HW3 Report

資應所碩二 111065531 郭芳妤

Implementation

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

   採取最簡單的 Floyd-Warshall 演算法，並使用 openmp 對其中的 for loop 進行平行化處理：

```
#pragma omp parallel num_threads(ncpus) shared(GMatrix) private(i, j)
    {
        for(int k = 0; k < G.V; k++){
            #pragma omp for schedule(dynamic)
            for(i = 0; i < G.V; i++){
                if(GMatrix[i][k] != OUR_INF){
                    for(j = 0; j < G.V; j++){
                        if (GMatrix[i][j] > (GMatrix[i][k] + GMatrix[k][j]
                            GMatrix[i][j] = GMatrix[i][k] + GMatrix[k][j];
                        }
                    }
                }
            }
        }
    }
```

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

   `3-2`

   在實作 blocked-FW 演算法時，最直觀的想法是需要用滿 thread，所以以 32 為一個 block-size 切分資料，然而如果需要使用 shared memory 加速，我們需要考慮 shared memory 可以開到多大，我們使用 deviceQuery 觀察 shared memory size = 49152 byte，然後在 Blocked-FW 中最多可能開到 3 個 blocks（phase 3），所以 49152/3*12（sizeof(int)*3 blocks）= 64*64，這表示我們的 shared memory matrix 最大可以開到 64，考量到 shared memory 的 access 時間比global memory 快非常多，我們將 block-size 調整成 64。

```
#define Block_size 64
#define offset 32
```

   但考量到 thread 最多還是只能開 32*32，所以當 thread 和 block 內的資料無法一一對應時，我們就要變動 thread 的資料處理量（1個 thread 需要處理 4 格），所以我們跟改動 phase1 ~ phase3 的 kernel function：

   > Phase1

```
__global__ void FWP1(int* dev_dist, int round, int n_padding){
    int i = threadIdx.y;
    int j = threadIdx.x;

    int up_left = round*Block_size*(n_padding+1) + i*n_padding + j;
    int up_right = round*Block_size*(n_padding+1) + i*n_padding + j + offs
    int down_left = round*Block_size*(n_padding+1) + (i+offset)*n_padding
    int down_right = round*Block_size*(n_padding+1) + (i+offset)*n_padding

    int offset_i = i+offset;
    int offset_j = j+offset;

    __shared__ int sdata[Block_size][Block_size];

    sdata[i][j] = dev_dist[up_left];
    sdata[i][offset_j] = dev_dist[up_right];
    sdata[offset_i][j] = dev_dist[down_left];
    sdata[offset_i][offset_j] = dev_dist[down_right];

    __syncthreads();

    #pragma unroll 64
    for(int k = 0; k < Block_size; k++){
        __syncthreads();
        sdata[i][j] = min(sdata[i][j], sdata[i][k]+sdata[k][j]);
        sdata[i][offset_j] = min(sdata[i][offset_j], sdata[i][k]+sdata[k][
        sdata[offset_i][j] = min(sdata[offset_i][j], sdata[offset_i][k]+sd
        sdata[offset_i][offset_j] = min(sdata[offset_i][offset_j], sdata[o
    }

    dev_dist[up_left] = sdata[i][j];
    dev_dist[up_right] = sdata[i][offset_j];
    dev_dist[down_left] = sdata[offset_i][j];
    dev_dist[down_right] = sdata[offset_i][offset_j];
}
```

