CS542200 Parallel Programming Homework 3: All-Pairs Shortest Path

Bo-Wei Lee (李柏葳)

111065525

1 程式建置

1.1 使用演算法

這一次的作業是 All-Pairs Shortest Path，其目的爲給予一個 graph 的所有邊的起始點、終點和權重。找出所有點對於其他點的所有最短距離。我選用的是實作 Blocked Floyd-Warshall 演算法來解決這次問題。跟原始的 Floyd-Warshall 相比，雖然在複雜度上都是 O(n3)，但是前者在平行化上佔據很大的優勢。這個優勢主要是來自能更好的利用到 thread cache，以及能更有效的去做 load balancing。順帶一提，原本我也有考慮使用複雜度較低的 Johnson algorithm，但因爲其中 priority queue 在實作上很難平行化的關係，後來就不使用。

1.2 資料分配

Blocked Floyd-Warshall 分爲三個階段，這三個階段會依序執行，總共會執行 V/B 個 round，每一個階段都會選取特定的 block，去做 block 內部小的 Floyd-Warshall 計算。首先可以觀察到對於每個 round，其資料是依賴上一輪的結果，所以很難做平行化。第二是對於三個 stages，第三個 stage 會依賴第二個 stage，第二個 stage 會依賴第一個 stage，也很難去做平行化。第三，在每個 round 裡面，會有不同的 block 被選定去做 APSP，這些小的 block 之間的運算是完全獨立的，而這些 block 如何分配到不同的 thread 上就是重點。
在這邊，我將整理好的相鄰矩陣切成 m * m 個等長的正方形，每個正方形的長爲 n，可根據處理器去做優化。Stage 1 總共會運算 1 個 block，stage 2 扣除掉重複計算的部分，會運算 2 * (m - 1) 個，stage 3 會運算 (m - 1) * (m - 1) 個。在 hw3-2 中，每個 kernel 的 block dimension 會正好對應到需要處理的正方形數量，而且根據 blockIdx.x 與 blockIdx.y，kernel 可以從 global memory 撈到其計算需要用到的 block。程式碼如 Listing 2 所示。
在 hw3-3 中，我將相鄰矩陣分爲上下兩部分，m 爲偶數的話爲均分，奇數的話 GPU 0 會被多分到一個 row 的 block 去做計算。

1.3 平程超參數決定

1. Thread number: 在我的實作中，如果 n <= 32，thread 的 dim3 即爲 (n, n, 1)。n > 32 的話，因爲硬體限制，thread 的 dim3 爲 (32, 32, 1)。取而代之的是每一個 thread 要處理 (n / 32) * (n / 32) 筆資料。

2. Block number: 在 hw3-2 中，stage 1 kernel 的 dim3 爲 (1, 1, 1)，stage 2 的爲 (m - 1, 2)，stage 3 的爲 (m - 1, m - 1)。剛好對應到要處理的 block 數量。hw3-3 的話，stage 1 跟 stage 2 的 dim3 跟 hw3-2 相同，stage 3 的則爲 (m - 1, m / 2 + (m % 2))。

3. Block size: 在這裡我設定 block size 爲 64，設爲 64 相比於 32 會有幾個好處。第一是在撈取 global memory 的資料時，可以一次撈取 32 KB 的資料，更有效利用 shared memory，第二是讓 kernel 被 call 到的數量減半，每一次 kerenl call 會需要十幾毫秒的 overhead，當被叫上千次時，累積的秒數會相當可觀。第三是在 stage 3 中，需要執行 block 的數量減少四倍，而且由於其資料不依賴其他 thread 的結果，換算下來減少了四倍 syncthread 的次數。

