Parallel Programming
Homework 3: All-Pairs Shortest Path

111065510 黃韋慈

---

1. Implementation

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

- 在 hw3-1 中我選擇 Floyd-Warshall algorithm 作為使用，一般解決最短路徑的方法通常是使用 Floyd-Warshall algorithm 及 Dijkstra Algorithm，但考量到 Floyd-Warshall algorithm 改為平行方式較為簡易加上後續功課需使用 Floyd-Warshall algorithm 的變體 Blocked-Floyd-Warshall algorithm 進行計算，因此希望能先透過 Floyd-Warshall algorithm 進行熟悉．
- 透過 Pthread 與 OpenMP 進行實作．(由於在 hw2 分別試過 Pthread 與 OpenMP+MPI 的組合，因此想要試試看使用 Pthread+OpenMP 進行實作)

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

- Hw3-2

  我主要是夠過使用以 32*32 為 threads unit 的方式去計算和更新，會這樣設定是因為一般通常還是以 32 個 thread 為一個 wrap（像是在我們所使用的設備上 thread 一次最多只能開 1024 個（32*32)），即使通常 thread 的數量都是任意的，但開到最大能夠避免 wrap 內的 thread 閒置，因此考量到 Occupancy Optimization，選擇以 32*32 當成單位去執行．

- HW3-3

  我主要是以 row block 為主，在一開始將整個 input 資料利用 row 平均分配給 GPU，再將有剩餘的 row 數從前面開始加（方法與 hw1、hw2 切割資料的方式相同)，而每次在經過 phase3 階段後只需要傳送 pivot 在的 row block 給另一個 GPU

就好，column block 的部分不需要傳遞，因為每次在 update 時只需要 pivot 在同個 GPU 上同個 column 的部分就好．

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

1. blocking factor,

在 hw3-2、hw3-3 中我將 blocking factor 設為 32 作為使用，其實也就是開到最大，因為沒開到最大反而有可能會浪費 wrap 中的 thread，進而有可能會降低平行化的程度．

2. blocks 與 threads

在不同的 phase 中會準備不同的 block 與 thread 數量

A. Phrase1

在這個階段，因為只會針對一個格子做一次計算和更新，因此只需要準備一個 block，然後其中有 32*32 個 thread．

B. Phrase2

這個階段會開始往同一層的 row 和 column 延伸更新，因此這邊會需要長度為 vertex number/blocking factor 的 block，且每個 block 一樣含 32*32 個 threads．

C. Phrase3

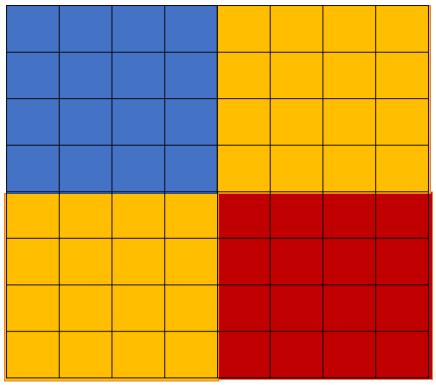
這個階段會處理剩下的 row 和 column 做 update，所以會需要長度為 vertex number/blocking factor*vertex number/blocking factor 的 block，並且每個 block 中的 thread 數也是 32*32．

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

在最一開始要做算法前會先從 host 取全部的資料給不同的 device，由於都是在同 node 上進行，因此在經過每次 phase1-3 階段後就會透過使用 cudaMemcpy 採 devicetodevice 方法將要傳送的 row block（也就是 pivot 在的）傳給另一個 device，而如前面所提，column block 的部分不需要傳遞，因為在更新 block 的時候，只需要 pivot 同 column 在同張卡的部分即可．接著跑完所有迭代後，兩個 GPU 只需要傳送自己有更新的部分就好，因此只需要把一半的部分傳回給 host 即可．

