Parallel Programming HW3

112164513 陳彥凱

Implementation

- Which algorithm do you choose in hw3-1?
  - 使用 Blocked Floyd-Warshall Algorithm 來實作 hw3-1。
- How do you divide your data in hw3-2, hw3-3?

```
Device 0: "NVIDIA GeForce GTX 1080"
  CUDA Driver Version / Runtime Version          12.3 / 11.8
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 8112 MBytes (8506114048 bytes)
  (20) Multiprocessors, (128) CUDA Cores/MP:     2560 CUDA Cores
  GPU Max Clock rate:                            1734 MHz (1.73 GHz)
  Memory Clock rate:                             5005 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
```

  - 因為從 devicequery 中得知每個 GPU block 最多有 1024 個 threads，因此 blocking factor (block_size) 就設為 32。
  - hw3-2:

    將 n*n 的 **Dist** 做 padding 使 n(vertex_num) 變成 32 的倍數 (blocking factor)，接著以 32*32 筆 data 為單位送進一個 GPU block 中執行，在每個 block 中會有 32*32 個 thread，每個 thread 負責處理 1 筆 data。
  - hw3-3:

    首先一樣做 padding，然後以 32*32 筆 data 為單位，送到一個 GPU block 中執行，每個 thread 處理 1 筆 data。因為有兩個 GPU，所以將 data 平分為兩半，在 phase3 時，將上半部的 data 交給 GPU0 執行，下半部的 data 交給 GPU1 執行，並在每一個 round 開始前，將當前 pivot row 的 data 傳給另一個 GPU。
- What's your configuration in hw3-2, hw3-3? And why? (e.g. blocking factor, #blocks, #threads)
  - hw3-2
    - blocking factor

      設定 blocking factor 為 32，因為最多有 1024 個 threads。在 share memory 的利用上，因為在 phase2, 3 時需要兩個 share memory s1, s2 儲存 data (Dist[i][j] = Dist[i][k] + Dist[k][j])，而每個 int 為 4 bytes，也就是使用了 2*32*32*4 = 8 kb。

- blocks & threads

  根據 blocked floyd warshall algorithm，在不同階段，block 的數量會有所變化，而總共需要執行 vertex_num / block_size 個 round。在 phase1 中，僅需計算 pivot block，因此需要 1 個 block；而在 phase2 中，需要計算跟 pivot block 同一個 row 和 column 的所有 block (pivot row, pivot column)，再扣掉 phase1 已計算好的 pivot，因此需要 2 * (round-1) 個 block；在 phase3 中，扣除 phase1 以及 phase2 計算過的部分，總共需要 (round-1) * (round-1) 個 block。

  每個 thread 都負責一個 block 中某個 (i,j) 的計算，而每一個 GPU block 都負責一個 floyd warshall block 的運算。

```
int round = vertex_num/Block_Size;
for(int block_id = 0; block_id < round; block_id++){
    phase1 <<<1, dim3(Block_Size, Block_Size)>>>
        (Block_Size, block_id, Dist_GPU, vertex_num, pitch/ sizeof(int));
    phase2 <<<dim3(2, round-1), dim3(Block_Size, Block_Size)>>>
        (Block_Size, block_id, Dist_GPU, vertex_num, pitch/ sizeof(int));
    phase3 <<<dim3(round-1, round-1), dim3(Block_Size, Block_Size)>>>
        (Block_Size, block_id, Dist_GPU, vertex_num, pitch/ sizeof(int));
}
```

- hw3-3
  - 與 hw3-2 的差別只有 phase3，因為 phase3 的計算量是最主要的((round-1)*(round-1))。直接切成上下兩半分別給 GPU0 跟 GPU1 計算，如果是奇數則將多的 row 給 GPU1，所以各自需要 (end_round - start_round) * (round - 1) 個 block，而 phase 1、phase 2 還是可以兩個 GPU 個別計算。

```
  int round = vertex_num/Block_Size;

  int id = omp_get_thread_num();//device id
  cudaSetDevice(id);

  int start_round = (round/2)*id;
  int end_round = start_round + (round/2) + (round%2)*id;

  #pragma omp barrier

  // From host to device
  cudaMemcpy(Dist_GPU[id]+(start_round * Block_Size * vertex_num),
           Dist + ( start_round * Block_Size * vertex_num),
           (end_round - start_round) * Block_Size * vertex_num * sizeof(int),
           cudaMemcpyHostToDevice);

  for(int block_id = 0; block_id < round; block_id++){
      bool is_copy_need = (block_id >= start_round && block_id < end_round);

      cudaMemcpyPeer(Dist_GPU[!id] + (block_id * Block_Size * vertex_num), !id,
               Dist_GPU[id]+(block_id * Block_Size * vertex_num), id,
               is_copy_need * Block_Size * vertex_num * sizeof(int));
      #pragma omp barrier

      phase1 <<<1, dim3(Block_Size, Block_Size)>>>
            (Block_Size, block_id, Dist_GPU[id], vertex_num);
      phase2 <<<dim3(2, round-1), dim3(Block_Size, Block_Size)>>>
            (Block_Size, block_id, Dist_GPU[id], vertex_num);
      phase3 <<<dim3(end_round - start_round, round-1), dim3(Block_Size, Block_Size)>>>
            (Block_Size, block_id, Dist_GPU[id], vertex_num, start_round);
```