1.4 程式碼實作簡介

1.4.1 Hw3-1 CPU version

在 CPU 版本，我選用的是 openMP 去做平行化處理，每個 thread 會根據 openMP 的調度去計算不同 block 的 APSP。以 stage 3 爲例，最外面的兩個迴圈代表目前要處理哪個 block，因爲互相獨立，所以使用 collapse(2) 來增加 load balancing 彈性。注意到因爲相鄰矩陣是放在 global memory 裡的，所以沒有 thread 溝通間的問題。另外，相鄰矩陣會被 padding 到 block size 的倍數，並且以一維矩陣表示。以避免使用 if else 來判斷邊界，使得效能降低。尤其是在 GPU 上會造成 branching，使得 warp 的使用率下降。在 block size 選取方面，我最後選定的是 85，三個 block 的大小和剛好比 cache size 還小一點，根據測試 block size 爲 85 的效能比 128 在 hw3-1-judge 還好上快一倍 (41s -> 20s)。

```
#pragma omp for schedule(dynamic) collapse(2)
for (int x_start = 0; x_start < n; x_start += BLOCK_SIZE) {
    for (int y_start = 0; y_start < n; y_start += BLOCK_SIZE) {
        if (x_start == k_start || y_start == k_start)
            continue;

        int x_end = x_start + BLOCK_SIZE;
        int y_end = y_start + BLOCK_SIZE;
        int k_end = k_start + BLOCK_SIZE;
        for (int k = k_start; k < k_end; k++) {
            for (int y = y_start; y < y_end; y++) {
                for (int x = x_start; x < x_end; x++) {
                    adjMat[y * n + x] = std::min(adjMat[y * n + x],
                        adjMat[y * n + k] + adjMat[k * n + x]);
                }
            }
        }
    }
}
```

Listing 1: Hw3-1 stage 3 code

1.4.2 Hw3-2 Single GPU version

在 GPU 版本中，總共有三個 kernel，分別代表三個 stage。每一個 kernel 處理的大小爲 64 * 64，亦即每個 thread 會處理四筆資料。三個 kernel 的實作方法都差不多，只是需要從 global

memory 拿的資料不太一樣。以 stage 3 爲例，首先程式碼會根據 blockIdx.x 跟 blockIdx.y 來計算需要存取的三個 block，其中自身會被放到 register 中，兩個依賴的矩陣會被放到 shared memory。全部的 thread 都存好後，會進入 apsp 的環節，每個 thread 會對自己需要處理的資料做 relaxation，共處理 k 個 iteration。因爲其資料不依賴其他 thread 的結果，所以 stage 3 的迴圈內部不需要同步化。計算完後再將結果傳回至 global memory 中。詳細的程式碼如 Listing 3 所示，因爲版面關係所以放上的是 block size 爲 32 的程式碼。

```
    for (int g_k = 0; g_k < n; g_k += BLK_n) {
        stage1<<< 1, dim3(THREAD_n, THREAD_n), 0 >>> (devMat, g_k, n);
        stage2<<< dim3(block_dim - 1, 2), dim3(THREAD_n, THREAD_n), 0 >>>
            (devMat, g_k, n);
        stage3<<< dim3(block_dim - 1, block_dim - 1), dim3(THREAD_n,
            THREAD_n), 0 >>> (devMat, g_k, n);
    }
```

Listing 2: Hw3-2 call kernel function

```
__global__ void stage3(int *devMat, int g_k, int n) {
    // Load adj. matrix from global memory to shared memory
    int s_x = threadIdx.x;
    int s_y = threadIdx.y;
    int g_x = (blockIdx.x + (blockIdx.x >= (g_k >> log2BLK_n))) * BLK_n + s_x;
    int g_y = (blockIdx.y + (blockIdx.y >= (g_k >> log2BLK_n))) * BLK_n + s_y;

    __shared__ int mat[2 * BLK_n * BLK_n];

    // Load adj. matrixs from global memory to shared memory
    int num1 = devMat[g_y * n + g_x];
    mat[s_y * BLK_n + s_x] = devMat[g_y * n + (g_k + s_x)];
    mat[s_y * BLK_n + s_x + BLK_n * BLK_n] = devMat[(g_k + s_y) * n + g_x];

    __syncthreads();

    // Perform APSP on the block
    for (int k = 0; k < BLK_n; k++) {
        num1 = min(num1, mat[s_y * BLK_n + k] + mat[k * BLK_n + s_x + BLK_n *
            BLK_n]);
    }

    // Write data back to global memory
    devMat[g_y * n + g_x] = num1;
}
```