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

- Hw3-2



以此圖為例，在 Phase1 會處理藍色底的部分，Phase2 會處理黃色底的部分，Phase3 會處裡紅色的部分．

i. Phase1：更新 pivot 所在 block 的最短路徑

在實作開始之前由於 block 內部會有相依性，也就是 CUDA 的特色能夠共享資料，因此我將時常需要存取的部分，在這階段即是 pivot 所在的 block 由 global memory 搬進 shared memory．主要處理方式是例如在 1 到 3 中要找最小路徑時，會去取用（1 到 2）+（2 到 3）等等的權重，以

此來找尋最短路徑．最後在比較完畢後會去更新 shared memory 的值，然後再更新到 global memory．

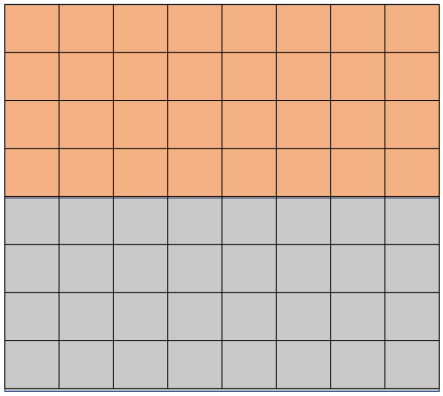
ii. Phase2：更新和 pivot 所在 block 同 row 和 column 的 block 的最短路徑

如同 Phase1 一樣將需要常存取的資料由 global memory 複製進 shared memory，這個階段是將 pivot 和自己所在的 block 以及和 pivot 所在同個 row 和 column 放進來，然後進行計算處理，最後更新 shared memory 後再更新 global memory．

iii. Phase3：更新剩餘 block 的最短路徑

在這階段將自己與 pivot 所在 block 同 row、column 的 block 一起放到 shared memory，因為接續再計算剩餘的 block 時會使用到，而在進行計算後，將結果直接存入 global memory 即可，不需要再存入 shared memory，因為主要都是使用一開始複製進來的資料．

- Hw3-3



以此圖為例，橘色部分為 GPU 0，灰色部分為 GPU 1．

在一開始將資料從 host 複製進 device，接著平均分配要處理的 row block 數給不同的 device（如上圖），若是無法平均分配，則將剩餘的數由前面的數開始發（方法類似於之前 hw1、hw2 的資料分配），透過變數存取這些計算的數，接著進行計

算，其他部分大致與 hw3-2 一樣，只是差在 Phase3 的 block row 的 index 需要去更新，因為 device 每次拿到的資料都要再加上 block offset，最後計算一系列與將要傳送的 row block 的相關資訊，並在每次 phase1-3 階段後就會透過使用 cudaMemcpy 採 devicetodevice 方法將要傳送的 row block（也就是 pivot 在的）傳給另一個 device，而如前面所提，column block 的部分不需要傳遞，因為在更新 block 的時候，只需要 pivot 同 column 在同張卡的部分即可．接著跑完所有迭代後，兩個 GPU 只需要傳送自己有更新的部分就好，因此只需要把一半的部分傳回給 host 即可．

f. Profiling Results (hw3-2)

- 以下針對 hw3-2 透過下列的 metrics 進行測量
  1. occupancy
  2. sm efficiency
  3. shared memory load/store throughput
  4. global load/store throughput
- 使用 nvprof 以 c21.3 testcase 為例：