- How do you implement the communication in hw3-3?
  - 在每個 round 負責該 pivot row 的 GPU 需要將 pivot row 的資料同步到另一個 GPU 上，因此使用 cudaMemcpyPeer 傳遞，若透過 device to host/ host to device 會很慢。

```
bool is_copy_need = (block_id >= start_round && block_id < end_round);

cudaMemcpyPeer(Dist_GPU[!id] + (block_id * Block_Size * vertex_num), !id,
```

```
            Dist_GPU[id]+(block_id * Block_Size * vertex_num), id,
            is_copy_need * Block_Size * vertex_num * sizeof(int));
```

- Briefly describe your implementations in diagrams, figures or sentences.
  - hw3-1 (cpu version):
    - 基本上跟 seq.cc 相同，只有在 **cal()** 使用了 OpenMp dynamic schedule 做加速。
    - 而經過測試 (blocking factor = 32, 64, 128) ，blocking factor 設為 64 的計算速度會是最快的。
    - **dist_ik** 可以提出去迴圈記錄起來，這樣每次在算 **Dist[i][j] = min(Dist[i][j], dist_ik + Dist[k][j])** 時就不需要重複的讀取。

```
  void cal(int B, int Round, int block_start_x, int block_start_y, int block_width, int
block_height) {
  int block_end_x = block_start_x + block_height;
  int block_end_y = block_start_y + block_width;
  //int k_start = Round * B, k_end = min((Round + 1) * B, n);

  #pragma omp parallel for num_threads(cpu_num) schedule(dynamic)
  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) {
          for (int k = Round * B; k < (Round + 1) * B && k < n; ++k) {
              int block_internal_start_x = b_i * B;
              int block_internal_end_x = min((b_i + 1) * B, n);
              int block_internal_start_y = b_j * B;
              int block_internal_end_y = min((b_j + 1) * B, n);

              for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
                  int dist_ik = Dist[i][k];
                  for (int j = block_internal_start_y; j < block_internal_end_y; ++j) {
                      Dist[i][j] = min(Dist[i][j], dist_ik + Dist[k][j]);
                  }
              }
          }
      }
    }
}
```

- hw3-2 (single GPU):
  - Padding
    - 測資給定的 vertex_num 不一定是 blocking factor 的倍數，如果沒有做 padding 會導致在 kernel code 裡需要多做判斷，來確定有沒有超過陣列的範圍，因此用額外的空間來換取速度的提升。
    - 在讀入 vertex_num 之後，放大到 blocking factor 的倍數，假設有更多的點，因為其他的點都是沒有邊的，因此不影響原圖的計算結果。使用新算出來的 vertex_num 計算完畢後，寫回檔案時只要把部分矩陣寫回即可。

```
vertex_num_origin = vertex_num;
// Padding
vertex_num += (Block_Size - vertex_num % Block_Size);
```

```
FILE *file = fopen(outFileName, "w");
for(int i = 0; i < n_origin; i++){
    fwrite(&Dist[i*vertex_num], sizeof(int), vertex_num_origin, file);
}
fclose(file);
```

  - CUDA 2D Alignment & Pin Memory

- 對於 host memory → device global memory 的優化，除了使用 **cudaMallocHost** pin 住 host memory 來提升 I/O 的速度，另外使用 **cudaMallocPitch** 以及 **cudaMemcpy2D** 來將 2D 的陣列對齊，使得 memory access 的速度可以提升。
- 因此在計算 global memory 的索引時，都將原本 i*n+j 替換成 i*pitch+j。其中 pitch 是 **cudaMallocPitch** 時回傳的 **pitch** 代表每個 row 的長度（bytes）再除以 integer 的大小 sizeof(int)。

```
size_t pitch;
cudaMallocPitch(&Dist_GPU, &pitch, vertex_num * sizeof(int), vertex_num);
cudaMemcpy2D(Dist_GPU, pitch, Dist, vertex_num * sizeof(int), vertex_num * sizeof(int),
                                                vertex_num, cudaMemcpyHostToDevice);
```