Listing 3: Hw3-2 stage 3 code

1.4.3 Hw3-3 Multiple GPU version && GPU 間的資料溝通

在作業 3-3 中，因爲 GPU 之間傳輸的 cost 很大，所以我只有在 stage 3 的時候做平行化，其他兩個 stage 的計算是獨立運行的。在 stage 前的開頭時，程式會將 stage3 會用到的 block row 利用 cudMemcpyPeer 傳到另一張 GPU，column 的話因爲兩張 GPU 都有各自做運算，所以就不需要傳。

```
#pragma omp barrier
    cudaMemcpy(devMat[id] + (start_y * n), &adjMat[start_y * n], block_n *
        BLK_n * n * sizeof(int), cudaMemcpyHostToDevice);

    for (int g_k = 0; g_k < n; g_k += BLK_n) {
        int copy = (g_k >= start_y && g_k < (start_y + block_n * BLK_n));
        cudaMemcpyPeer(devMat[!id] + (g_k * n), !id, devMat[id] + (g_k * n),
            id, copy * BLK_n * n * sizeof(int));
#pragma omp barrier
        stage1<<< 1, dim3(THREAD_n, THREAD_n), 0 >>> (devMat[id], g_k, n);
        stage2<<< dim3(block_dim - 1, 2), dim3(THREAD_n, THREAD_n), 0 >>>
            (devMat[id], g_k, n);
        stage3<<< dim3(block_dim - 1, block_n), dim3(THREAD_n, THREAD_n), 0
            >>> (devMat[id], g_k, n, start_y);
    }

    cudaMemcpy(&adjMat[start_y * n], devMat[id] + (start_y * n), block_n *
        BLK_n * n * sizeof(int), cudaMemcpyDeviceToHost);
```

Listing 4: Hw3-3 show communication between GPU

2 Profiling 結果

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "NVIDIA GeForce GTX 1080 (0)"					
Kernel: stage1(int*, int, int)					
172	achieved_occupancy	Achieved Occupancy	0.497380	0.497547	0.497475
172	shared_load_throughput	Shared Memory Load Throughput	110.43GB/s	127.54GB/s	125.82GB/s
172	shared_store_throughput	Shared Memory Store Throughput	37.239GB/s	43.009GB/s	42.430GB/s
172	gld_throughput	Global Load Throughput	586.66MB/s	677.55MB/s	668.44MB/s
172	gst_throughput	Global Store Throughput	586.66MB/s	677.55MB/s	668.44MB/s
Kernel: stage2(int*, int, int)					
172	achieved_occupancy	Achieved Occupancy	0.968005	0.989725	0.978118
172	shared_load_throughput	Shared Memory Load Throughput	2460.6GB/s	2826.9GB/s	2799.4GB/s
172	shared_store_throughput	Shared Memory Store Throughput	842.56GB/s	967.96GB/s	958.55GB/s
172	gld_throughput	Global Load Throughput	25.532GB/s	29.332GB/s	29.047GB/s
172	gst_throughput	Global Store Throughput	12.766GB/s	14.666GB/s	14.523GB/s
Kernel: stage3(int*, int, int)					
172	achieved_occupancy	Achieved Occupancy	0.916643	0.919395	0.917832
172	shared_load_throughput	Shared Memory Load Throughput	3294.5GB/s	3380.4GB/s	3352.1GB/s
172	shared_store_throughput	Shared Memory Store Throughput	137.27GB/s	140.85GB/s	139.67GB/s
172	gld_throughput	Global Load Throughput	205.91GB/s	211.27GB/s	209.50GB/s
172	gst_throughput	Global Store Throughput	68.636GB/s	70.424GB/s	69.835GB/s