  Command: srun -p prof -N1 -n1 --gres=gpu:1 nvprof -m sm_efficiency,achieved_occupancy,shared_load_throughput,shared_store_throughput,gld_throughput,gst_throughput ./hw3-2 ./cases/c21.3 my.out

```
    Kernel: phase1(int, int, int*)
        157                            sm_efficiency                   Multiprocessor Activity      4.46%      4.84%      4.83%
        157                        achieved_occupancy                        Achieved Occupancy    0.497609   0.499475   0.497664
        157                            gld_throughput                  Global Load Throughput  4.3300MB/s  68.656MB/s  67.491MB/s
        157                            gst_throughput                 Global Store Throughput  4.3300MB/s  68.656MB/s  67.491MB/s
        157                     shared_load_throughput           Shared Memory Load Throughput  123.22GB/s  141.60GB/s  140.02GB/s
        157                    shared_store_throughput          Shared Memory Store Throughput  61.608GB/s  71.444GB/s  70.018GB/s
    Kernel: phase2(int, int, int*)
        157                            sm_efficiency                   Multiprocessor Activity      96.56%      97.52%      97.48%
        157                        achieved_occupancy                        Achieved Occupancy    0.645042   0.985190   0.982746
        157                            gld_throughput                  Global Load Throughput  1.5171GB/s  2.7149GB/s  2.6886GB/s
        157                            gst_throughput                 Global Store Throughput  1.1837GB/s  1.3542GB/s  1.3419GB/s
        157                     shared_load_throughput           Shared Memory Load Throughput  2500.2GB/s  2860.2GB/s  2834.3GB/s
        157                    shared_store_throughput          Shared Memory Store Throughput  1288.1GB/s  1473.6GB/s  1460.4GB/s
    Kernel: phase3(int, int, int*)
        157                            sm_efficiency                   Multiprocessor Activity      99.93%      99.96%      99.95%
        157                        achieved_occupancy                        Achieved Occupancy    0.853446   0.855737   0.854608
        157                            gld_throughput                  Global Load Throughput  18.913GB/s  37.897GB/s  37.456GB/s
        157                            gst_throughput                 Global Store Throughput  11.035GB/s  12.592GB/s  12.485GB/s
        157                     shared_load_throughput           Shared Memory Load Throughput  3181.4GB/s  3630.4GB/s  3599.6GB/s
        157                    shared_store_throughput          Shared Memory Store Throughput  713.02GB/s  813.64GB/s  806.75GB/s
[TOTAL_TIME] 35027669
```

由 Occupancy 及 sm efficiency 可以看出不同 phase 的 GPU 占用以及使用率，phase 1 因為只計算一個 block，所以占用率及使用率相當低，phase 2 因為是計算與 pivot 所在 block 相同的 row 及 column 的 block，在所有 block 中會包含 pivot block，因此使用率大概 96 - 98%，而 phase 3 則是要計算所有剩餘的 block，因此利用率及占用率來到 99%·

由 global memory 的 throughput 及 shared memory 的 throughput 的數據可以看出，shared memory 的 throughput 幾乎都比 global memory 的 throughput 還要多·

2. Experiment & Analysis

a. System Spec

所有程式皆在課堂所提供的設備上進行運行·

b. Blocking Factor (hw3-2)

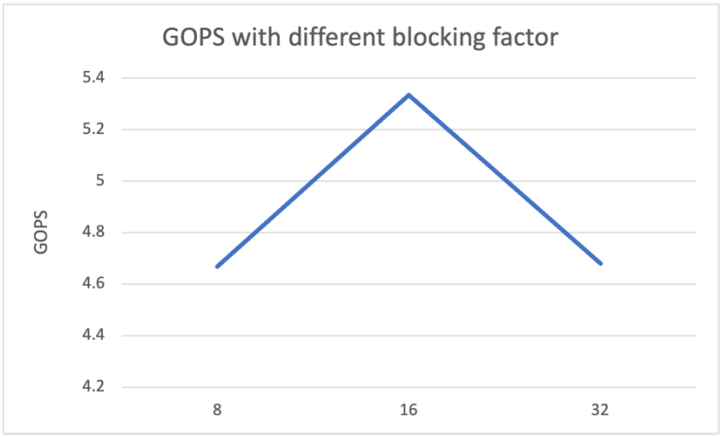
針對 Blocking Factor，測試在不同 Blocking Factor 下的各項表現·