- Kernel & Share Memory

  將 3 個 phase 分別寫成 3 個 kernel 來執行，每個 kernel 的 block 數就是要計算的 block 數量，每個 block有 block_szie*block_size 個 thread，每個 thread 負責 1 筆資料。

  - **Phase 1**
  - 使用一個 block_size*block_size 大小的 share memory，紀錄 block 中每個位置的值。首先把 thread 負責的 data 都複製到 share memory 中：

```
// Get index of thread
int b_i = block_id<<LBS;
int b_j = block_id<<LBS;
int i = threadIdx.y;
int j = threadIdx.x;

// Copy data from global memory to share memory
__shared__ int s[Block_Size][Block_Size];
s[i][j] = Dist_GPU[(b_i+i)*pitch+(b_j+j)];
```

  - 再來就是計算結果，因為 phase 1 是 dependent phase（跟一般的 floyd warshall 一樣），每一輪的結果都依賴於上一輪的結果，因此需要 **__syncthreads()** 來同步其他 threads 的計算結果。

```
// Compute phase 1 - dependent phase
#pragma unroll
for(int k = 0; k < Block_Size; k++){
    __syncthreads();
    s[i][j] = min(s[i][j], s[i][k]+s[k][j]);
}
```

  - 最後再將值更新回 global memory 中：

```
// Load data from shared memory to global memory
Dist_GPU[(b_i+i)*pitch+(b_j+j)] = s[i][j];
```

  - **Phase 2**
  - 首先開兩個大小為 block_size*block_size 大小的 int 陣列，負責儲存所有這個 block 會讀取到的 memory，s1 儲存在做 floyd-warshall 時的 Dist[i][k] 所屬的 block，後半部儲存 Dist[k][j] 所屬的 block，最後 **__syncthreads()** 確認資料載入完畢：

```
// Get index of thread
// ROW: (blockIdx.x = 1), COL: (blockIdx.y = 0)
int b_i = (blockIdx.x*block_id+(!blockIdx.x)*(blockIdx.y+(blockIdx.y>=block_id)))<<LBS;
int b_j = (blockIdx.x*(blockIdx.y+(blockIdx.y>=block_id))+(!blockIdx.x)*block_id)<<LBS;
int b_k = block_id<<LBS;
int i = threadIdx.y, j = threadIdx.x;

__shared__ int s1[Block_Size][Block_Size], s2[Block_Size][Block_Size];
int new_dist = Dist_GPU[(b_i+i)*pitch+(b_j+j)];
```

```
s1[i][j] = Dist_GPU[(b_i+i)*pitch+(b_k+j)];
s2[i][j] = Dist_GPU[(b_k+i)*pitch+(b_j+j)];

__syncthreads();
```

- 接著開始 phase 2 的計算，phase 2 的運算只依賴於同一個點的運算結果以及 phase 1 的運算結果，因此不需要在每一輪都做同步：

```
// Compute phase 2 - partial dependent phase
#pragma unroll
for(int k = 0; k < Block_Size; k++){
    new_dist = min(new_dist, s1[i][k]+s2[k][j]);
}
```

- 最後將 share memory 中的資料載入回 global memory 中：

```
// Load data from share memory to global memory
Dist_GPU[(b_i+i)*pitch+(b_j+j)] = new_dist;
```

- **Phase 3**
- 首先一樣開兩個大小為 block_size*block_size 大小的 int 陣列，負責儲存所有這個 block 會讀取到的 memory，s1 儲存在做 floyd-warshall 時的 Dist[i][k] 所屬的 block，後半部儲存 Dist[k][j] 所屬的 block，最後 **__syncthreads()** 確認資料載入完畢：

```
int b_i = (blockIdx.x+(blockIdx.x>=block_id))<<LBS;
int b_j = (blockIdx.y+(blockIdx.y>=block_id))<<LBS;
int b_k = block_id<<LBS;
int i = threadIdx.y;
int j = threadIdx.x;

__shared__ int s1[Block_Size][Block_Size], s2[Block_Size][Block_Size];
int new_dist = Dist_GPU[(b_i+i)*pitch+(b_j+j)];
s1[i][j] = Dist_GPU[(b_i+i)*pitch+(b_k+j)];
s2[i][j] = Dist_GPU[(b_k+i)*pitch+(b_j+j)];

__syncthreads();
```

- 再來一樣做計算，phase 3 只依賴於同一個點的運算結果以及 phase 2 的運算結果，因此也不需要做額外的同步：

```
// Compute phase 3 - independent phase
#pragma unroll
for(int k = 0; k < Block_Size; k++){
    new_dist = min(new_dist, s1[i][k]+s2[k][j]);
}
```

