Parallel Programming

Homework 4-2 Report: Blocked All-Pairs Shortest Path

108062605 呂宸漢

1. Implementation

Multi-GPU：hw4-2.cu

a. Blocked Floyd-Warshall 的概念其實就是把原本的 Floyd-Warshall 分割成好幾塊區塊，再分割成 spec 上描述的 3 個 phase 執行，在此先稍微解釋一下每個 phase 的執行方法以方便理解接下來的描述。

phase 1：self-dependent blocks

也就是 pivot block 內自己做 Floyd Warshall，$D_{i,j}^{K}$ 就是比較 vertex i 到 vertex j 的距離與 vertex i 經過 vertex k 到 vertex j 的距離，取最小的距離並更新 $D_{i,j}^{K}$，K 的範圍就是 block 內所有的 vertex id。若將 i 與 j 標準化則會發現其實這個階段就是小規模的 Floyd-Warshall。

phase 2：pivot-row and pivot-column blocks

也就是將 pivot 所在的 row 及 column 的 block 內的 vertex 距離，一樣透過 Floyd-Warshall 計算最短路徑，不過這次是用原路徑與經過 pivot 計算出的路徑取最小的距離當作最短路徑，以 $D_{i,j}^{K}$ 為例，$D_{i,j}^{K}$ 就是比較 vertex i 到 vertex j 的距離與 vertex i 經過 vertex k 到 vertex j 的距離，其中 vertex i 到 vertex j 就是 pivot 所計算的數值，讓 pivot 所在的 row 及 column 的 block 都更新為經過 K 個 vertex 的最短距離。

phase 3：other blocks

用剩餘 block 的位置可以找到該 block 對應的 pivot-row 及 pivot-column，以 $D_{i,j}^{K}$ 為例，$D_{i,j}^{K}$ 就是比較 vertex i 到 vertex j 的距離與 vertex i 經過 vertex k 到 vertex j 的距離，其中 vertex i 到 vertex k 的距離是 pivot-column 所計算的數值，vertex k 到 vertex j 的距離是 pivot-row 所計算的距離。計算完後就完成了這一個 round 的計算。

b. 我是用 GPU block 的 thread 總數當作基本單位分割原本的 distance matrix，如此 GPU 內一個 block 的 thread 就可以一對一 handle 每個 entry，再將 block 組合成 GPU 可以 align 的 2D grid，如此就可以平均分配所有 entry 給所有 thread，雖然會有 thread 因為 padding 的關係被浪費，memory 也會因為 padding 的關係被浪費，不過比起要用判斷式處理而脫慢速度，這樣的方式會比較有效率。最後再將 grid 切成上下兩塊，分配給兩塊 GPU，兩塊 GPU 開的 global memory 與原本的 distance matrix 相同大小，只是需要計算的數量變成原本的一半（Figure 1），即分配完成。