具體的分配如下圖所示：

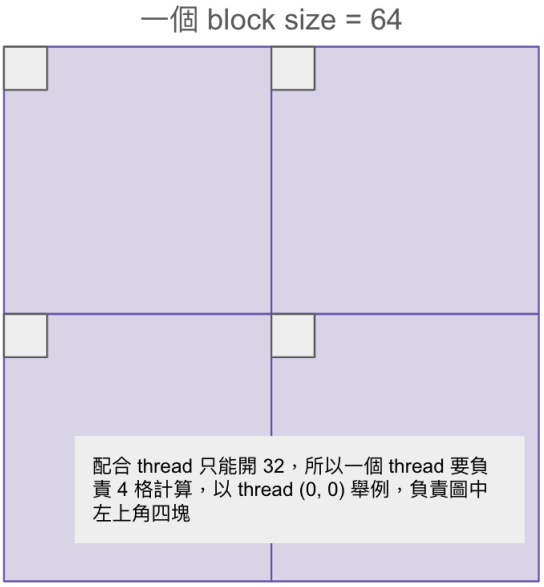


圖 1

而 phase 2 和 phase 3 也是以此類推，只是 function 中的 loop 變成：

```
#pragma unroll 64
for(int k = 0; k < Block_size; k++){
    __syncthreads();
    pivot_row[i][j] = min(pivot_row[i][j], pivot[i][k] + pivot_row[k][j]);
    pivot_row[i][offset_j] = min(pivot_row[i][offset_j], pivot[i][k] + piv
    pivot_row[offset_i][j] = min(pivot_row[offset_i][j], pivot[offset_i][k
    pivot_row[offset_i][offset_j] = min(pivot_row[offset_i][offset_j], piv

    pivot_col[i][j] = min(pivot_col[i][j], pivot_col[i][k] + pivot[k][j]);
    pivot_col[i][offset_j] = min(pivot_col[i][offset_j], pivot_col[i][k] +
    pivot_col[offset_i][j] = min(pivot_col[offset_i][j], pivot_col[offset_
    pivot_col[offset_i][offset_j] = min(pivot_col[offset_i][offset_j], piv

}
```

```
__syncthreads();
#pragma unroll 64
for(int k = 0; k < Block_size; k++){
    outcome[i][j] = min(outcome[i][j], pivot_col[i][k] + pivot_row[k][j]);
    outcome[i][offset_j] = min(outcome[i][offset_j], pivot_col[i][k] + piv
    outcome[offset_i][j] = min(outcome[offset_i][j],  pivot_col[offset_i][
    outcome[offset_i][offset_j] = min(outcome[offset_i][offset_j],  pivot_
}
```

3-3

而 3-3 的處理方方式是把 phase 3 分成上半和下半交由不同的 GPU 處理，在傳送資料時分為兩個 round 進行判斷，如果是前半個 round，就是由 GPU0 傳送一個 row（pivot 的下一個 row 給 GPU1）而後半個 round 則由 GPU1 傳送一個 row 給 GPU0。

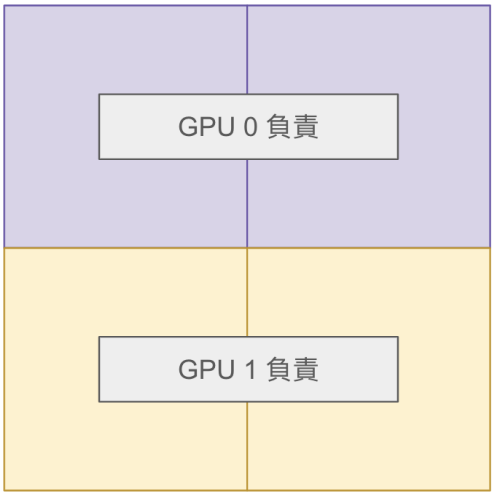
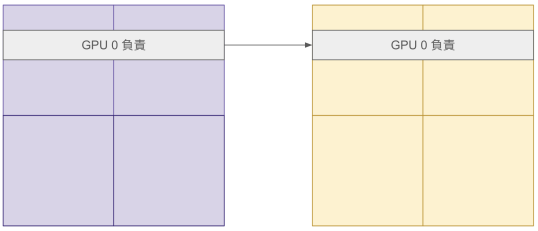