- 最後將 share memory 中的資料載入回 global memory 中：

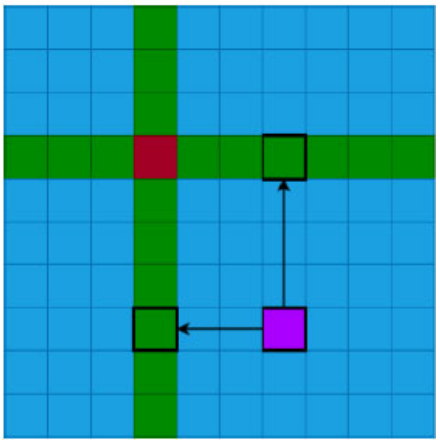
```
// Load data from share memory to global memory
Dist_GPU[(b_i+i)*pitch+(b_j+j)] = new_dist;
```

- Prevent branching & Unroll

  使用 unroll 將迴圈展開，可以減少 branching。 另外不使用 if-else 來做運算，而改用 cuda 提供的 min 函數，一樣可以降低 branching 提高運算速度。

- hw3-3 (mutli-GPU)

- 使用 OpenMP 來開啟兩個 threads，分別操作兩個 GPU 的工作。使用兩個 GPU 的版本跟 Single-GPU 差不多，優化的是 phase3 計算部分，因為 phase3 的運算量是最主要的，phase 1、phase 2 還是由兩個 GPU 個別計算。
- 將 phase3 要計算的區塊直接平分為上下兩半。以計算下半部的 GPU1 為例，在 phase3 的某一次計算(紫色的格子)只依賴對應於 pivot row 和 pivot column 的這兩個資料(兩個箭頭分別指向的黑色框框)，以 pivot row 為界線，上半部的 pivot column 不會參與計算。所以透過傳遞 pivot row， GPU1 便能獲得正確的 dependency。



- 總的來說，在每次計算之前，負責計算該 pivot row 的 GPU 只需將該 pivot row 傳送給另一個 GPU。這樣另一個 GPU 就能使用正確的 pivot row 進行計算。而上/下半部的 pivot column 一開始就已經傳送過，所以這樣兩個 GPU 都各自獲得了正確的 dependency，也就能得到正確的計算結果。

```
// Block FW using 2 GPUs
void block_FW(int BS){
    int round = vertex_num/Block_Size;
    #pragma omp parallel num_threads(2)
    {
        // Set device
        int id = omp_get_thread_num();//device id
        cudaSetDevice(id);

        int start_round = (round/2)*id;
        int end_round = start_round + (round/2) + (round%2)*id;

        cudaMalloc(&Dist_GPU[id], vertex_num*vertex_num*sizeof(int));
        #pragma omp barrier

        // From host to device
        cudaMemcpy(Dist_GPU[id]+(start_round * Block_Size * vertex_num),
                   Dist + ( start_round * Block_Size * vertex_num),
                   (end_round - start_round) * Block_Size * vertex_num * sizeof(int),
                   cudaMemcpyHostToDevice);

        for(int block_id = 0; block_id < round; block_id++){
            bool is_copy_need = (block_id >= start_round && block_id < end_round);

            cudaMemcpyPeer(Dist_GPU[!id] + (block_id * Block_Size * vertex_num), !id,
                           Dist_GPU[id]+(block_id * Block_Size * vertex_num), id,
                           is_copy_need * Block_Size * vertex_num * sizeof(int));
            #pragma omp barrier

            phase1 <<<1, dim3(Block_Size, Block_Size)>>>
                (Block_Size, block_id, Dist_GPU[id], vertex_num);
            phase2 <<<dim3(2, round-1), dim3(Block_Size, Block_Size)>>>
                (Block_Size, block_id, Dist_GPU[id], vertex_num);
            phase3 <<<dim3(end_round - start_round, round-1), dim3(Block_Size, Block_Size)>>>
                (Block_Size, block_id, Dist_GPU[id], vertex_num, start_round);
        }
        cudaMemcpy(Dist + (start_round * Block_Size * vertex_num),
                   Dist_GPU[id] + (start_round * Block_Size * vertex_num),
                   (end_round - start_round) * Block_Size * vertex_num * sizeof(int),
                   cudaMemcpyDeviceToHost);
```

```
        cudaFree(Dist_GPU[id]);
    }
}
```

Profiling Results (hw3-2)

