

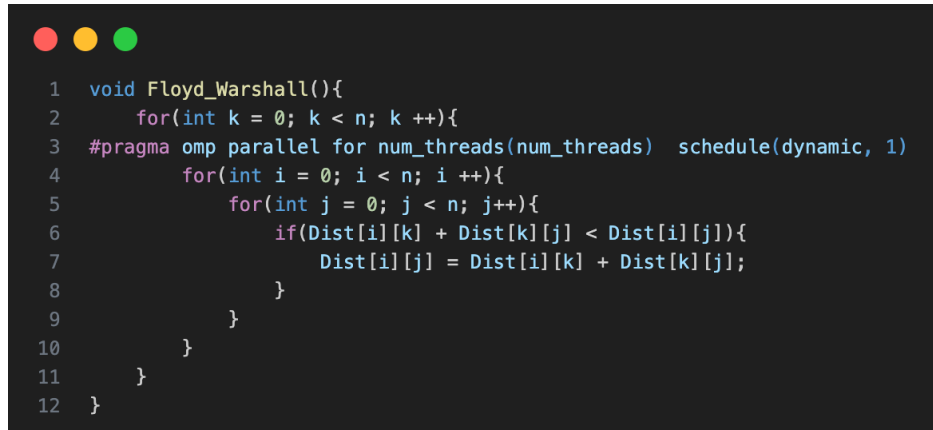
Parallel Programming Hw3

111062503 資工系碩二 吳冠志

1. Implementation

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

本次 hw3-1 使用基本的 Floyd-Warshall algorithm，並透過 OpenMP 進行平行化加速



```
1 void Floyd_Warshall(){
2     for(int k = 0; k < n; k++){
3         #pragma omp parallel for num_threads(num_threads) schedule(dynamic, 1)
4         for(int i = 0; i < n; i++){
5             for(int j = 0; j < n; j++){
6                 if(Dist[i][k] + Dist[k][j] < Dist[i][j]){
7                     Dist[i][j] = Dist[i][k] + Dist[k][j];
8                 }
9             }
10        }
11    }
12 }
```

Fig. 1 hw3-1 平行化演算法

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

在 hw3-2、hw3-3 中，每個 block 可以有 $32 * 32$ 共 1024 個 threads，為了將 block 的 shared memory 塞滿以最大化利用 shared memory(shared memory 共 48 KB 可塞 $64 * 64 * 3$ 個 int)，因此我們原 matrix padding 到 64 的倍數，並將 block size 設為 $64 * 64$ ，以最大化 Blocked Floyd-Warshall algorithm 計算效能。

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

blocking factor 的設定如上題，為了最大化 Shared Memory 使用率使用 $64 * 64$ 的 block。block 數則根據輸入大小判定，為 $\text{ceil}(N/64)^2$ ，其中 N 為 input 節點數。Threads 根據硬體設定每個 block 開 $32 * 32$ 個 thread，負責處理 $64 * 64$ 的 block，詳細演算法於題 e 說明。

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

hw3-3 演算法與 hw3-2 相同，皆為 Blocked Floyd-Warshall algorithm 的實作，所需要新增的部分為跨 GPU 的分工及溝通。透過 OpenMP 開啟兩個 thread，分別操控不同 GPU，讓 GPU 0 負責 Blocked Floyd-Warshall 前半部的計算，GPU 1 負責 Blocked Floyd-Warshall 後半部的計算，在不同 Round 計算前，必須確保每顆 GPU 都擁有此 Round pivot row 的正確資訊，因此要判斷此 row 由哪顆 GPU 負責，並將其傳給另一顆 GPU。

```

1   for(int r = 0; r < round ; r++){
2       if(r >= start_block && r < (start_block + job_size)){
3           pivot_offset = r * block_size * N;
4           cudaMemcpy(deviceDist[nei_cpu_thread_id] + pivot_offset,
5                       deviceDist[cpu_thread_id] + pivot_offset, length_per_row_byte, cudaMemcpyDefault);
6       }
7       #pragma omp barrier
8       phase1<<<1, block>>>(r, deviceDist[cpu_thread_id], N);
9       phase2<<<round - 1, block>>>(r, deviceDist[cpu_thread_id], N);
10      phase3<<<block3, block>>>(r, deviceDist[cpu_thread_id], N, start_block);
11  }

```

Fig. 2 hw3-3 兩顆 GPU 分工及溝程式碼

因此，在 Blocked Floyd-Warshall 每一 Round 中，總傳輸量為一個 Block Row 的資料量，並讓兩顆 GPU 分別負責計算上下半部一半的資料量（如果總 Block Row 為奇數，GPU 1 會多負責 1 Row 計算）。