測試案例採用最後一個 c21.3 進行測試·

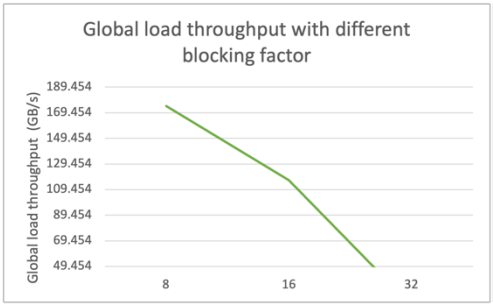
1. GOPS

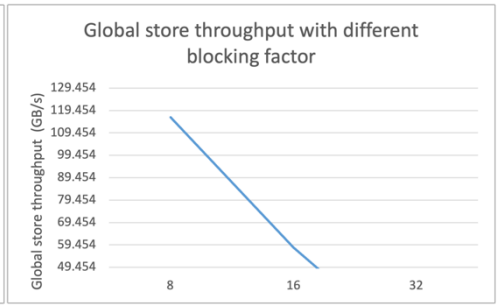
Command：在執行時加上 nvprof --metrics inst_integer ·

下圖的 GOPS 並沒有隨著 blocking factor 的增加而上升，可是是因為 GOPS 主要是計算每個 thread 所計算的 integer 總數的加總在除以時間平均，因此有可能是因為在 blocking factor 等於 8 或 32 得時候就為接近 matrix 的 size，因此需要 padding 的 thread 數量比較少，才讓兩者的數量較為接近·

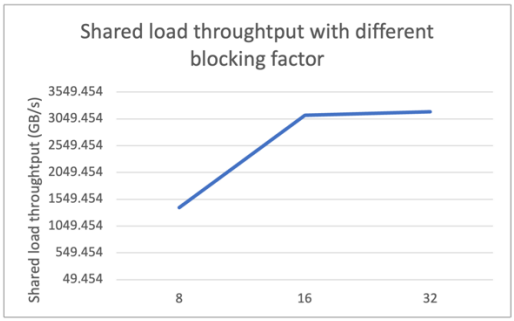


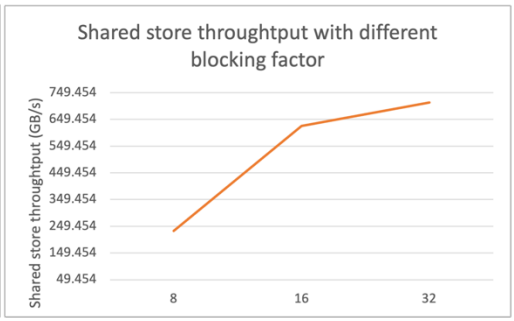
2. global memory bandwidth





3. shared memory bandwidth





c. Optimization (hw3-2)

- Shared memory

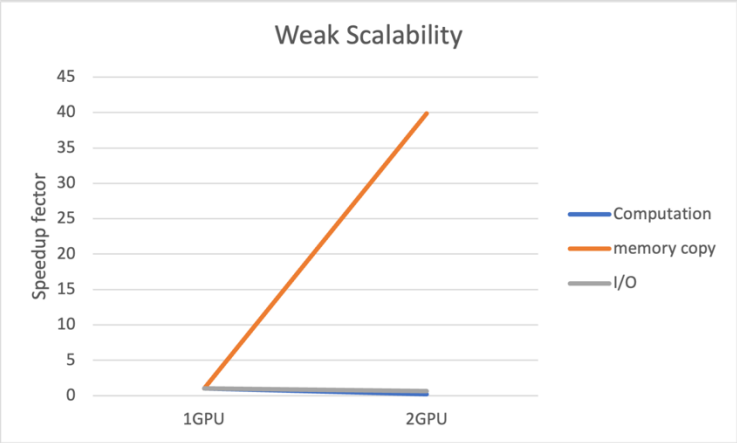
在每次進行計算前，thread 先會把資料從 global memory 複製到 share memory， 計算完再複製回去，以加速每次 thread 計算時取資料的時間．