- 使用的是 p11k1 這筆測資來做 profiling，Vertex num: 11000、Edge num: 505586，太大的資料會使得 profiling timeout，因此選擇較小的測資。可以觀察到在 phase2 和 phase3 處理的資料量較龐大。在 occupancy、sm efficiency 以及各種 throughput 方面，phase2 跟 phase3 都明顯優於只處理 1 個 block 的 phase1。這表示了更有效地利用資源，使其能夠處理更大量的資料。

```
==3359896== Profiling application: ./hw3-2-32 /home/pp23/share/hw3-2/cases/p11k1 /dev/null
==3359896== Profiling result:
==3359896== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase1(int, int, int*, int, unsigned long)
        190                        achieved_occupancy                        Achieved Occupancy    0.497045    0.497544    0.497297
        190                            sm_efficiency                   Multiprocessor Activity      1.93%       3.75%       3.64%
        190                    shared_load_throughput             Shared Memory Load Throughput  56.098GB/s  62.112GB/s  59.802GB/s
        190                   shared_store_throughput            Shared Memory Store Throughput  19.284GB/s  21.351GB/s  20.557GB/s
        190                            gld_throughput                    Global Load Throughput  598.38MB/s  662.53MB/s  637.89MB/s
        190                            gst_throughput                   Global Store Throughput  598.38MB/s  662.53MB/s  637.89MB/s
    Kernel: phase2(int, int, int*, int, unsigned long)
        190                        achieved_occupancy                        Achieved Occupancy    0.888863    0.897963    0.894279
        190                            sm_efficiency                   Multiprocessor Activity      78.16%      95.17%      94.22%
        190                    shared_load_throughput             Shared Memory Load Throughput  2461.7GB/s  2720.8GB/s  2667.9GB/s
        190                   shared_store_throughput            Shared Memory Store Throughput  102.57GB/s  113.37GB/s  111.16GB/s
        190                            gld_throughput                    Global Load Throughput  153.86GB/s  170.05GB/s  166.75GB/s
        190                            gst_throughput                   Global Store Throughput  51.286GB/s  56.684GB/s  55.582GB/s
    Kernel: phase3(int, int, int*, int, unsigned long)
        189                        achieved_occupancy                        Achieved Occupancy    0.914049    0.915354    0.914713
        189                            sm_efficiency                   Multiprocessor Activity      99.87%      99.96%      99.95%
        190                    shared_load_throughput             Shared Memory Load Throughput  2511.7GB/s  2529.0GB/s  2521.9GB/s
        189                   shared_store_throughput            Shared Memory Store Throughput  104.65GB/s  105.37GB/s  105.08GB/s
        190                            gld_throughput                    Global Load Throughput  156.98GB/s  158.06GB/s  157.62GB/s
        190                            gst_throughput                   Global Store Throughput  52.326GB/s  52.687GB/s  52.540GB/s
```