圖 2 - 將 phase 3 分為兩半進行處理



GPU0 傳送一個 row 給 GPU1

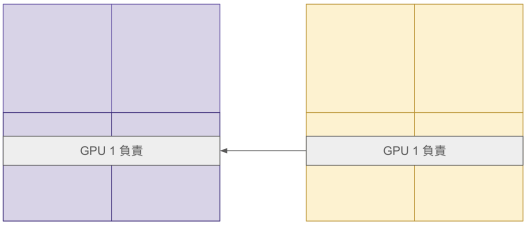


圖 3 - GPU1 傳送一個 row 給 GPU0

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

   `3-2 & 3-3 在參數設置上基本相同`：

   1. blocking factor = 64：為了用滿 share memory
   2. block 數量為 padding 後的 N / block_size(64)：在此 padding 是為了運算方便（不用再額外在 kernel function 內判斷是否超過原本的 node 數量而 break 掉）。是以 n = 78 舉例，padding 以後為 128，總共會有 2 個 block。
   3. thread 數量：為了用滿資源的狀態，我們將 thread 數量設置為 32*32 = 1024

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

   在 3-3 的實作中，我使用 peer-to-peer（P2P）API：

   1. 首先在一開始設置兩個 GPU 之間設定可以存取對方的 memory:

      ```
      int mainDevice = 0;
      int helperDevice = 1;
      cudaDeviceEnablePeerAccess(mainDevice, 0);
      cudaDeviceEnablePeerAccess(helperDevice, 0);
      ```

   2. 接著我們將最耗時的 phase 3 工作量切一半（phase1 和 phase2 都照算），分別分給 GPU-0 和 GPU-1，由 GPU0 負責上半、GPU1 負責下半。在傳送資料時，即使是傳送自己計算的那一半份量也會超時，所以只需要傳必要的 part，以確保上半、下半計算會是正確的。所以這邊傳送的方法為：先判斷在每一個 round 的 pivot 是由 0 或 1 計算，並把該正確的 pivot (包含 pivot row) 傳送給對方。

```
for(int r = 0; r < round; r++){
#pragma omp parallel private(pid)
    {
        pid = omp_get_thread_num();
        if (pid == 0){
            cudaSetDevice(0);
            FWP1<<<nB1, nT>>>(dev_dist, r, n_padding);
            FWP2<<<nB2, nT>>>(dev_dist, r, n_padding);
        }
        else{
            cudaSetDevice(1);
            FWP1<<<nB1, nT>>>(dev_dist1, r, n_padding);
            FWP2<<<nB2, nT>>>(dev_dist1, r, n_padding);
        }
        /* Phase 3 */
        #pragma omp barrier
        if(pid == 0){
            cudaSetDevice(0);
            FWP3<<<nB3_upper, nT>>>(dev_dist, r, n_padding, 0, div);
            if(r < div-1){
                for(int i = 0; i < Block_size; i++){
                    cudaMemcpyPeer(dev_dist1 + (((r+1)*Block_size + i)*
                }
            }
        }
        else{
            cudaSetDevice(1);
            FWP3<<<nB3_lower, nT>>>(dev_dist1, r, n_padding, div, round
            if(r >= div-1 && r < round-1){
                for(int i = 0; i < Block_size; i++){
                    cudaMemcpyPeer(dev_dist + (((r+1)*Block_size+i)*n_p
                }
            }
        }
        #pragma omp barrier
    }
    }
```

Profiling Results

取用 h3-2 的最後一筆測資（p30k1）作為 Profiling 指標

- node = 30000
- edge = 3907489

我們僅取用 biggest kernel function，即 phase 3 的平均指標結果（平均）：

指標	nvprof metric	avg
occupancy	achieved_occupancy	95%
sm_efficiency	sm_efficiency	99.97%
global load throughput	gld_throughput	18.510GB/s
global store throughput	gst_throughput	60.942GB/s
shared load throughput	shared_load_throughput	3059.0GB/s
shared store throughput	shared_store_throughtput	249.80GB