3 實驗

3.1 實驗用資料與裝置描述

1. 測試用的機器爲課程所提供的 Hades 伺服器
2. 伺服器提供 GTX1080 顯卡
3. 在以下實驗中，除了 weak-scaling 外我選用的是公開資料集裡的 p11k1 做爲輸入測資，測資的節點數爲 11000，邊數爲 505586，會選用這筆資料是因爲其數據量夠大，可以凸顯平行程度帶來的優勢，但又不會跑太久。

4. 計時方式使用 clock_gettime(CLOCK_MONOTONIC, &time); 或是 nvprof。

3.2 Blocking factor

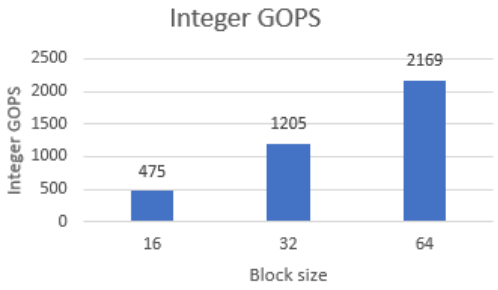


Figure 1: Integer GOPS vs Block size

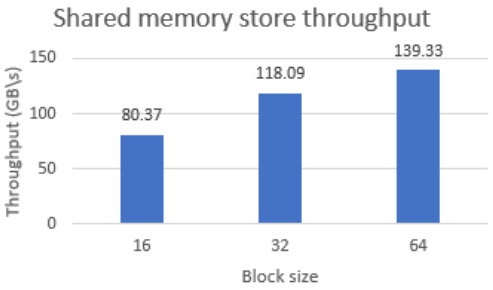


Figure 2: Shared memory store throughput vs Block size

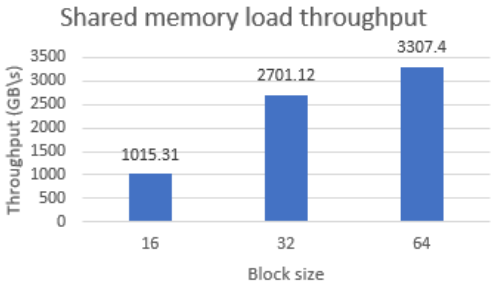


Figure 3: Shared memory load throughput vs Block size

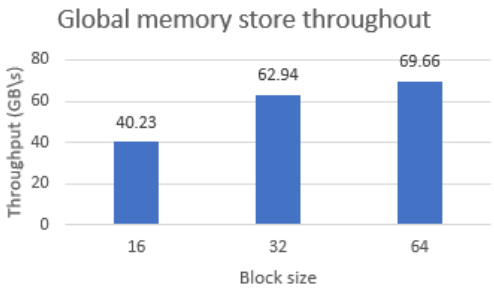


Figure 4: Global memory store throughput vs Block size

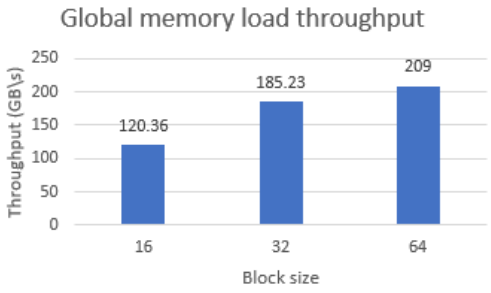


Figure 5: Global memory load throughput vs Block size

3.3 Optimization

看 Figure 6。

3.4 Weak scalability

使用的測試數據在單 GPU 版本爲 p27k1，雙 GPU 版本爲 p34k1，因爲 p27k1 的節點數爲 27000，p34k1 的節點數爲 34000。2 * 27000 * 27000 * 27000 約等於 34000 * 34000 * 34000。