Experiment and Analysis

- System Spec

  使用課程提供的 hades server 做實驗。

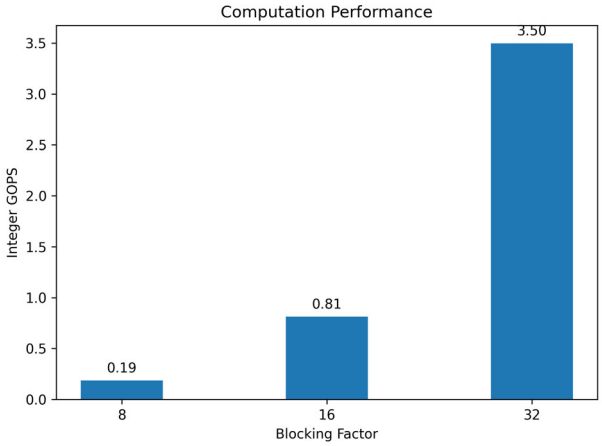
- Blocking Factor

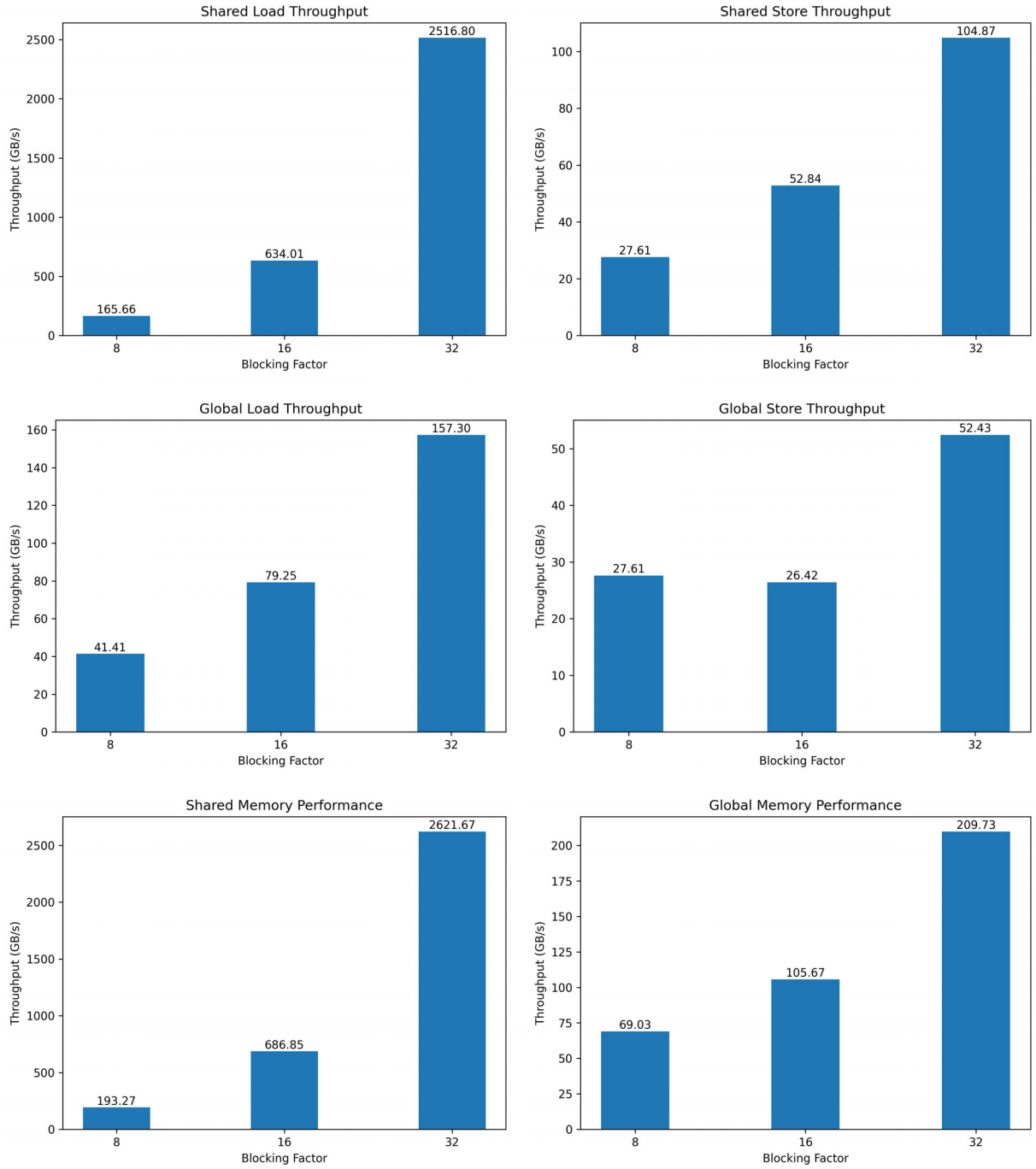
  由於使用過大的測資會使得 profiling timeout，因此這裡同樣使用較小的 p11k1 測資來做測試，Vertex num: 11000、Edge num: 505586，以下為 phase3 的數據。

  - 測量方式：

    GOPS 計算為總 integer instruction 數量除以 total time。除了 Share/Global Memory bandwidth 各自的 load/store 數據外，總體 Share/Global Memory Performance 的部分則是將 load 與 store 加總。

```
METRIC=inst_integer,shared_load_throughput,shared_store_throughput,gld_throughput,gst_throughput
srun -p prof -N1 -n1 --gres=gpu:1 nvprof -m $METRIC ./hw3-2-$i $IN $OUT
```





觀察圖表可以發現當 blocking factor 越大時 computation performance 就越好。而這次作業 I/O 的部分較難優化，所以最主要優化的部分是 computation，因此選擇 blocking factor 32 在我的實作中是最好的。Blocking factor 為 64 時，會需要進一步修改 code 將每個 thread 改為一次處理 4 筆資料，由於來不及實作完成因此無法測試。總體而言，share memory 的效能比 global 好非常多，由此可見 fully utilize share memory 的重要性，另外可以看到當 blocking factor 到 16 時，global memory store bandwidth 反而是下降的，推測是一次處理的資料量多到超過 memory 的寫入傳輸上限，因此反而造成下降。