Experiment & Analysis

▼ System Spec

本次 hw3-1 實作使用課堂提供之 Apollo，hw3-2、hw3-3 使用課堂提供之 hades 與 NCHC。

▼ Blocking Factor

> 💡 **Blocking Factor**
>
> - 測資：`hw3-2/c21.1` (node = 5000)
> - 觀察與總結：
>
> 1. Global memory version：
>
> a. 可以看到在 blocking factor < 64 時，computation performance 表現相當，但當 blocking factor 在 128、256 表現則明顯較差
>
> b. bandwidth：blocking factor < 64 時 bandwidth 表現較佳，且 load 均 > store
>
> 2. 在 shared memory 版本中，我們雖然沒有實作 128、256 的版本，但在 computation performance 中表現較均較 global memory 的版本好（且 shared memory 版本的這三個 blocking factor 表現相當）；而 bandwidth 則以 shared memory 較 global memory 高。
>
> a. 從下方 shared memory 的 load/store 分開呈現的圖表（圖 5(d)）也可以看出主要 bandwidth 在 shared memory 的 load

使用 global memory 版本實作，結果如下：

- 我們使用 global memory 的版本實作以下 Blocking Factor（16, 32, 64, 128, 256）
- Integer GOPS = Integer Instruction * round / phase3 time / 10^9
- Bandwidth：看 load+store 以及 load、store 分開（只看 global）

Blocking Factor	Integer GOPS	BandWidth (global - store)	BandWidth (global - load)
16	626	434.64GB/s	461.80GB/s
`32`	657	470.24GB/s	602.50GB/s
64	503	371.50GB/s	835.88GB/s
128	82	61.735GB/s	111.18GB/s

Blocking Factor	Integer GOPS	BandWidth (global - store)	BandWidth (global - load)
256	81	60.589GB/s	136.33GB/s

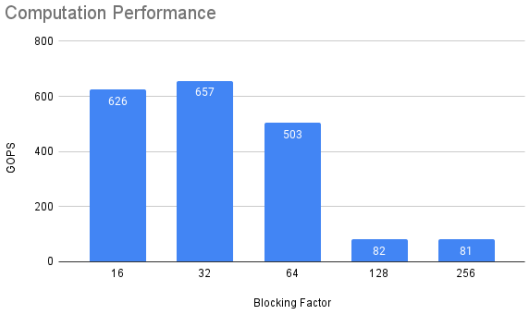


圖 4(a) - Computation

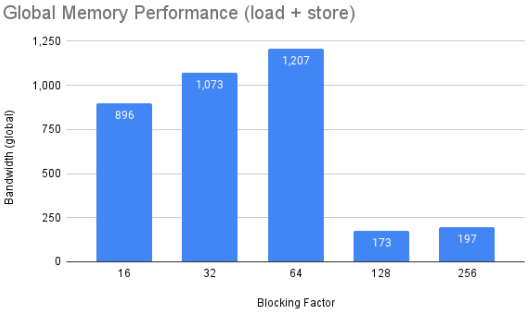


圖 4(b) - Bandwidth (縱軸單位：GB/s)