- e. Briefly describe your implementations in diagrams, figures or sentences. Blocked Floyd-Warshall algorithm 就是將 Floyd-Warshall 根據 block size 進行切分，並在每一 round 的 block 執行 3 個階段更新整個 matrix，最後得到一個更新完成的 All pair shortest path matrix。不同 phase 流程已於 spec 有詳細說明，這邊就不細提，主要說明不同 phase 如何放入 shared memory 做運算。假設我們做到第 r round。

Phase 1：我們會開 $32 * 32$ 的 thread 供 phase1 做運算，每一個 thread(i, j)需要搬運 $\text{pivot}[i][j]$, $\text{pivot}[i][j+32]$, $\text{pivot}[i+32][j]$, $\text{pivot}[i+32][j+32]$ 進入 shared memory 中，其中 pivot 為原 matrix 中第 (r, r) 個 block，在確認各 thread 都有將此 block 中對應資料搬入 shared memory 中後依照 Blocked Floyd-Warshall algorithm phase1 進行運算。更新完成後存回原 dist。

Phase2: 此 phase 需要更新與 pivot block 同行及同列之 block，令其為 Ver block 以及 Hor block。

我們會開 $32 * 32$ 的 thread 給每個 block 並開 $\text{round} - 1$ 的 block，每一個 block 需要處理一個 Ver 以及一個 Hor 的 block，需要同時處理兩個 block 因此一個 thread 需要負責 8 個點的計算，同時在計算前後需要把 pivot block、Ver block 以及 Hor block 搬入 shared memory 中以及從 shared memory 搬回原 dist 中，原理與 phase1 相同。

Phase3: 此 phase 需要更新剩餘 block 距離，因此同樣對於每個 block 開啟 $32 * 32$ 個 thread，同時開啟 $(\text{round} - 1, \text{round} - 1)$ 個 block，可以透過其 ID 得知其需要計算的 block，令其為 my_block。此 block 中各 thread 是要負責 my_block 中的四個點的運算，並在計算前後將 my_block、其對應的 Ver block、Hor block 搬入 Shared Memory 以及從 Shared Memory 中將 my_block 更新完的值搬回原 dist 中。

在完成 3 個 phase 的運算後，此 Round 計算完成，完成所有 Round 的運算及完成此次 Blocked Floyd-Warshall 的運算。

在進行 Blocked Floyd-Warshall 運算前，我們會先將此 matrix 進行 padding，將 matrix 擴充為 block size 的倍數（此次實作 block size 為 64），其目的是為了避免多餘的判斷造成效能損耗，讓 block 中各 threa 都有對應的計算點雖然會造成部分無用的計算但可以減少整體程式判斷增加效能。

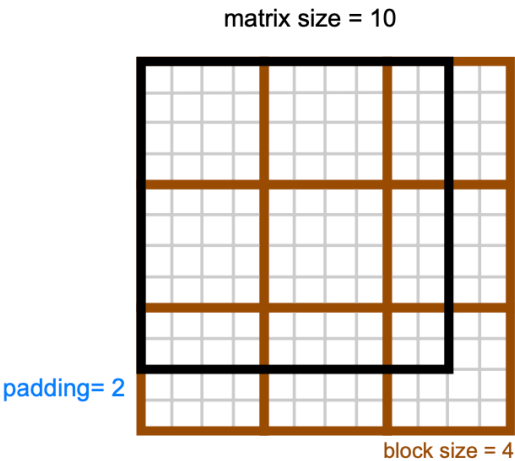


Fig. 3 Padding 範例展示，從黑色 matrix 擴充成咖啡色 matrix 減少判斷

2. Profiling Results (hw3-2)

本次使用 hades server 中/home/pp23/share/hw3-2/cases/p21k3 作為測試資料，其共有 20832 個 nodes，經過 padding 共有 20864 個 nodes，透過 NVIDIA profiling tools 量測本次作業 hw3-2 biggest kernel(phase3)效能