- Optimization

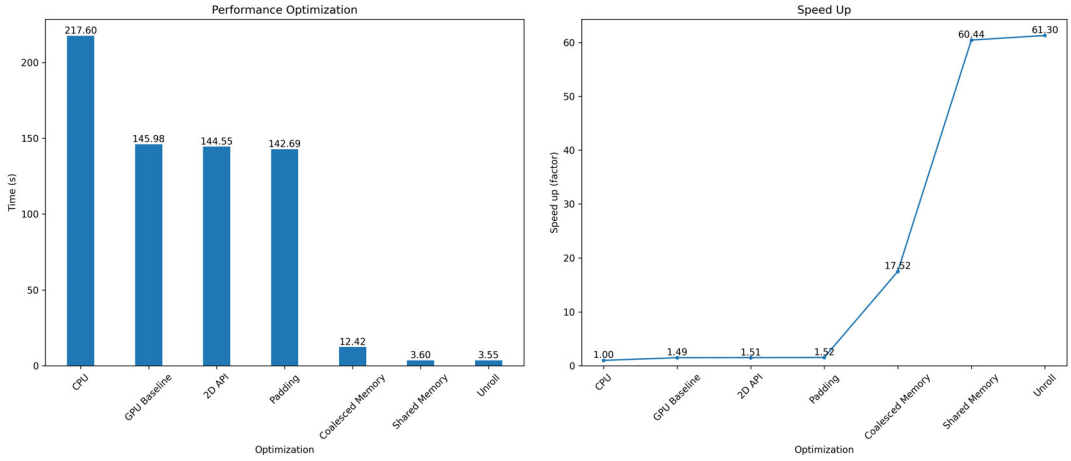
  以下數據同樣是以 p11k1 為測資所測量出來的，Vertex num: 11000、Edge num: 505586。

  - 測量方式：

    使用 **sys/time.h** libarary。

    ```
    struct timeval start, end;
    gettimeofday(&start, NULL);
    // ...
    gettimeofday(&end, NULL);
    ```

```
double time = (double)(US_PER_SEC*(end.tv_sec-start.tv_sec)+(end.tv_usec-start.tv_usec))/US_PER_SEC;
printf("%.2lf\n", time);
```



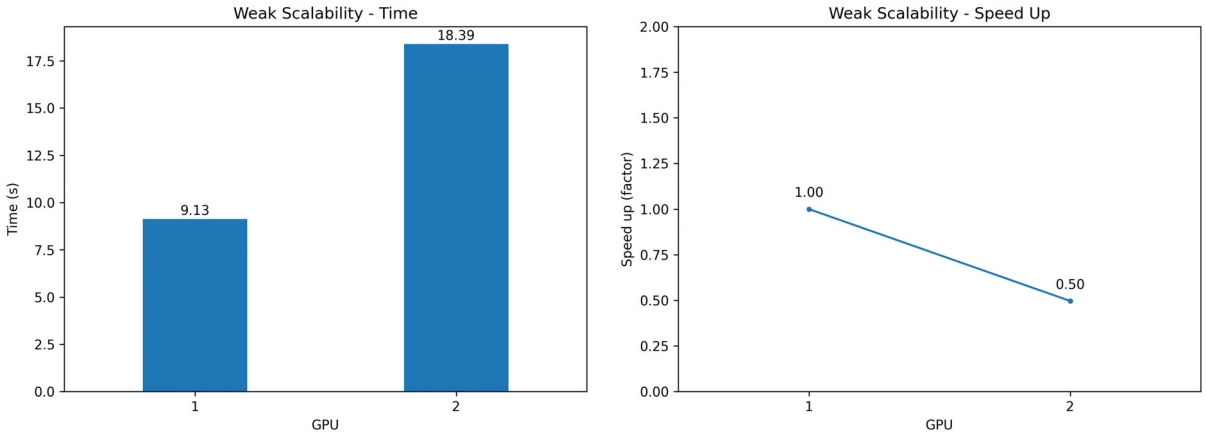
根據實驗發現除了 coalesced memory 以及 share memory 以外的優化基本上都只有加速一點點而已。Share memory 直接優化了 baseline 的版本超過 60 倍，由此也可以看出 fully utilize share memory 的重要性。Optimization 的流程已在上面實作的部分說明。

- Weak Scalability

  因為 weak scalability 是希望兩邊的運算量約等比例於計算資源量，而 Floyd-Warshall 的時間複雜度為 $O(V^3)$，因此我找到兩筆測資使得 $2 * V_1^3 \approx V_2^3$。

  使用 p15k1 以及 p19k1 兩筆測資來實驗 ($V_1 = 15000, V_2 = 18947, V_1^3 = 3.375e12, V_2^3 = 6.80e12 \approx 2 * V_1^3$)。