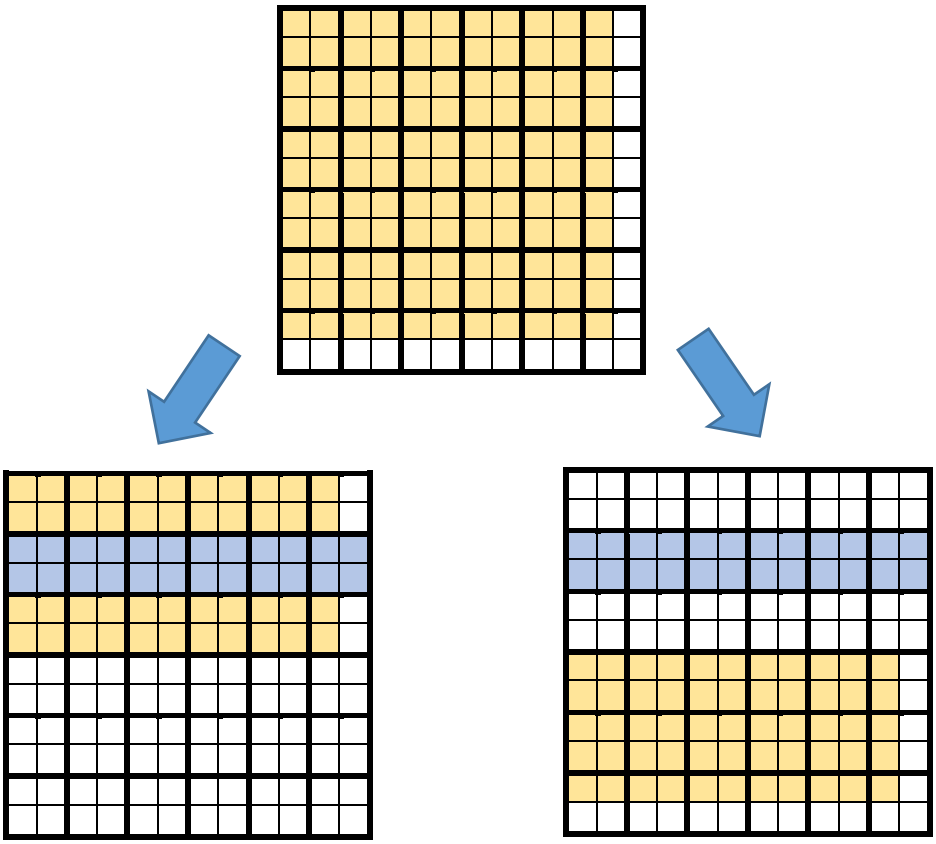


Figure 1. 上圖展示 vertex 數為 11，1 warp = 2 thread，1 block 有 4 個 thread，每個粗線框為一個 block，整個粗線框則是 grid 大小，左邊及右邊的圖分別代表兩片 GPU，黃色的部分為分配完的 data，白色的部分則是 padding，藍色的部分是交換的 pivot row。

c. 兩張卡之間的傳輸我是用 device to device 的 memory copy，而且每個 round 只傳遞 pivot row 的資料而已，選擇 device to device 是為了降低透過 host 的傳遞 overhead，只傳遞 pivot row 則是減少傳遞的資料量。

2. Experiment & Analysis

a. System Spec

所有程式皆在課程所提供的 cluster 上進行測試。

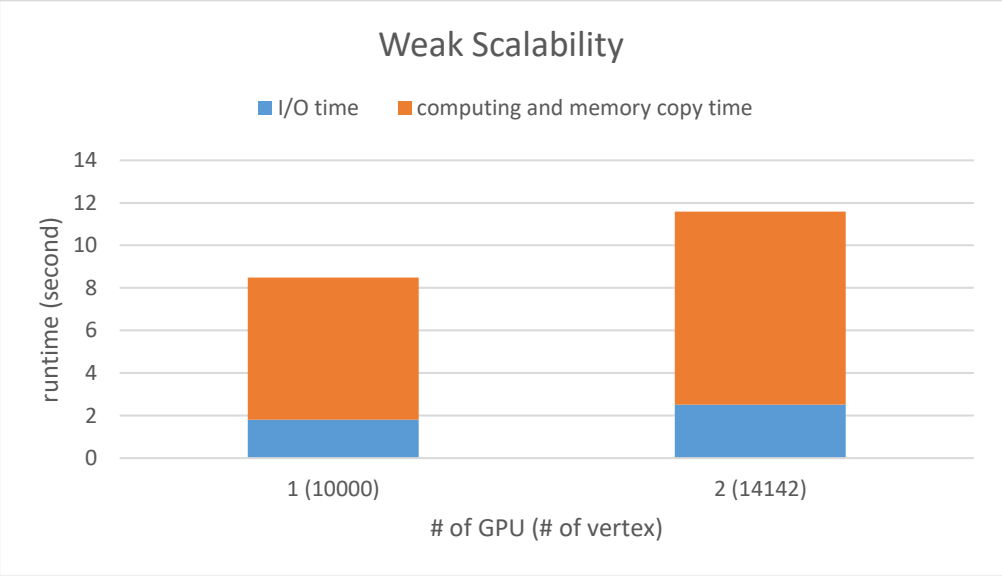
GPU imformation：

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.81                 Driver Version: 384.81                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080    On   | 00000000:4B:00.0 Off |                  N/A |
| 32%   47C    P8    14W / 216W |      2MiB /  8114MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080    On   | 00000000:4D:00.0 Off |                  N/A |
| 10%   47C    P8    18W / 216W |     11MiB /  8113MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

b. Weak Scalability

weak scalability 的計算方式是在基於每個計算單元的計算量皆相同的情況下，比較不同數量的計算單元完成計算所需要的執行時間，若執行時間皆相同，則表示 weak scalability 良好。由於在 blocked Floyd-Warshall 中，實際計算的數量是 vertex 數的平方倍，因此假設 single GPU 的計算數量是 1 倍的資料量的話，則兩片 GPU 需要計算的資料量則是兩倍，以 vertex 數而言，multi-GPU 需要計算的 vertex 數為 single GPU 的$\sqrt{2}$倍，經測試後就會得到以下圖表，其中單一 GPU 的 vertex 數為 10000，兩片 GPU 的 vertex 數為 14142。