● Occupancy

透過此指令分析可得知 GPU 中 warp 的平均活躍數相較於最大 warp 數的比例，透過--metric achieved_occupancy 可得到

	min	Max	Avg
Occupancy	0.833790	0.872720	0.852527

Fig. 4 Occupancy 展示

上表可看出在不同時期約有 85%的 warp 是活躍的，

● sm efficiency

透過此指令分析可得知此特定 GPU 至少有一個 warp 處於活躍狀態的時間百分比，透過--metric sm_efficiency 可得到

	min	Max	Avg
sm efficiency	71.33%	74.54%	72.49%

Fig. 5 sm efficiency 展示

- shared memory load/store throughput
shared_load_throughput 和 shared_store_throughput 分別代表在 shared memory 讀取和寫入的速度，較高的數值代表此 GPU 擁有良好的性能

	min	Max	Avg
shared memory load throughput	2130.1GB/s	2189.4GB/s	2169.0GB/s

	min	Max	Avg
shared memory store throughput	173.89GB/s	178.73GB/s	177.06GB/s

Fig. 6 Shared Memory Throughput 展示

- global load/store throughput
這兩個數值分別代表 GPU 在 global memory 的讀取和寫入速度，雖然遠慢於 shared memory 但我們仍然希望此數值越高越好

	min	Max	Avg
global memory load throughput	147.31GB/s	152.33GB/s	150.69GB/s

	min	Max	Avg
global memory store throughput	49.103GB/s	50.778GB/s	50.230GB/s

Fig. 7 Global Memory Throughput 展示

3. Experiment & Analysis

a. System Spec

本次作業 hw3-1 在課程提供的 apollo server 上進行，hw3-2、hw3-3 則是在 hades 上進行

b. Blocking Factor (hw3-2)

本次使用/home/pp23/share/hw3-2/cases 中 c20.1 資料，其共有 5000 個 Node，測量不同 block size 設置下的 Integer Gops 以及 Bandwidth Integer Gops 透過 nvprof 的 inst_integer metric 計算 kernel 的 integer 指令數，並同時透過 nvprof 觀察各 phase 時間計算最後的 integer Gops

Block size	16	32	64	128	256
Kernel Time(ms)	3.4402	6.3263	15.131	62.319	269.93
Int Operation	2242805760	4286251008	5728499712	9265591296	2.1110e+10
Integer GOPs	651.94	677.529	378.594	148.68	78.205

Fig. 8 多個 Blocking Factor Integer GOPs 比較

我原先預期不同 block size 的 Integer GOPs 應該要差不多，理由是大的 block size 每個 thread 需要處理比較多的點，但其複雜度不變，處理四個點和將處理一個點的時間放大四倍應該相差不遠，但結果是隨著時間變大，Integer GOPs 會遞減，而且相當明顯，推測可能原因如下
指令未攤平：因為隨著 block size 增加，每個 thread 需要做的點會變多，block size 為 256 時一個 thread 需要做 64 個點，不使用 for loop 需要撰寫許多程式碼，因此本次需要 thread 負責多點計算的實驗都使用 for loop 去計算 offset 考慮所有計算的點，但代價是一次 for loop 就需要一次 branch，此行為可能造成額外的效能負擔。

接著我們透過 nvprof 的 gld_throughput、gst_throughput 對量測不同 block size 在運算時的 bandwidth

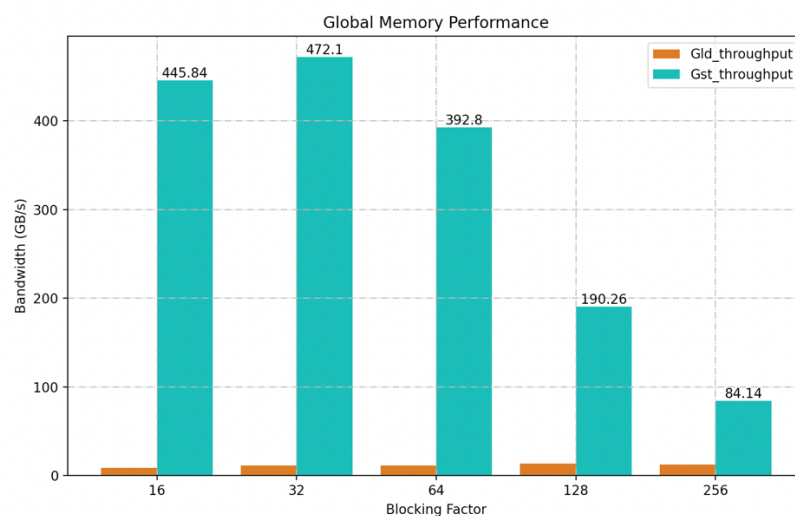


Fig. 9 多個 Blocking Factor gld、gst throughput 比較

從上圖可以看到，Global Memory 寫入 bandwidth 比讀取高出許多，推測是 Memory Coalescing 使得記憶體可以有效地被合併成一個大的區塊，讓寫入可以最大化的利用存取通道，相較讀取獲得的優化效果較明顯。且其隨著 Blocking Factor 增加有減少的趨勢，推測原因與前者分析類似，因為大 Blocking Size 的寫入無法平行化而且透過 for-loop 來完成寫入動作造成額外效能損耗。

c. Optimization (hw3-2)