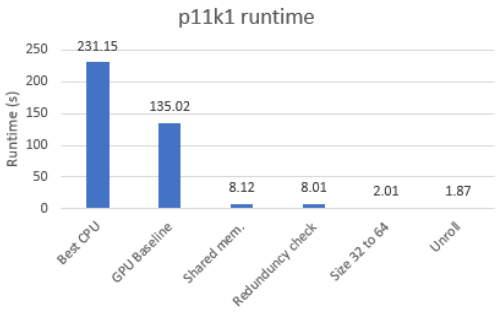


Figure 6: Baseline and improvements run-time

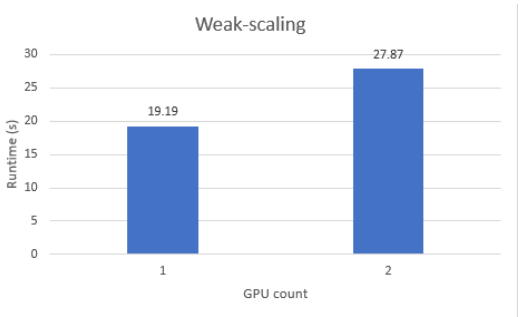
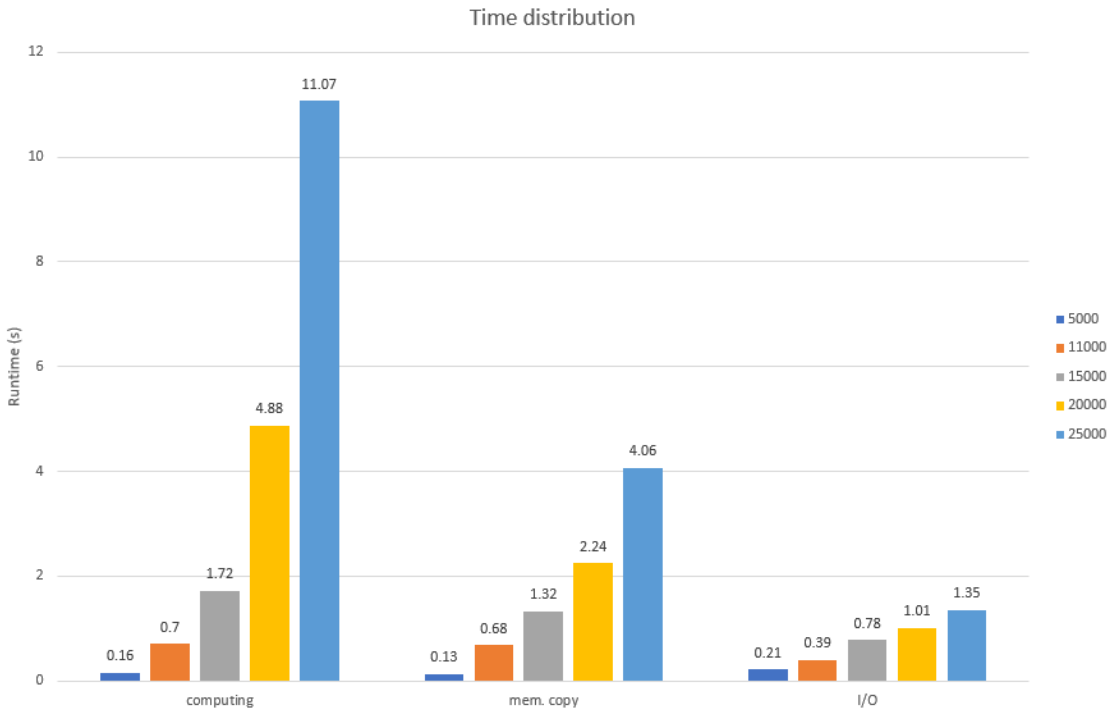


Figure 7: Weak-scaling test on multi GPU

3.5 Time Distribution



4 結論

這一次的作業最有挑戰的地方是要對於 GPU 的架構熟悉，寫出來的效能才會好，除蟲的時候也會比較順。這是我第一次在寫程式的時候，還要特別的花很多時間去找尋硬體本身相關的文件，是蠻有趣的體驗。