  147.09GB/s
         79                         shared_efficiency                  Shared Memory Efficiency      42.31%      42.31%      42.31%
         79                        achieved_occupancy                        Achieved Occupancy    0.346492    0.350602    0.348002
    Kernel: phase1_cal(int*, int, int, int)
         79                            gld_throughput                    Global Load Throughput  329.62MB/s  387.02MB/s  381.66MB/s
         79                            gst_throughput                   Global Store Throughput  329.62MB/s  387.02MB/s  381.66MB/s
         79                     shared_load_throughput             Shared Memory Load Throughput  61.804GB/s  72.566GB/s  71.560GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  20.923GB/s  24.566GB/s  24.226GB/s
         79                         shared_efficiency                  Shared Memory Efficiency      75.88%      75.88%      75.88%
         79                        achieved_occupancy                        Achieved Occupancy    0.124930    0.124934    0.124932
    Kernel: phase2_cal(int*, int, int, int)
         79                            gld_throughput                    Global Load Throughput  32.810GB/s  41.078GB/s  39.648GB/s
         79                            gst_throughput                   Global Store Throughput  16.405GB/s  20.539GB/s  19.824GB/s
         79                     shared_load_throughput             Shared Memory Load Throughput  1394.4GB/s  1745.8GB/s  1685.0GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  1082.7GB/s  1355.6GB/s  1308.4GB/s