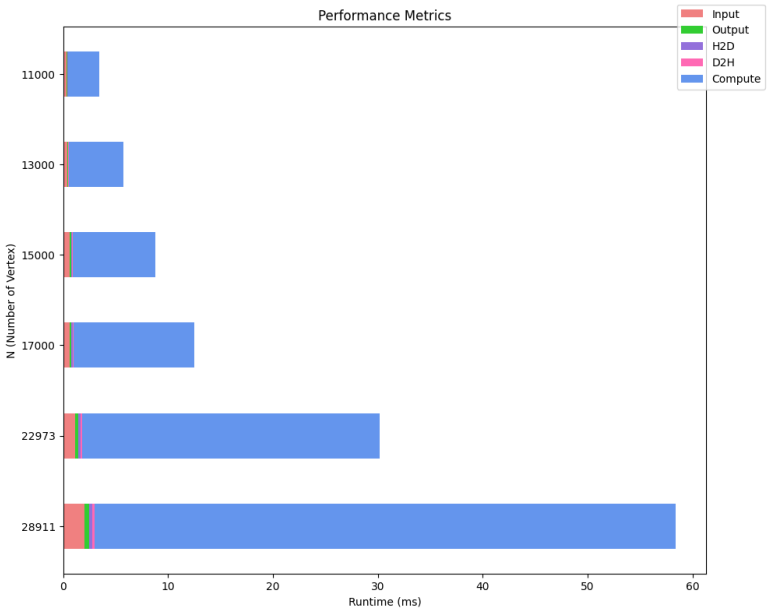
```
File: p15k1, Vertex Count: 15000, Edge Count: 5591272
File: p15k2, Vertex Count: 14788, Edge Count: 5325850
File: p15k3, Vertex Count: 14897, Edge Count: 1014029
File: p16k1, Vertex Count: 16000, Edge Count: 5444739
File: p16k2, Vertex Count: 15787, Edge Count: 5545845
File: p16k3, Vertex Count: 15973, Edge Count: 1647850
File: p17k1, Vertex Count: 17000, Edge Count: 4326829
File: p17k2, Vertex Count: 16979, Edge Count: 1643268
File: p17k3, Vertex Count: 16876, Edge Count: 7710574
File: p18k1, Vertex Count: 17728, Edge Count: 8565428
File: p18k2, Vertex Count: 17939, Edge Count: 2294421
File: p18k3, Vertex Count: 18000, Edge Count: 7166615
File: p19k1, Vertex Count: 18947, Edge Count: 333397
File: p19k2, Vertex Count: 18817, Edge Count: 8119933
File: p19k3, Vertex Count: 19000, Edge Count: 488342
```



由圖表可以看出 weak scalability 很糟糕，理想情況下兩者運算時間應該差不多，而 speedup 理想應為穩定的水平線；造成這個結果的原因最主要可能是因為沒有 fully utilize share memory，使得 computation 成為 bottle neck，其他可能的原因則是 I/O bottle neck 或是 communication overhead，不過在目前的實作中應該就是 computation 導致的。

- Time Distribution

Computation time 與 memory copy time(H2D/D2H) 利用 nvprof 的 summary mode 來計算，而 I/O 部分一樣用 **sys/time.h** 計算，另外，在我的實作中 communication time 跟 memory copy time 是一樣的。使用 p11k1, p13k1, p15k1, p17k1, p23k1, p29k1 六筆測資來實驗，vertex_num 依序為 11000, 13000, 15000, 17000, 22973, 28911。



由圖表可以看出 I/O time 與 Computation Time 都隨著 vertex_num 增加而變長，Memcpy time(H2D/D2H) 相較之下都微乎其微。 然而 I/O Time 是這次比較難優化的部分，因此主要要優化的是 Computation 的部分。

Experience & conclusion

What have you learned from this homework?

這次的作業花了許多時間才完成，除了因為不夠熟悉 cuda programming，也為了確認 blocked floyd warshall 的正確操作方式，花了不少時間 debug。最刺激的是在 deadline 前幾天學校的冷氣機房停電，導致 hades 停機，國網的資源也不夠所有人使用，一時之間只好跟同學借用他的 RTX 4070 來寫作業做實驗，明顯感受到算力又比 hades 的 GTX 1080 高很多，執行速度非常快。不過因為比較新的 GPU 只能用 ncu 進行 profile，使用的過程也遇到權限設定問題、一些環境設定的問題，試了一個晚上都沒有搞定，還好隔天中午 hades 就恢復運作了，於是就繼續使用 hades 來進行實驗。

這次 server 的突然停機把所有修課的朋友們都嚇了好大一跳，也因此做作業的時間變得更緊繃，不過看到一步一步地完成 spec 所要求的實驗項目，還是覺得很有趣並且很有成就感；而因為時間緊迫，這次可惜的部分是沒有實作出把 share memory 完全用滿，blocking factor 只到 32，如果再把這部分也優化，performance 應該可以提升很大一截(share memory 8kb → 48kb)。有空我也會再研究一下，如何把自己本地的環境設定好以便 profile，這對於我們實驗室做模型的訓練會是很大的幫助，算是停機帶來的意外收穫。