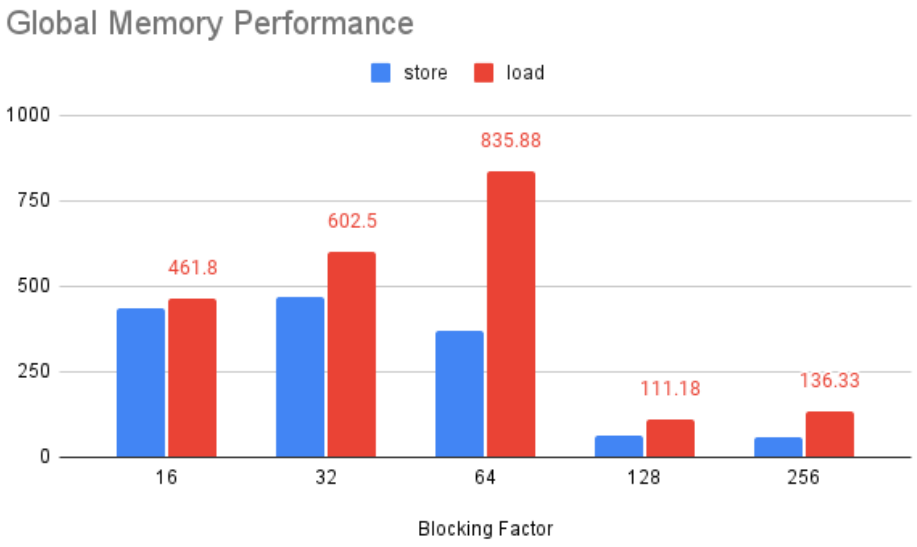


圖 4(c) - 將 load 與 store 分開比較（縱軸單位：GB/s）

使用 share memory 實作之版本：

- 由於我們優化的版本為 shared memory 版本，因此也附上 shared memory 版本中，blocking factor 為 16, 32, 64 時的表現

Blocking Factor	Integer GOPS	BandWidth (global - store)	BandWidth (global - load)	BandWidth (share - store)	BandWidth (share - load)
16	1303	94.987GB/s	284.96GB/s	379.95GB/	2374.7GB/s
32	1198	61.119GB/s	183.36GB/s	244.48GB/s	2994.8GB/s
64	1587	61.175GB/s	183.53GB/s	225.90GB/s	2997.6GB/s

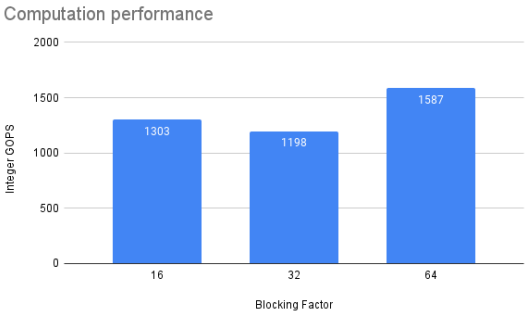


圖 5(a)

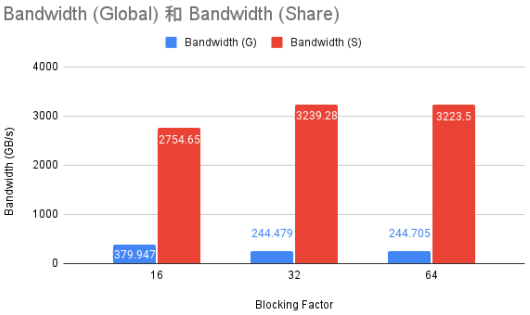


圖 5(b)

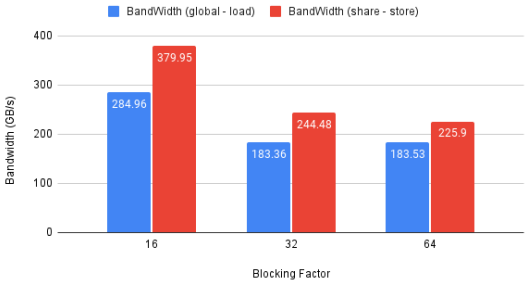


圖 5(c)

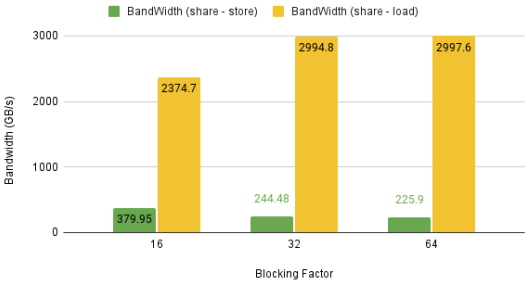


圖 5(d)

▼ Optimization

Optimization

- 測資：`hw3-2/p11k1`
- 觀察與總結：
  1. 可以看到在所有的優化版本中，第一個大幅進步是使用了 shared memory，再次印證上課時老師說的 global memory 和 shared memory 造成的差異（shared memory 在 chip 上）
  2. 第二個大幅進步是使用了 coalesced memory：當我們讓同個 warp 中的 thread access 連續的 memory，就可以進行加速