         79                         shared_efficiency                  Shared Memory Efficiency      69.21%      69.21%      69.21%
         79                        achieved_occupancy                        Achieved Occupancy    0.306287    0.320653    0.315491
```

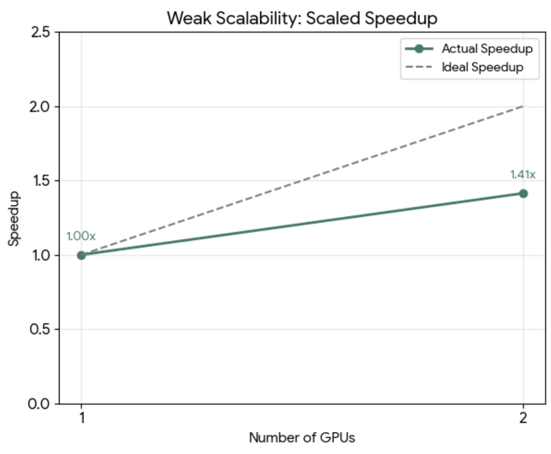
可以看到沒了 bank conflicts，但由於對於效能影響最大的是在 phase3 kernel 的 shared load 原本就因 access 的方式沒有造成 bank conflicts，所以即便其他 phase 沒了 bank conflicts，整體效率提升不大，shared memory efficiency 也只有些微上升。

d. Weak scalability (hw3-3)

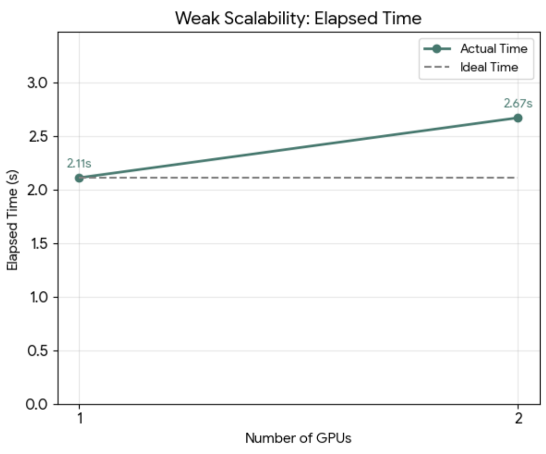
針對 weak scalability 我採用幾對 testcases，由於 Floyd Warshall 演算法 time complexity 為 O(N^3)，所以 2 張 GPU 的 testcase 的 vertices 應該要為 1.26 倍，例如 p25k1 的 complexity 約為 p20k1 的 1.95 倍(p20k1 有 20000 個 vertices，p25k1 則有 25000 個 vertices)，可以比較一張及兩張 GPU 下的 weak scalability 及 speedup。

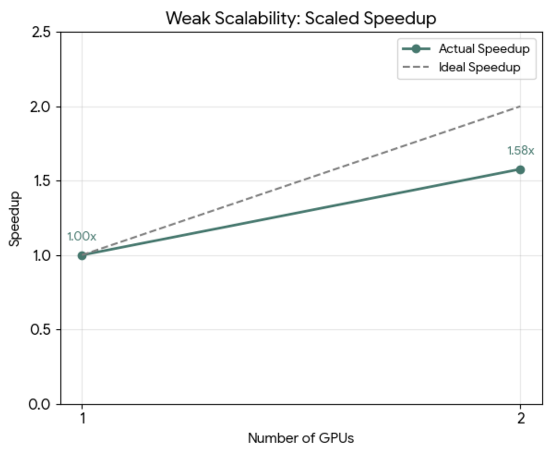
p20k1 v.s. p25k1:



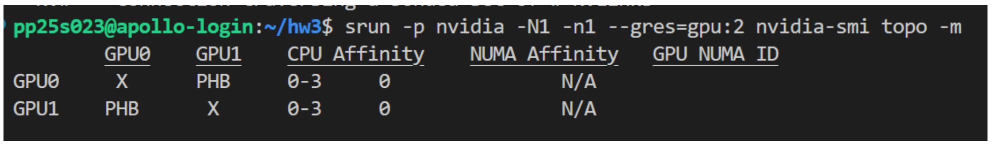


p11k1 v.s. p14k1:





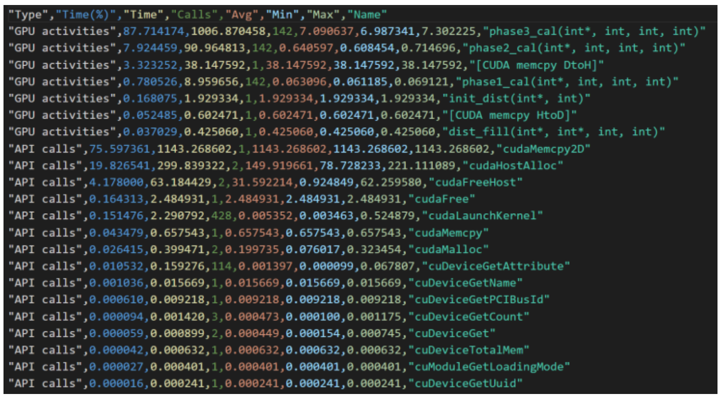
可以看到用 multi-gpu 時因為 GPU 間 communication 的成本很難達到 ideal speedup，且隨著 problem size 越大，speedup 反而變差，主要是因為每次 cudaMemcpy overhead 隨著資料及 rounds 數變大而提高，再來，透過 nvidia-smi 可以看到，兩張 GPUs 是透過 PCIe host bridge 而非像是 PXB, PIX 甚至是更快的 NVlink 直連，而是要經過 CPU，所以即便使用了 `cudaMemcpyPeer()`，communication overhead 仍很大:



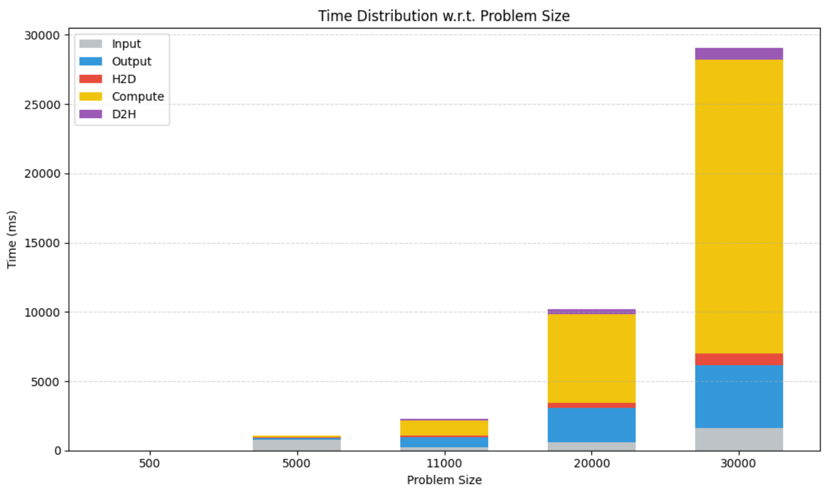
```
pp25s023@apollo-login:~/hw3$ srun -p nvidia -N1 -n1 --gres=gpu:2 nvidia-smi topo -m
        GPU0    GPU1    CPU Affinity    NUMA Affinity   GPU NUMA ID
GPU0     X      PHB     0-3     0               N/A
GPU1    PHB      X      0-3     0               N/A
```

e. Time Distribution (hw3-2)

這邊一樣用 nvprof 去 profile 每個 GPU activities 的時間組成，input 跟 output 則另外用 clock_gettime()去紀錄:



```
"Type","Time(%)","Time","Calls","Avg","Min","Max","Name"
"GPU activities",87.714174,1006.870458,142,7.090637,6.987341,7.302225,"phase3_cal(int*, int, int, int)"
"GPU activities",7.924459,90.964813,142,0.640597,0.608454,0.714696,"phase2_cal(int*, int, int, int)"
"GPU activities",3.323252,38.147592,1,38.147592,38.147592,38.147592,"[CUDA memcpy DtoH]"
"GPU activities",0.780526,8.959656,142,0.063096,0.061185,0.069121,"phase1_cal(int*, int, int, int)"
"GPU activities",0.168075,1.929334,1,1.929334,1.929334,1.929334,"init_dist(int*, int)"
"GPU activities",0.052485,0.602471,1,0.602471,0.602471,0.602471,"[CUDA memcpy HtoD]"
"GPU activities",0.037029,0.425060,1,0.425060,0.425060,0.425060,"dist_fill(int*, int*, int, int)"
"API calls",75.597361,1143.268602,1,1143.268602,1143.268602,1143.268602,"cudaMemcpy2D"
"API calls",19.826541,299.839322,2,149.919661,78.728233,221.111089,"cudaHostAlloc"
"API calls",4.178000,63.184429,2,31.592214,0.924849,62.259580,"cudaFreeHost"
"API calls",0.164313,2.484931,1,2.484931,2.484931,2.484931,"cudaFree"
"API calls",0.151476,2.290792,428,0.005352,0.003463,0.524879,"cudaLaunchKernel"
"API calls",0.043479,0.657543,1,0.657543,0.657543,0.657543,"cudaMemcpy"
"API calls",0.026415,0.399471,2,0.199735,0.076017,0.323454,"cudaMalloc"
"API calls",0.010532,0.159276,114,0.001397,0.000099,0.067807,"cuDeviceGetAttribute"
"API calls",0.001036,0.015669,1,0.015669,0.015669,0.015669,"cuDeviceGetName"
"API calls",0.000610,0.009218,1,0.009218,0.009218,0.009218,"cuDeviceGetPCIBusId"
"API calls",0.000094,0.001420,3,0.000473,0.000100,0.001175,"cuDeviceGetCount"
"API calls",0.000059,0.000899,2,0.000449,0.000154,0.000745,"cuDeviceGet"
"API calls",0.000042,0.000632,1,0.000632,0.000632,0.000632,"cuDeviceTotalMem"
"API calls",0.000027,0.000401,1,0.000401,0.000401,0.000401,"cuModuleGetLoadingMode"
"API calls",0.000016,0.000241,1,0.000241,0.000241,0.000241,"cuDeviceGetUuid"
```

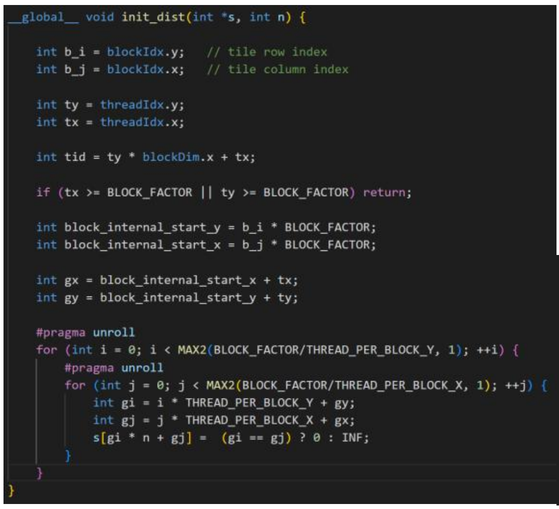


可以看到當 problem size 很小時，input 的時間佔比大，主要是因為 GPU kernel 很快就計算完了，隨著 problem size 越大，主要的 bottleneck 來到了 computation。

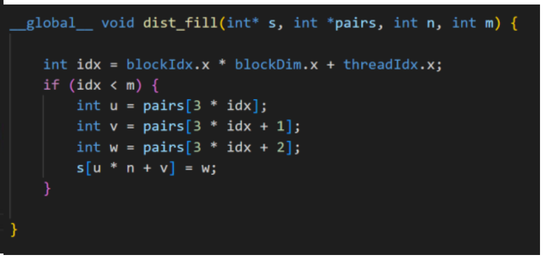
f. Others:

主要是針對 IO, H2D 及 D2H 去做一些細微的優化:

1. 用 cudaMemcpy2D 去 unpad 結果；
2. 用 cudaMallocHost 將 host memory pinned 住來達到較好的 H2D/D2H 的速度；
3. 改用 GPU kernel 去平行化 n*n data matrix 的 initialize 跟 assignment

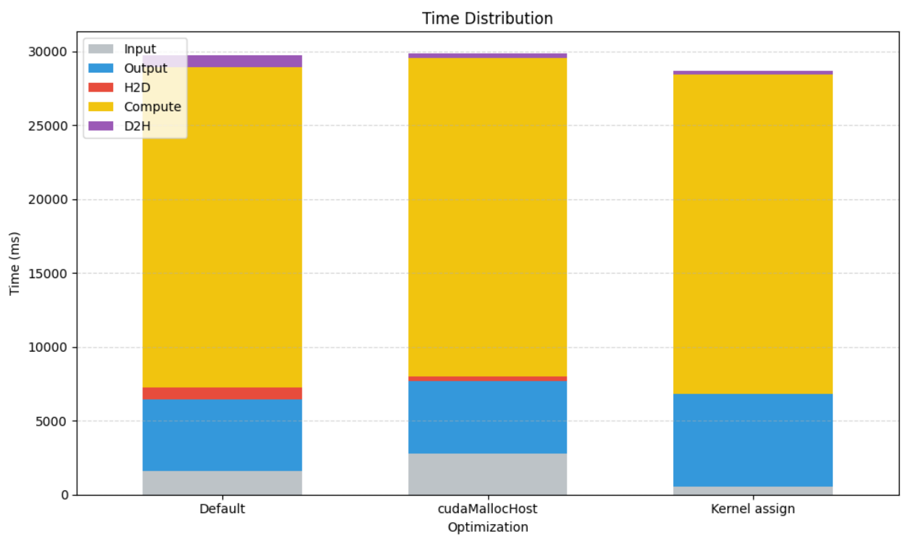