由下圖可見，在每個計算單元計算相同資料量的情況下，weak scalability 並不是很好，我認為是因為兩張卡之間仍然需要 memory copy 傳遞資料，即使是使用 device to device 傳遞，仍然有額外的 overhead，才呈現這樣的結果。



c. Time Distribution

i. computing time

用 nvprof 取得 3 個 phase 的執行時間。

ii. communication time

runtime 減掉 computing time、memory copy time 與 I/O time 即可。

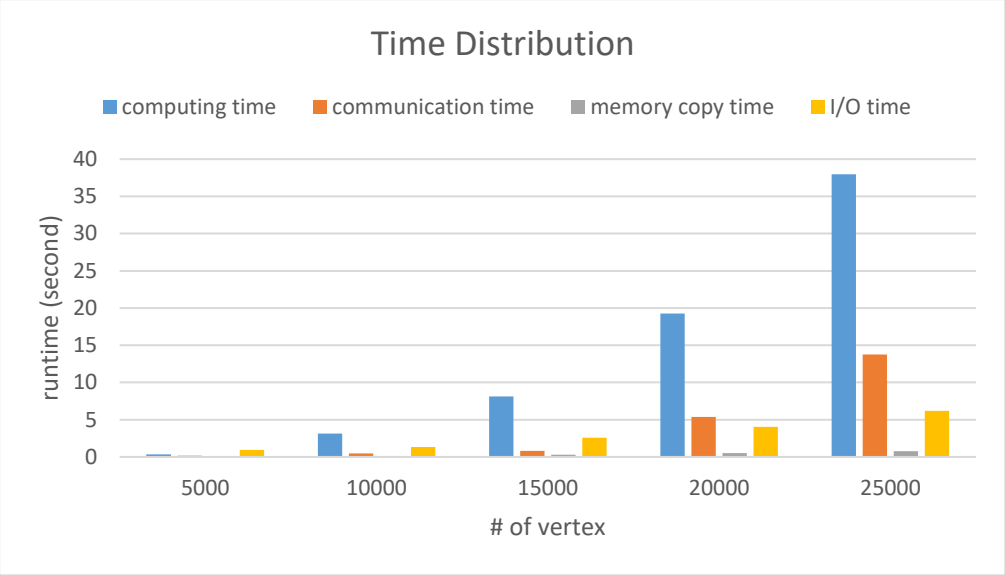
iii. memory copy(H2D, D2H)

用 nvprof 取得 memory copy 的執行時間。

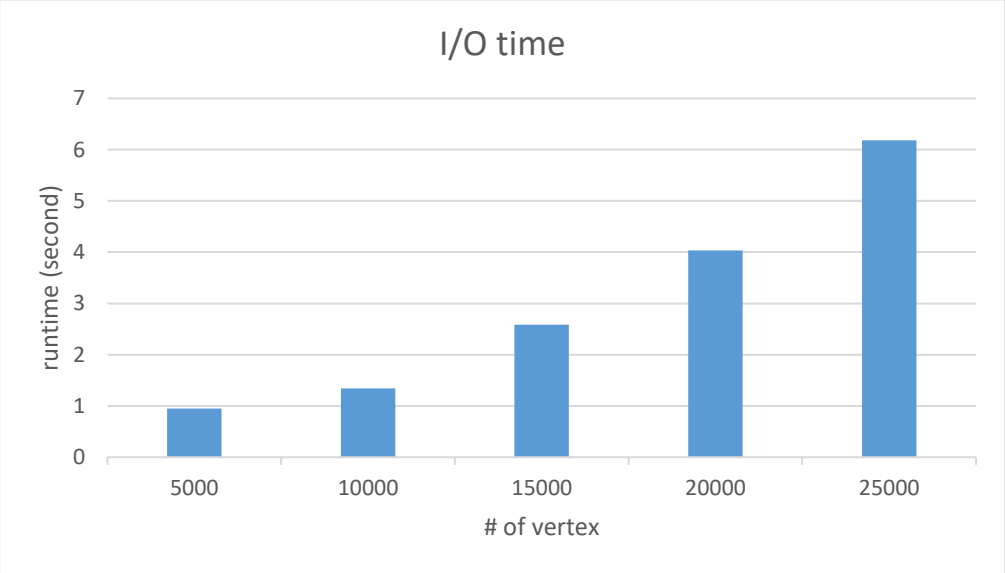
iv. I/O time

用 clock_gettime(…)取 CLOCK_MONOTONIC 的時間，將此函式加在檔案 input/output 的前後取差值相加即可。

由下圖可以看出，communication time 會隨著 vertex 數變多跟著變大，甚至在 25000 個 vertex 的 case 中，已經快到 computing time 的一半了，尤其 computing time 中最花費時間的就是 phase 3，因此最需要對 phase 3 及 communication 優化，才能真正減少整個 case 的執行時間，I/O time 也是關鍵，在大的 case 中 I/O 也花費了許多的時間，也必須對 I/O 優化。