本次預期比較以下優化技術的實驗成果

- CPU
- GPU baseline
- Padding
- Coalesced Memory
- Share Memory
- Large blocking factor(32 -> 64)
- Unroll

使用/home/pp23/share/hw3-2/cases/p11k2 作為本次測資，共有 10939 個 Nodes。

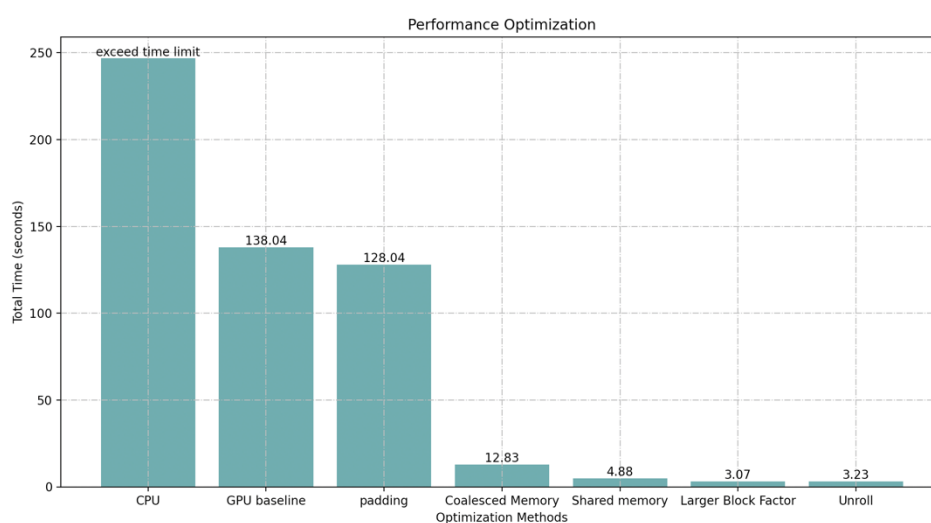


Fig. 10 Performance Optimization 展示

從上圖可以看出優化效果最明顯的是加入 **Coalesced Memory** 的優化，在此問題中其同時解決了 **Bank Conflict** 問題，因此與沒加入此優化技術的設置差異很大，加入 **Shared Memory** 也效果明顯，且可預期隨著資料越來越多使用 **Shared Memory** 帶來的優化效果會越顯著。最後加入 **Unroll** 並沒有帶來效能提升，我在測試其他資料時也有類似的結論，在最後的 judge 中並沒有透過 **Unroll** 來進行優化。

d. Weak scalability (hw3-3)

在進行 hw3-3 weak scalability 分析之前，我先分析 hw3-3 在不同 scale 資料下每個 row 的 Computing Time 以及 Communication Time

	C03.1	C05.1	p31k1
Number of Nodes	999	11000	31000
Computing Time	0.100038	3.52391	26.347919
Communication Time	0.069252	0.44783	1.206303
Comp/ Comm Ratio	1.44455	7.86885	21.84187

Fig. 11 跨 GPU Computing/ Communication Time 分析

可看到在不同 scale 下，GPU 主要皆花時間在處理矩陣的更新，雖然 Communication 會造成部分的 overhead，但隨著資料大小的增加，Computing Time/ Communication Time Ratio 越來越大，可預期 Single GPU 到 Multi GPU 的 Weak Scalability 應該會不錯。

本次挑選/home/pp23/share/hw3-2/cases/ p28k1 (node 數: 28000)以及 /home/pp23/share/hw3-3/cases/c06.1 (node 數: 39857) 作為本次實驗的測資，後者的 matrix size 約為前者的兩倍(matrix size: 784,000,000 v.s. 1,588,580,449)，觀察隨著 problem size 的增大，整體系統效能是否能維持穩定。

	Hw3-2	Hw3-3	Speedup Ratio
P28k1(ms)	18445.20507	11162.661133	1.6524
C06.1(ms)	53371.218750	30368.005859	1.7575