```
__global__ void init_dist(int *s, int n) {

    int b_i = blockIdx.y;   // tile row index
    int b_j = blockIdx.x;   // tile column index

    int ty = threadIdx.y;
    int tx = threadIdx.x;

    int tid = ty * blockDim.x + tx;

    if (tx >= BLOCK_FACTOR || ty >= BLOCK_FACTOR) return;

    int block_internal_start_y = b_i * BLOCK_FACTOR;
    int block_internal_start_x = b_j * BLOCK_FACTOR;

    int gx = block_internal_start_x + tx;
    int gy = block_internal_start_y + ty;

    #pragma unroll
    for (int i = 0; i < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_Y, 1); ++i) {
        #pragma unroll
        for (int j = 0; j < MAX2(BLOCK_FACTOR/THREAD_PER_BLOCK_X, 1); ++j) {
            int gi = i * THREAD_PER_BLOCK_Y + gy;
            int gj = j * THREAD_PER_BLOCK_X + gx;
            s[gi * n + gj] =  (gi == gj) ? 0 : INF;
        }
    }
}
```



```
__global__ void dist_fill(int* s, int *pairs, int n, int m) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < m) {
        int u = pairs[3 * idx];
        int v = pairs[3 * idx + 1];
        int w = pairs[3 * idx + 2];
        s[u * n + v] = w;
    }

}
```

這裡透過大的 testcase(p30k1)較能看出 3 所帶來的 input time 的差異:



AMD GPU Porting and Analysis

a. Implementation

Explain how you ported the CUDA code to HIP code. **(hw3-2):**

基本上 implementation 跟 hw3-2 cuda 版本的一樣，只是將 cuda 的 function 替換成 hip，例 cudaMalloc -> hipMalloc。

b. Experiment & Analyze

我是用 rocprof 去做 profiling，metrics 有 SQ_INSTS_VMEM_WR SQ_INSTS_VMEM_RD SQ_INSTS_VALU SQ_LDS_BANK_CONFLICT SQ_INSTS_LDS，由於 rocprof 沒有直接提供 shared/global memory throughput，所以只能透過 SQ_INSTS_LDS 跟

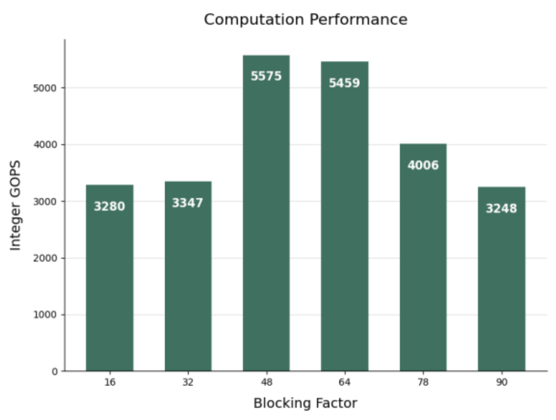
SQ_INSTS_VMEM_RD/WR 及 durations 去算，各算法如下:

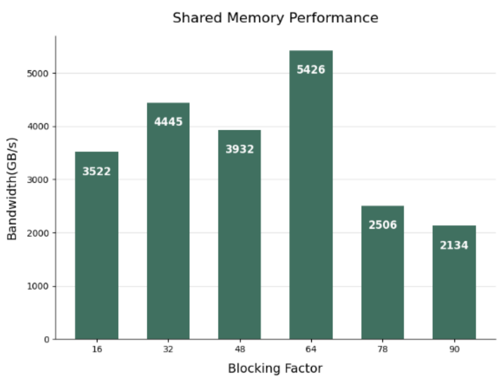
$$Int\ GOPS = \frac{\sum_{i=1}^{3} I_{phase_{i}}{}_{avg}}{\sum_{i=1}^{3} t_{phase\_i}{}_{avg}(ns) * 10^{-9})} * 10^{-9}$$

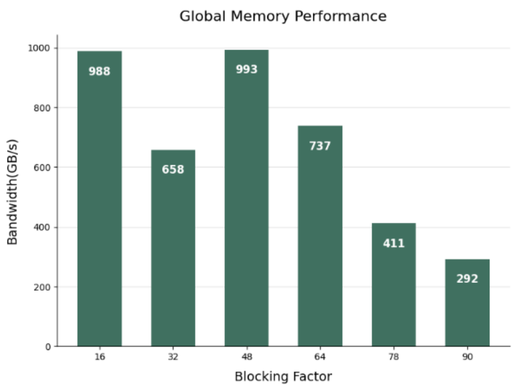
$$Bandwidth = \frac{\sum_{i=1}^{3} R_{phase_{i}}{}_{avg}}{\sum_{i=1}^{3} t_{phase\_i}{}_{avg}(ns) * 10^{-9})} * 10^{-9}$$

1. $I_{phase\_i}{}_{avg}$: 各 phase 一個 invocation 的平均 integer instructions，取自 SQ_INSTS_VALU*64(wavefront size)；
2. $R_{phase\_i}{}_{avg}$: 各 phase 一個 invocation 的平均 shared mem/global mem 的 instructions，shared mem : SQ_INSTS_LDS*64*4 , global mem: SQ_INSTS_VMEM_RD/WR *64*4bytes；
3. $t_{phase\_i}{}_{avg}$: 各 phase 一個 invocation 的平均執行時間，單位為 ns；

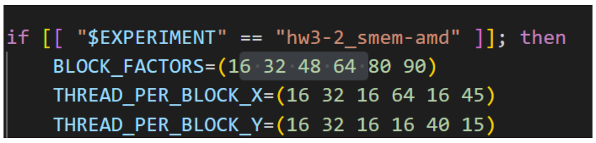
1. **Blocking Factor:**







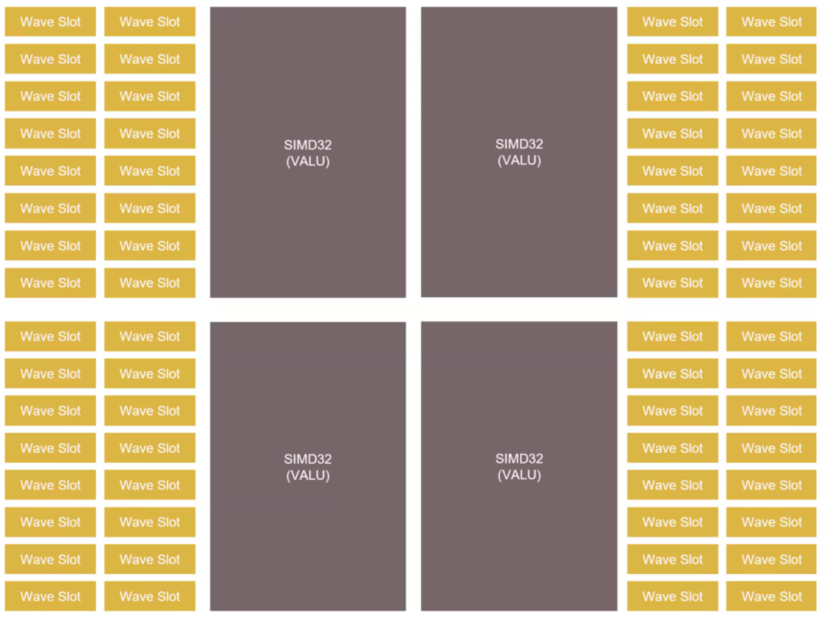
Testcase 為 c20.1，Block_Factor 與 threads 數設定如下:



```
if [[ "$EXPERIMENT" == "hw3-2_smem-amd" ]]; then
    BLOCK_FACTORS=(16 32 48 64 80 90)
    THREAD_PER_BLOCK_X=(16 32 16 64 16 45)
    THREAD_PER_BLOCK_Y=(16 32 16 16 40 15)
```

由於這張 AMD GPU 的 shared memory per block 較大(65536 bytes)，所以 BF 可以設的比 GTX 1080 還大，可以看到，BF 超過 64 整體 GOPS 就開始下降了，可以看到即便 BF=80 跟 90 拉高了 ILP，也很難像 GTX 1080 一樣提升 performance，且相較於 NV 的 warp size 為 32，AMD 的 warp size(Wavefront)為 64，而 MI210 這張卡為 CDNA2 架構，每個 CU 包含 4 組 16-wide 的 SIMD(而 NV 是 32 wide)，因此 VALU instructions 實際會分成 4 個 SIMD slice，以 4 個 cycle 完成一個 wavefront 的 64 threads(下圖為 RDNA 架構做為參考)，也就是說，只要確保每個 SIMD16 slice（即 wavefront 的 0~15、16~31、32~47、48~63 這四組 lanes）各自內部的 threads 都沒有 bank conflicts，那麼整個 wavefront 就不會有 bank conflicts。

此為 RDNA 架構，SIMD 為 32-wide:

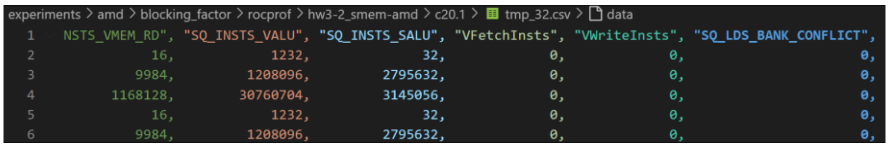


source: https://gpuopen.com/learn/occupancy-explained/

2. For the AMD GPU, perform **one optimization** that was not done on the NVIDIA GPU, or **modify one NVIDIA-specific optimization** to suit the AMD GPU. Explain and show why this optimization is effective or ineffective, and clarify why it is specific to the AMD GPU.

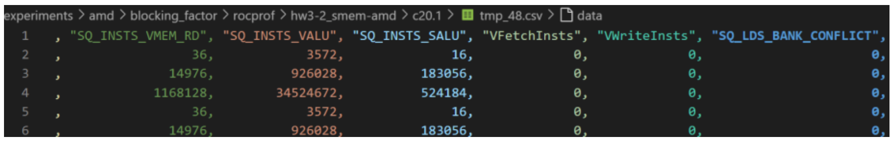
   基於 CDNA2 架構 16-wide SIMD 的架構，我們可以在現行的 access pattern 下，將 BF 及 blockDim.x 皆設為 16 的倍數，便可確保每個 simd instructions 的 shared mem access 不會跨 row，也就不會有 bank conflicts 發生:

BF=32, threads=32*32:



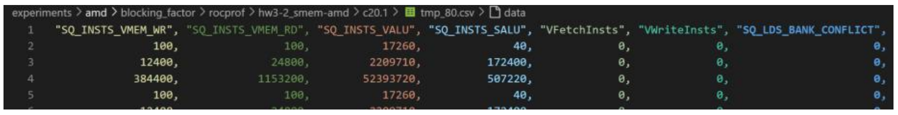
```
experiments > amd > blocking_factor > rocprof > hw3-2_smem-amd > c20.1 > tmp_32.csv > data
NSTS_VMEM_RD", "SQ_INSTS_VALU", "SQ_INSTS_SALU", "VFetchInsts", "VWriteInsts", "SQ_LDS_BANK_CONFLICT",
           16,           1232,             32,             0,             0,                      0,
         9984,        1208096,        2795632,             0,             0,                      0,
      1168128,       30760704,        3145056,             0,             0,                      0,
           16,           1232,             32,             0,             0,                      0,
         9984,        1208096,        2795632,             0,             0,                      0,
```

BF=48, threads=16*16:


experiments > amd > blocking_factor > rocprof > hw3-2_smem-amd > c20.1 > tmp_48.csv > data

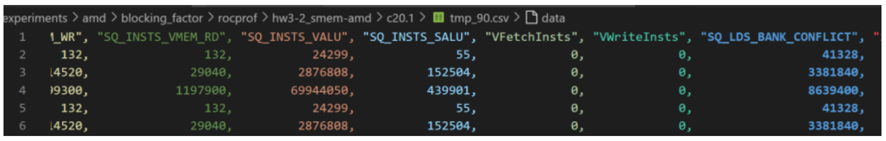
```
 , "SQ_INSTS_VMEM_RD", "SQ_INSTS_VALU", "SQ_INSTS_SALU", "VFetchInsts", "VWriteInsts", "SQ_LDS_BANK_CONFLICT",
 ,                36,           3572,              16,             0,             0,                      0,
 ,             14976,         926028,          183056,             0,             0,                      0,
 ,           1168128,       34524672,          524184,             0,             0,                      0,
 ,                36,           3572,              16,             0,             0,                      0,
 ,             14976,         926028,          183056,             0,             0,                      0,
```

BF=80, threads=16*40:


experiments > amd > blocking_factor > rocprof > hw3-2_smem-amd > c20.1 > tmp_80.csv > data

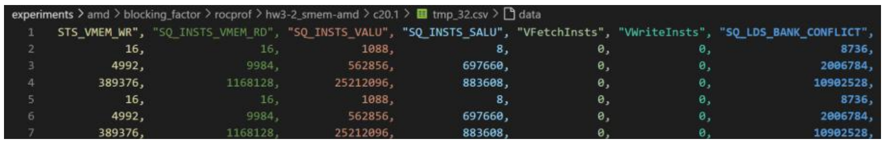
```
"SQ_INSTS_VMEM_WR", "SQ_INSTS_VMEM_RD", "SQ_INSTS_VALU", "SQ_INSTS_SALU", "VFetchInsts", "VWriteInsts", "SQ_LDS_BANK_CONFLICT",
               100,                100,           17260,              40,             0,             0,                      0,
             12400,              24800,         2209710,          172400,             0,             0,                      0,
            384400,            1153200,        52393720,          507220,             0,             0,                      0,
               100,                100,           17260,              40,             0,             0,                      0,
```

BF=90, threads=45*15:


experiments > amd > blocking_factor > rocprof > hw3-2_smem-amd > c20.1 > tmp_90.csv > data