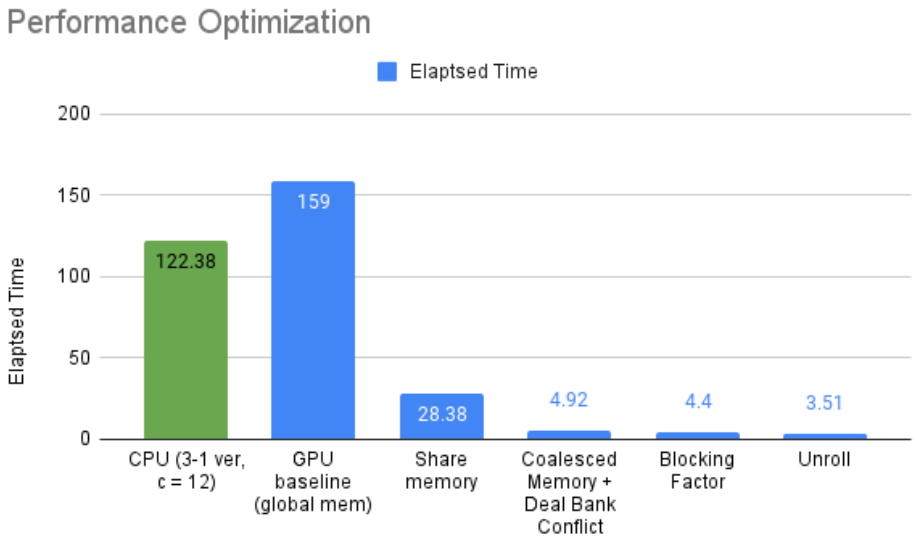


圖 6

1. `CPU version (serial)`：使用 3-2 提供的 sequential code 跑到超時，因此選用 3-1 CPU version 替代（上圖綠色柱狀圖）
   a. CPU version（3-1 code）：使用 12 個 thread，實作方法為一般 Floyd-Warshall。
2. `GPU baseline`：使用 global memory 實作，直接在 kernel function 中計算 index 並寫回 global memory（已用 padding，故沒有額外實作無 padding 版本），Blocking Factor = 32（依照 thread 開滿的大小設置）
3. `shared memory`：先把要計算的 block 資料搬到 shared memory（2D），方法為開一個 32*32 的 shared matrix 並在其中進行 k-for loop 計算，算完後在最後一步才寫回 global memory，目標是大幅降低 access global memory 的時間
4. `coalesced memory & bank conflict`：原本的方法為 使用 threadIdx.x 為 i、threadIdx.y 為 j 去 access shared memory，但這會導致同一個 warp 內的 thread 不會取到連續的記憶體，所以改為 threadIdx.y 為 i(row)、threadIdx.x 為 j(col)，讓連續的 thread 可以取到連續的 memory
5. `Blocking factor (32→64)`：由於 shared memory 最大其實可以允許我們開到 49152 byte（64*64*3*int），也就是最多我們可以開 64*64 的 matrix，所以把 Block-size 改為 64，並讓每一個 thread 做 4 格計算（詳見 implementation 描述）
6. `unroll`：在 Kernel function 中的 使用 `#pragma unroll 64`

▼ Weak Scalability (`3-3`)

> 💡 Weak Scalability
>
> - 小測資（C04.1）、中測資（p33k1）、大測資（C07.1）
>
> | | C04.1 | p33k1 | C07.1 |
> |---|---|---|---|
> | node | 5000 | 33000 | 44939 |
> | edge | 10723117 | 9027729 | 2418733 |
>
> - 結論：使用 2 顆 GPU 的 total time 約是 1 顆 GPU 的 1/2 倍