- Occupancy optimization

將 blocking factor 設為 32，也就是開到最大，也就是一個 wrap 的量，進而避免 wrap 內的 thread 閒置．

d. Weak scalability (hw3-3)

使用 c01.1 與 c05.1 的 testcase 測試在一個 GPU 與多個 GPU 模式的加速情況·



- Time Distribution (hw3-2)

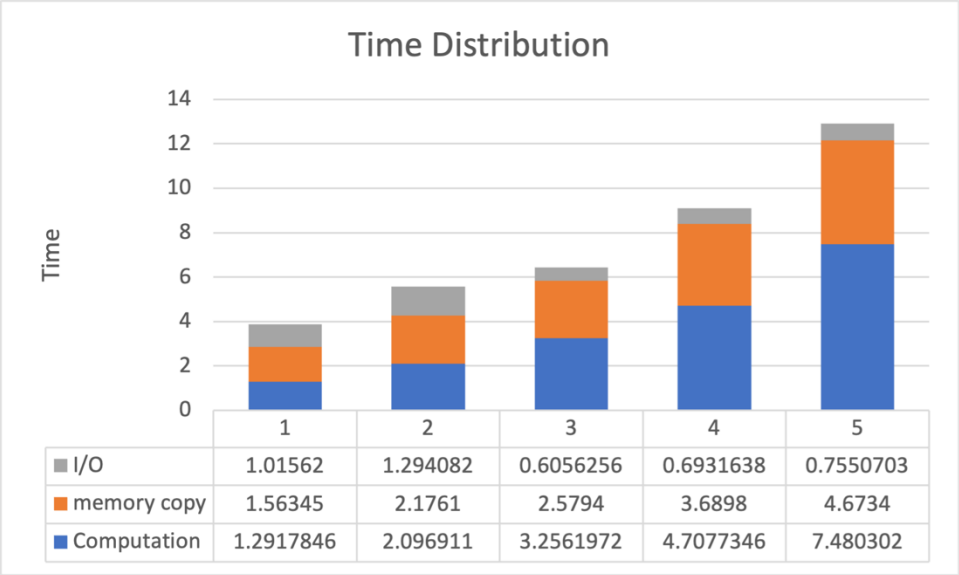
以下根據 hw3-2 分別執行(1: p11k1、2 : p13k1、3 : p15k1 、4 : p17k1、5 : p19k1)的 test case 進行測量·測量對象有：computing、memory copy (H2D+D2H)、I/O

Computing：使用 Chrono 在計算前後加上 std::chrono::steady_clock::now();再進行相減而成·

memory copy：透過在指令上使用--print-gpu-trace 以進行·

I/O：使用 Chrono 在 input 與 output 前後加上 std::chrono::steady_clock::now();再進行相減，然後再加總 input 與 output 的·

由下圖可以看出，computation time 主要是整個程式執行時間長短的決定因素，就上述觀察而言，phase 3 最為耗費時間·



	1	2	3	4	5
I/O	1.01562	1.294082	0.6056256	0.6931638	0.7550703
memory copy	1.56345	2.1761	2.5794	3.6898	4.6734
Computation	1.2917846	2.096911	3.2561972	4.7077346	7.480302

3. Experience & conclusion

這次的作業從一開始一般的平行化算法到逐漸到使用一個 GPU 再到多個，整體設計很完整的讓我們了解其中的不同之處，這次的作業難度真的不低，一開始便花費了好多時間在了解 Blocked-Floyd-Warshall algorithm，了解完後就要開始思考要怎麼改寫，這一步真的卡了很久也有請教去年修過課的學姊，才可以大致找到方向，現在再回頭看才發現之前 hw1、hw2 真的輕易許多．最後跑出來的效能其實都沒有算很好，但主要還是可以運行，三份功課其實都有沒有跑過的 testcase，但因為期中較為繁忙，沒有時間再進一步優化，這一點有點可惜．