```
1_WR", "SQ_INSTS_VMEM_RD", "SQ_INSTS_VALU", "SQ_INSTS_SALU", "VFetchInsts", "VWriteInsts", "SQ_LDS_BANK_CONFLICT", "
  132,                132,           24299,              55,             0,             0,                  41328,
l4520,              29040,         2876808,          152504,             0,             0,                3381840,
)9300,            1197900,        69944050,          439901,             0,             0,                8639400,
  132,                132,           24299,              55,             0,             0,                  41328,
l4520,              29040,         2876808,          152504,             0,             0,                3381840,
```

BF=32, threads=8*32:

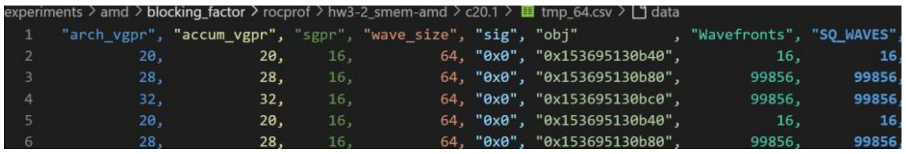

experiments > amd > blocking_factor > rocprof > hw3-2_smem-amd > c20.1 > tmp_32.csv > data

```
STS_VMEM_WR", "SQ_INSTS_VMEM_RD", "SQ_INSTS_VALU", "SQ_INSTS_SALU", "VFetchInsts", "VWriteInsts", "SQ_LDS_BANK_CONFLICT",
          16,                 16,            1088,               8,             0,             0,                   8736,
        4992,               9984,          562856,          697660,             0,             0,                2006784,
      389376,            1168128,        25212096,          883608,             0,             0,               10902528,
          16,                 16,            1088,               8,             0,             0,                   8736,
        4992,               9984,          562856,          697660,             0,             0,                2006784,
      389376,            1168128,        25212096,          883608,             0,             0,               10902528,
```

當 blockDim.x 為 8 時因為不滿 16，所以會跨 2 個 rows，導致 2-way bank conflict。

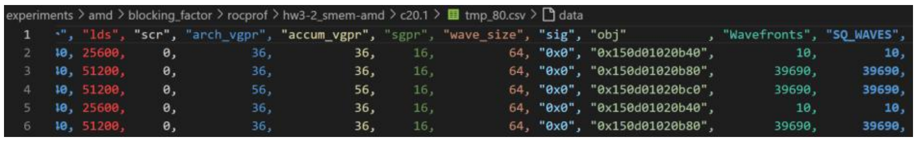
另外，有發現在 AMD 上 thread coarsening 的效益不如在 NVIDIA 上明顯，主要可能的原因為 AMD CDNA2 架構 CU 上一個 wave 執行一個 VALU instructions 需要 4 個 SIMD cycles 去執行，所以降低 occupancy 可能會導致 CU 上沒有足夠的 waves 給 warp scheduler 去做切換來掩蓋 latency，透過 SQ_waves 能算出 occupancy:

BF=64, threads=64*16:


experiments > amd > blocking_factor > rocprof > hw3-2_smem-amd > c20.1 > tmp_64.csv > data