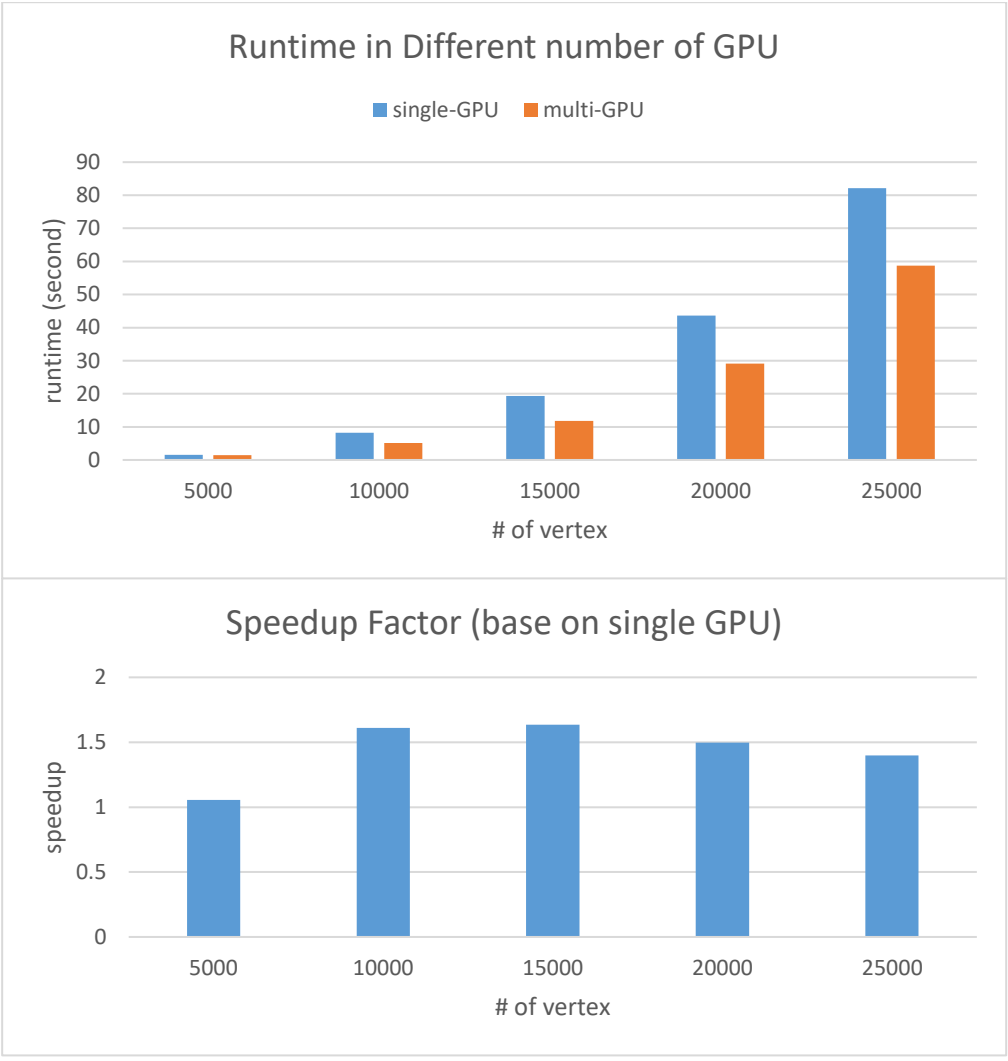


由下圖顯示，因為我沒有平行化讀寫檔的程式，導致讀取時間會隨著資料數越大而越長，因此若時限一直是 30 秒的情況下，這不是一個好的 I/O 策略，應該要平行化加速以防 I/O time 佔執行時間越多。



d. Other

這是我額外做的圖表，用以說明不同數量的 GPU 對執行時間的影響及使用 multi-GPU 基於 single-GPU 的加速情形，即 single-GPU runtime / multi-GPU runtime。由下圖的資料可以看出，兩張 GPU 並沒有加速到兩倍，而且隨著 vertex 數變多，加速越少，由 time distribution 的圖表可以得知，應該是 communication time 越花越多導致的。



3. Experiences / Conclusion

這次作業就是沿用之前 hw4-1 的程式下去改，主要的加速方式就是減少 GPU 間的溝通，我選擇使用 device to device 的 memory copy，避免透過 host 進行傳遞的 overhead，由於我是將原本的 distance matrix 切成上下兩塊分配給兩塊 GPU，因此每一 round 只需要傳遞 pivot row 即可，避免同步兩個 matrix 造成額外的 memory copy。雖然這次作業看似簡單，可是計算的速度是建立在在 hw4-1 的基礎上，當 hw4-1 優化不良，也會影響這次作業的執行速度，GPU 真的不太好優化，再上次作業後有與同學分享作法，會覺得明明想法很類似，可是時做出來的速度卻不同，是蠻不好優化的平行方式。