Fig. 12 Weak Scalability 實驗結果展示

從上圖可發現，雖然使用兩顆 GPU(Hw3-2)在相同 test case 下表現比 Hw3-2 要好，但並沒有達到兩倍的 Speedup，此實驗結果與預期相同，因為使用多顆 GPU 需要處理額外的負擔如溝通、if-判斷等，因此此加速效果是可預期的。至於為何 Hw3-3 在處理 c06.1 的時間為何與 Hw3-2 在處理 P28k1 相差這麼大，我原先的理解是後者使用了兩倍的 GPU，其在處理兩倍的資料時間應該與前者相去不遠，但我後來的想法是使用了兩顆 GPU 並沒有獲得兩倍的計算能力，在程式中兩顆 GPU 開的總 blocks 數與單顆 GPU 是相同的，其所獲得的效能提升是兩顆 GPU 中單一 GPU 需要開的 block 數較少，因此比較不需要去競爭 sm，每一個 block 可以比較快的獲得計算資源因此效能上升。

e. Time Distribution (hw3-2)

此節希望分析 hw3-2 的 time distribution，包含 computing time、communication time、memory copy (H2D, D2H)以及 I/O time，但以本題而言，communication 為 host 與 device 的溝通，每一次的 communication 皆涉及 memory copy，因此本題只討論 communication time 而不討論 memory copy。

本題從/home/pp23/share/hw3-2/cases 挑選多測資進行實驗，分析不同 scale 下不同階段耗時

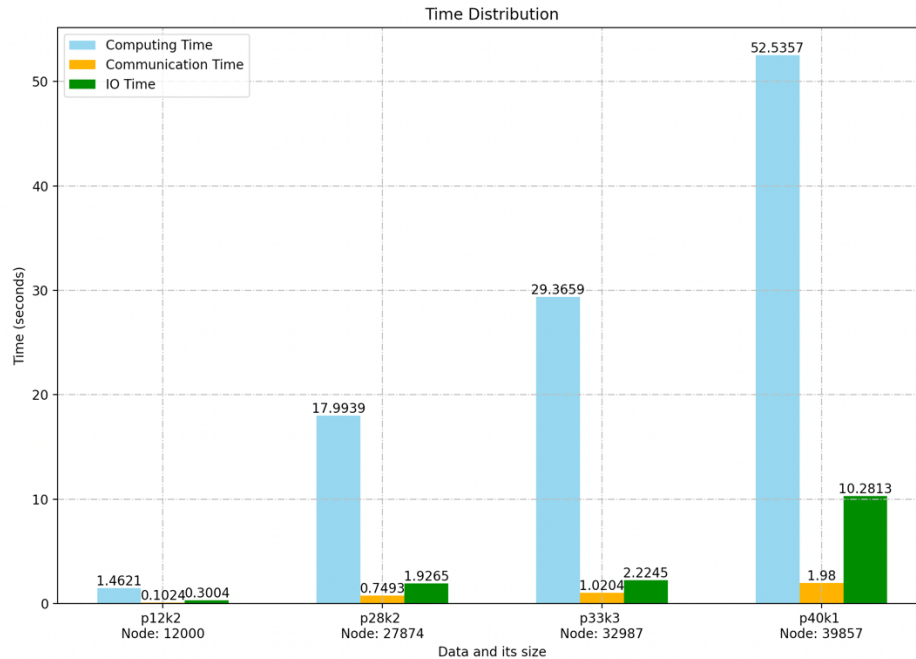


Fig. 12 不同 scale 資料 time distribution 實驗結果展示

Computing time 隨著資料變大而增加，與預期相同。Communication time 也會因為資料變大 memory copy 的資料量變多而成長，但成長幅度不明顯，推測其主要耗時來自於 host 與 device 的傳輸初始化連接，傳輸的資料量與時間並無太大關聯。最後是 IO time，雖然成長幅度沒有 Computing time 大，但也隨著資料變大有明顯成長，因此推測其讀取寫入的資料量也會很大的影響其時間。

4. Experience & conclusion

a. What have you learned from this homework?

透過這次作業學到了很多，以前雖然會拿 GPU train model，但都僅限於透過程式呼叫 API，對 GPU 內部並無深入了解。透過這次作業的實作過程了解到了 GPU 的運算機制、溝通過程、還有其記憶體架構等，需要反覆翻閱上課 ppt 還有一些網路上的文件才能寫出功能正確且效能良好的程式碼。且撰寫執行 cuda 的程式與以往的撰寫經驗大不相同，涉及到 host 及 device 的溝通還有 device 及 device 的溝通，在沒有深入了解其交互溝通和執行順序前有時會卡在一個相當不直覺的錯誤，花了許多時間才解決問題，透過此作業獲得了許多 cuda 程式撰寫的經驗，對我來說獲益良多。

b. Feedback

跟助教說聲辛苦了，到後期常常需要處理 hades server 的問題，在討論區看到許多災情，同學們遇到問題時常常需要助教出手解決，非常感謝助教們的用心。