```
"arch_vgpr", "accum_vgpr", "sgpr", "wave_size", "sig", "obj"                 , "Wavefronts", "SQ_WAVES"
         20,           20,     16,          64, "0x0", "0x153695130b40",              16,         16
         28,           28,     16,          64, "0x0", "0x153695130b80",           99856,      99856
         32,           32,     16,          64, "0x0", "0x153695130bc0",           99856,      99856
         20,           20,     16,          64, "0x0", "0x153695130b40",              16,         16
         28,           28,     16,          64, "0x0", "0x153695130b80",           99856,      99856
```

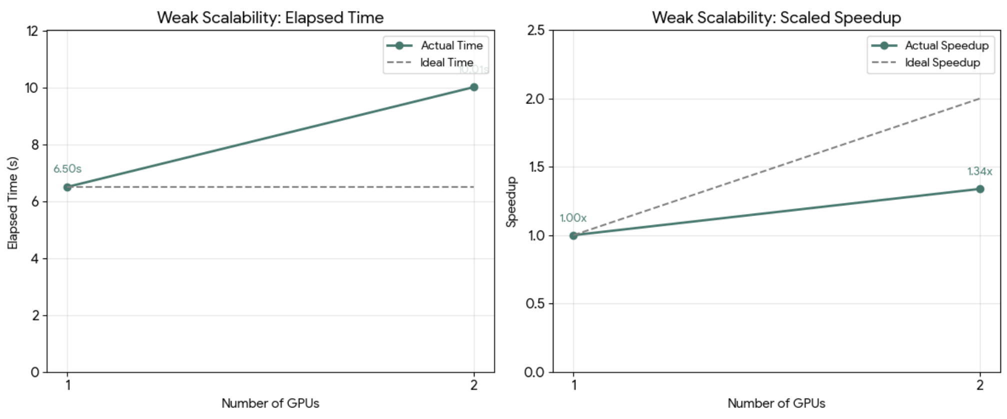
BF=80, threads=16*40:


experiments > amd > blocking_factor > rocprof > hw3-2_smem-amd > c20.1 > tmp_80.csv > data

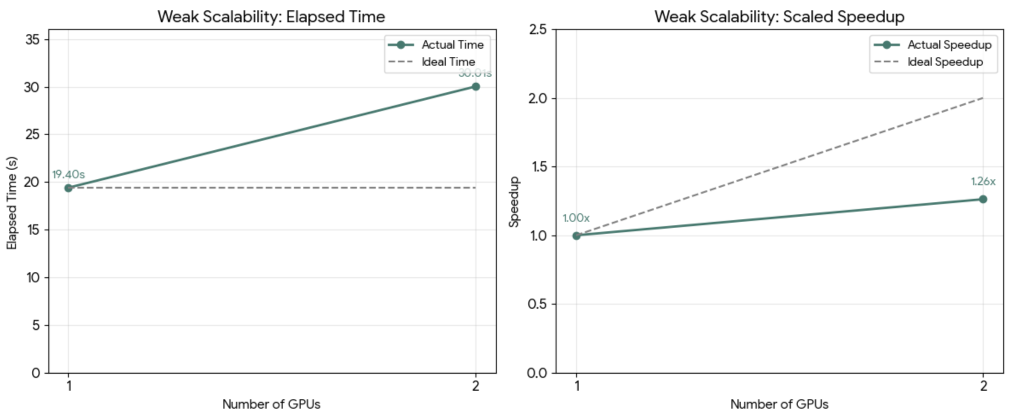
```
", "lds", "scr", "arch_vgpr", "accum_vgpr", "sgpr", "wave_size", "sig", "obj"                 , "Wavefronts", "SQ_WAVES",
10, 25600,     0,          36,           36,     16,          64, "0x0", "0x150d01020b40",              10,         10,
10, 51200,     0,          36,           36,     16,          64, "0x0", "0x150d01020b80",           39690,      39690,
10, 51200,     0,          56,           56,     16,          64, "0x0", "0x150d01020bc0",           39690,      39690,
10, 25600,     0,          36,           36,     16,          64, "0x0", "0x150d01020b40",              10,         10,
10, 51200,     0,          36,           36,     16,          64, "0x0", "0x150d01020b80",           39690,      39690,
```

3. According to Section 3-d, also conduct a **weak-scaling** experiment on the AMD GPU, and compare as well as discuss the results between the NVIDIA and AMD GPUs.