結果呈現：

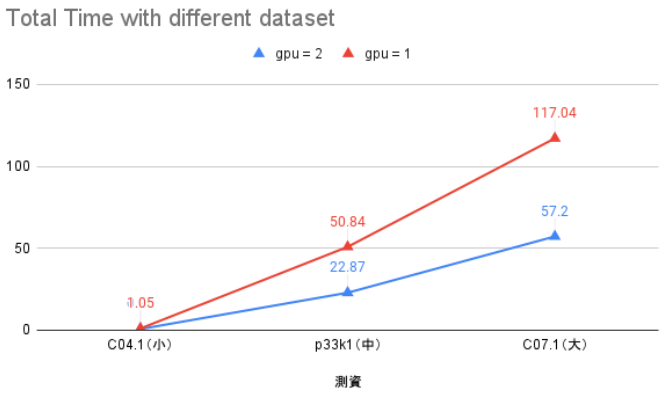


圖 7

測資	C04.1（小）	p33k1（中）	C07.1（大）
GPU(2)	0.72	22.87	57.20
GPU(1)	1.05	50.84	117.04

▼ Time Distribution (`3-2`)：

Time Distribution

- 在此挑選四筆不同 size 的測資進行 distribution 比較，其中四筆測資的基本資訊如下：

	p11k1	p15k1	p20k1	p30k1
node	11000	15000	20000	30000
edge	505586	5591272	264275	3907489

- 觀察與總結：
  1. 隨資料的 size 上升，total time 上升，且可以從左圖發現上升的主因為 computation part
  2. IO(r/w)：read 和 write 沒有差異，但可以發現 p15k1（有較多 edge 的資料）read 時間較其他測資長
  3. 比較 H2D 和 D2H：可以發現 D2H 的 memory copy 時間都比 H2D 長（落在 1-2 倍之間）

各時間分佈數據與圖表如下：

time part (ms)	p11k1	p15k1	p20k1	p30k1
computing (s)	1.25	3.23	7.6	25.36
IO (read) (s)	0.02	0.2	0.01	0.140
IO (write) (s)	0.02	0.02	0.01	0.146
H2D(ms)	79.1	151.1	161.1	602.08

time part (ms)	p11k1	p15k1	p20k1	p30k1
D2H(ms)	94.2	210.9	347.6	838.5
total	3.62	7.55	13.9	39.72

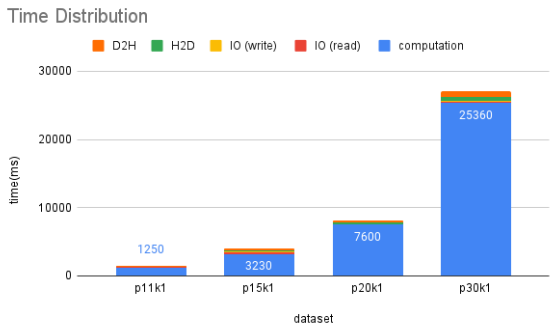


圖 8(a)

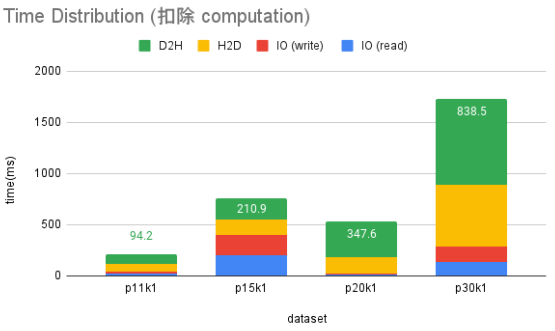


圖 8(b)

Experience & Conclusion

在本次作業中因為分別實作了 3-2 和 3-3 而對於 CUDA 有了更深刻的理解，尤其是每完成一個小 part 都是一次翻山越嶺（XD），而且難點各有不同：3-2 最困難的部分是在眾多優化中要想到「用滿 shared memory」（不然過不了大測資），3-3 更困難的是要想到「並不是需要傳所有的 row 才會正確」，只能說即使老師上課說平行的演算法相較之下很單純，但要在有限的時間內想到、並且驗證其正確性還是有一定難度。另外一個感想是： GPU 有關的 debug 真的很困難，而且當 GPU 運作的方式和想像中大不相同的時候就很難立即發現錯誤。

▼ References（感謝網路上眾多的資源）

- https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Asmita-Gautam-Spring-2019.pdf
- https://hackmd.io/@zing9264/ryqoS6ncY