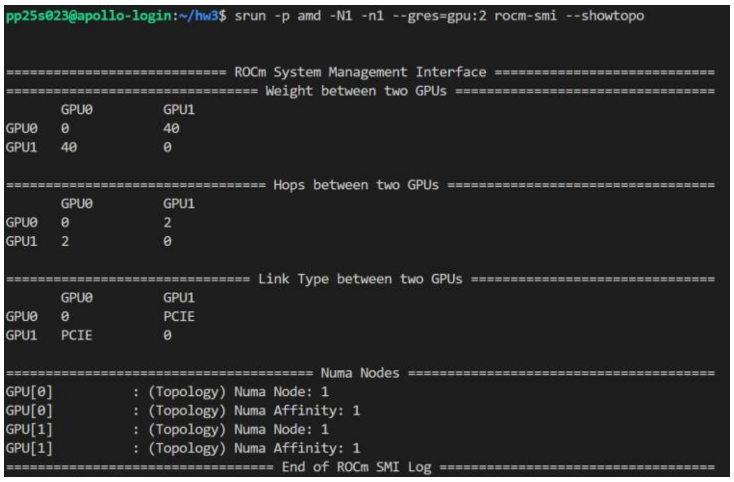
p11k1 v.s. p14k1:



p20k1 v.s. p25k1:



透過 weak scalability 的實驗可看出，AMD 跟 Nvidia 在 multi-gpu 時都因 communication 的成本難以達到 ideal speedup，甚至 speedup 比 Nvidia 還稍差一些，可以看到 GPU 間為透過 PCIE 溝通，但 hops 數為 2，應該是類似像 Nvidia PHB 有經過 CPU，所以 overhead 大:



```
pp25s023@apollo-login:~/hw3$ srun -p amd -N1 -n1 --gres=gpu:2 rocm-smi --showtopo


============================ ROCm System Management Interface ============================
================================ Weight between two GPUs =================================
       GPU0         GPU1
GPU0   0            40
GPU1   40           0

================================= Hops between two GPUs ==================================
       GPU0         GPU1
GPU0   0            2
GPU1   2            0

=============================== Link Type between two GPUs ===============================
       GPU0         GPU1
GPU0   0            PCIE
GPU1   PCIE         0

======================================= Numa Nodes =======================================
GPU[0]          : (Topology) Numa Node: 1
GPU[0]          : (Topology) Numa Affinity: 1
GPU[1]          : (Topology) Numa Node: 1
GPU[1]          : (Topology) Numa Affinity: 1
================================== End of ROCm SMI Log ===================================
```

Experience & conclusion

a. What have you learned from this homework?

透過這次作業實作了多種 GPU programming 的優化技巧，深刻體會到 memory access 對效能的影響至關重要，其中像是 shared memory access pattern、global memory coalescing、BLOCK_FACTOR 的選擇等，都會極大影響 cuda 程式的效能，而 profiler 讓我在優化的過程中可以清楚看出程式哪邊還需要優化，以及讓我注意到了一些做 lab 時沒注意到的細節，此外，我也發現在使用 shared memory 的情況下，occupancy 並不是越大越好，降低 threads 數並適度地提高每個 thread 的工作量也可以提升整體效率；Multi-GPU 的優化則是比預期困難，GPU 間資料交換極大影響了整體的效能，很難達到 ideal 加速。