CS542200 Parallel Programming Homework 3:

All-Pairs Shortest Path

112062673 吳文雯

1.Implementation

a. CPU version(hw3-1):

我選擇的演算法是 Floyd-Warshall，因為他可以達到的平行度較高。一開始使用 openMP 來實作，在初始化 dist 陣列及更新中間的 d(i,j)時，透過 omp parallel for 來平行計算，但是最後的兩個 cases 會超時而無法通過。因此後來改用 pthread，在 floydwarshall 函式中，讓每個 thread 負責一部份的 i，分配方式是用 vertex 數量除以 thread 數量，再另外 handle 有餘數的情況，盡量讓每個 thread 分到的 i 數量較為平均。

b. Single GPU version(hw3-2):

首先根據 vertex 的數量決定 matrix size，假如他不是 blocking factor(B)的倍數，就調整 matrix size(例如 block factor=64, 當 v=40 時 matrix size 就取 64，而當 v=65 時就以 128 作為 matrix size)。讓 matrix size 成為 blocking factor 的倍數是為了減少 branch 的使用，讓所有 block 中的 threads 進行相同指令。而 3 個 phase 所需要進行的回合數(number of blocks)為 matrix size 除以 blocking factor。

由於我們使用的 gpu 中，一個 block 的 threads 數量上限為 1024，而在這個計算問題中，二維的 block dimension 較為合適，因此使用(32,32)的方式來宣告 block dimension。

一開始為了計算方便，我將 blocking factor 設為 32，讓 thread 一對一處理每個 entry，但是即使已經將每個 block 的所需 data 都搬到 shared memory，還是無法通過最後的 10 筆測資。後來發現我只使用了 1/4 的 shared memory，因此決定提高 shared memory 的使用，以減少在 global memory 及 shared memory 之間搬運資料的次數。

透過 deviceQuery 得知 shared memory 的大小為 49152 bytes，其中一個 integer 需要 4 bytes，而在 phase 3 中會同時需要 3 個 entry 來更新結果，40912/(4*3)=4096 正好為 64 的平方，因此採用 64 做為新的 blocking factor。接著就需要分配 64*64 個 entry 給 32*32 個 threads，每個 thread 需要負責 4 個 entry，考量到 block 中以 32 個 thread 為一個 warp，我讓 thread 以在 x,y 方向皆間隔 32 的方式來分配 entry。例如 threadId 為(0,0)的 thread 負責(0,0),(0,32),(32,0),(32,32)等。

接著說明在每個 phase 中的實作:

Phase 1:

在 pivot block 中進行 Floyd-Warshall 計算，使用 B*B 大小的 shared memory，這個階段的計算量是最少的，程式碼也較為簡潔，因此我也在這裡改變資料存到 shared memory 的方式，以避免 bank conflict。

在這個階段中的 k 具有相依性，因此在 for loop 中需要加上_syncthreads()，讓前一輪都計算完後才進行下一輪。

Phase 2:

這個階段分別計算 pivot row 及 pivot column 中的 block，由於這兩者並沒有 dependency，因此我利用兩個 stream 分別計算。

Phase 2_1 計算 pivot row 的部分，需要的 data 為 pivot block 及自己 block 的資料，因此使用 B*B*2 大小的 shared memory。這裡的 k 並沒有 dependency，只要在 for loop 開始前加上一個_syncthreads()，確保開始計算前已經將所需資料都搬進 shared memory。

Phase 2_2 計算 pivot column 的部分，需要的 data 為 pivot block 及自己 block 的資料，因此使用 B*B*2 大小的 shared memory。其餘的進行方式如同 Phase 2_1。

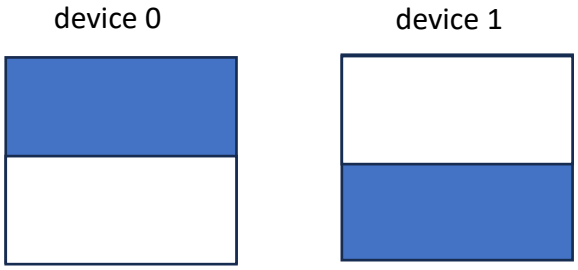
Phase 3:

這是計算量最大的階段，所需的資料除了自己的 block 還有對應的 pivot row block 及 pivot column block，因此使用 B*B*3 大小的 shared memory，在這裡可以將 shared memory 的利用達到上限。這裡的 k 同樣沒有 dependency，因此 for loop 內不需要加上 _syncthreads()。

c. Multi-GPU version(hw3-3):

我使用 OpenMP 產生兩個 threads 來處理兩個 gpu 的計算，選擇 omp 是因為程式碼較簡潔不易出錯。

這裡的實作大部分跟 hw3-2 相同，在每個 gpu 上配置的大小也相同，主要是針對 phase 3 這個計算量最大的階段，將工作量分到兩張卡上。分割方式是將 matrix 分成上下兩部分，假如 matrix size 是奇數，多出來的部分就會分配到 device 1 。為了計算方便，device 0 的資料會放在上半部，device 1 的資料會接續放在下半部，也就是說兩者的 index 是連續的，只是分別在不同的 gpu 上，示意圖如下:(正方形為整個 malloc 空間，藍色部分為放置資料位置)



兩張 gpu 的溝通透過 cudaMemcpyDeviceToDevice，在每一回合傳遞 pivot 所在 row 的資料，在 cudaMemcpy()的程式碼之後必須加上 omp barrier 確保完成資料傳輸後才繼續計算。

2.Profiling Results

使用 p14k1 進行測資，利用 nvprof 取得資訊如下:

Kernel: phase 3

	min	max	Avg
occupancy	0.926044	0.927686	0.926935
sm efficiency	99.84%	99.95%	99.94%
Shared memory load throughput	2818.8 GB/s	3198.1 GB/s	3130 GB/s
Shared memory store throughput	259.48 GB/s	265.1 GB/s	261.37 GB/s
Global memory load throughput	18.375 GB/s	18.804 GB/s	18.502 GB/s
Global memory store throughput	63.491 GB/s	65.224 GB/s	64.672 GB/s

根據這個結果可以發現，這個程式的 occupancy 以及 sm efficiency 算是蠻好的，但是在 global memory 的讀寫就非常不理想，可能還有優化的空間，不過幸好透過充分利用 shared memory，需要進行 global memory 讀寫的次數降低了許多，也因此讓整體效能不至於太差。

3. Experiment & Analysis

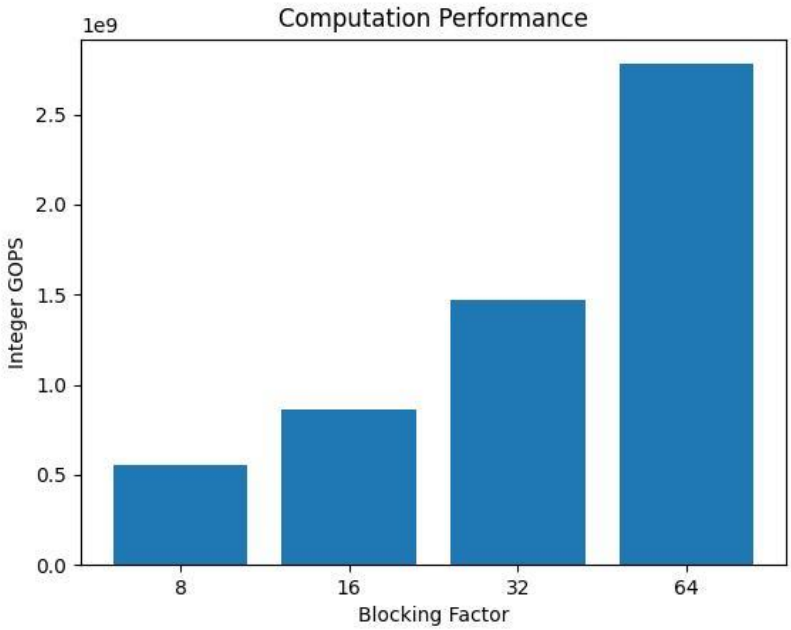
a. System spec: 使用課堂提供的 hades。

b. Blocking Factor :使用 p14k1 進行測試，vertex 數為 11000。

(1) Computation performance:

這裡的數據是透過 nvprof 設定參數 inst_integer 取得，以下顯示最大的 kernel:phase3 的數據。

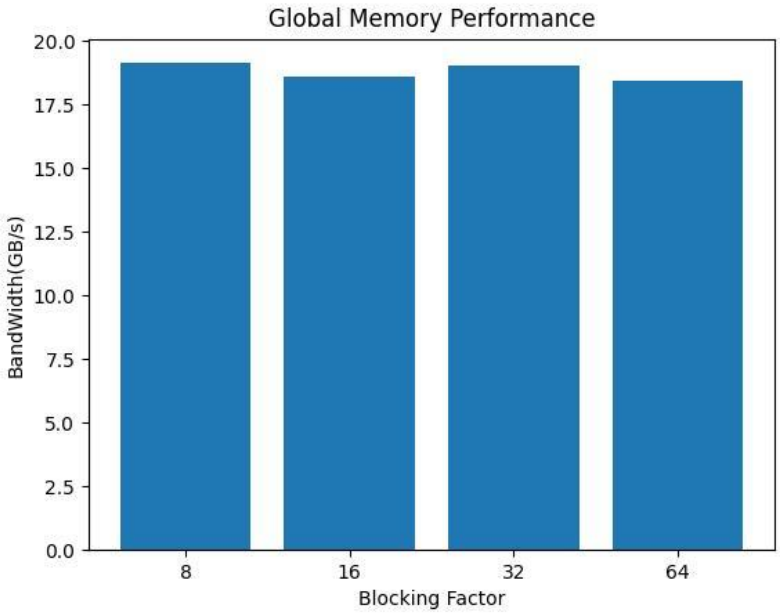
從下圖可以看見，隨著 blocking factor 增加，GOPS 也逐漸增加，表示使用越大的 blocking factor 確實有較好的效果。



(2) Global memory bandwidth:

這裡的數據是透過 nvprof 設定參數 gld_throughput 取得，以下顯示最大的 kernel:phase3 的數據。

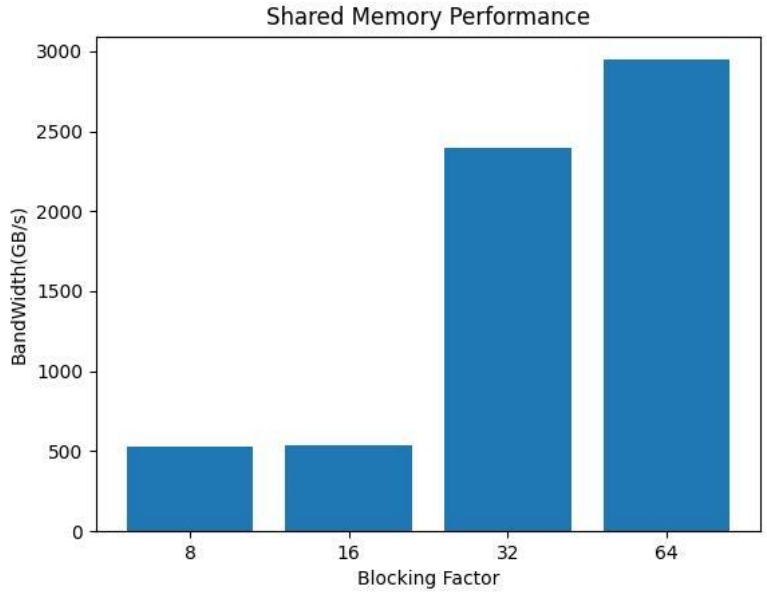
從下圖顯示出 bandwidth 在各個 blocking factor 的表現差不多，我覺得原因可能在於我的 global memory bandwidth 實在不是很理想，所以看不太出來在各個情形下的差異，這是需要在優化的地方。



(3) Shared memory bandwidth:

這裡的數據是透過 nvprof 設定參數 shared_load_throughput 取得，以下顯示最大的 kernel:phase3 的數據。

從下圖顯示出 bandwidth 從 16 改成 32 時有顯著的提升，然而到 64 時雖然也有變好，但進步幅度就沒有那麼大了。

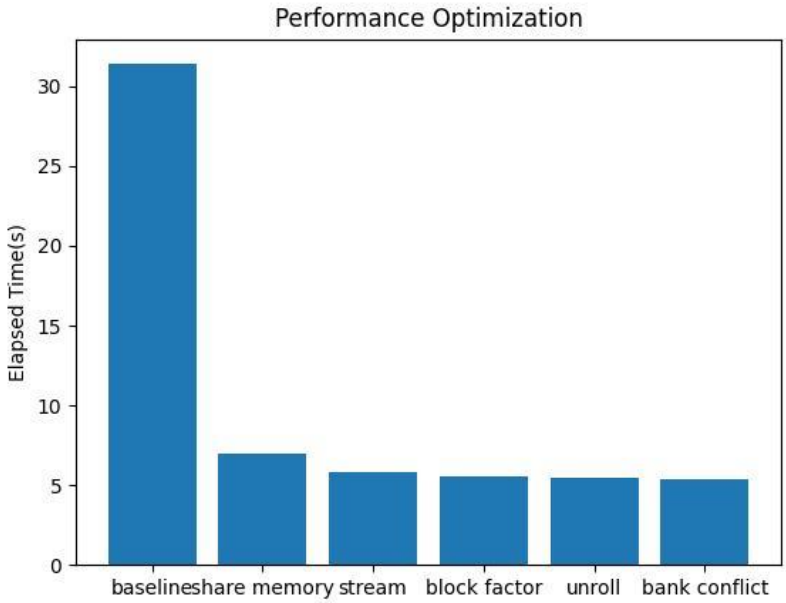


c. Optimization:

在 hw3-2 我使用了以下優化技巧:

1. Occupancy optimization: 在一個 block 中使用達到上限的 threads，不過由於我一開始就採用這個方法，因此沒有特別在下圖中呈現優化後的差異。
2. Shared memory: 將要使用的 data 都搬到 shared memory，等計算完一個 phase 在搬回 global memory。
3. Streaming:使用兩個 stream 分別計算 phase 2 中的 pivot column 及 pivot row。
4. Large blocking factor: 一開始採用 32，後來加大到 64。
5. Unroll: 在更新 k 的 for loop 前使用#pragma omp unroll。
6. Handle bank conflict: 這個技巧是使用在 phase 1，有調整放入 shared memory 的位置，避免同一個 warp 的 thread 產生 bank conflict。

優化結果如下圖，優化方式是由左至右逐漸加進去的:



從圖中可以明顯看出採用 shared memory 的優化效果是最明顯的，這是因為 global memory 跟 shared memory 的 bandwidth 差距很大，因此要充分利用 shared memory 並減少從 global memory 讀寫的動作。其餘的方式雖然也有優化結果，但並沒有非常顯著的減少執行時間。

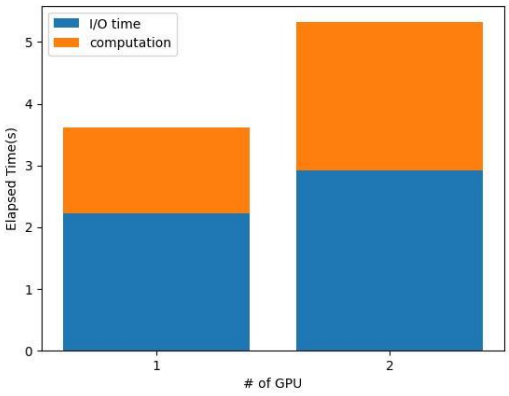
d. Weak scalability:

根據 Weak scalability 的定義，運算量必須跟使用資源成比例，而在這個作業的問題當中，程式的實際運算量為 vertex 數的平方，因此我採用以下 cases 來測試:

1 張 gpu: P11k1: vertex=11000->v^2=121000000

2 張 gpu: P15k1: vertex=15000->v^2=225000000

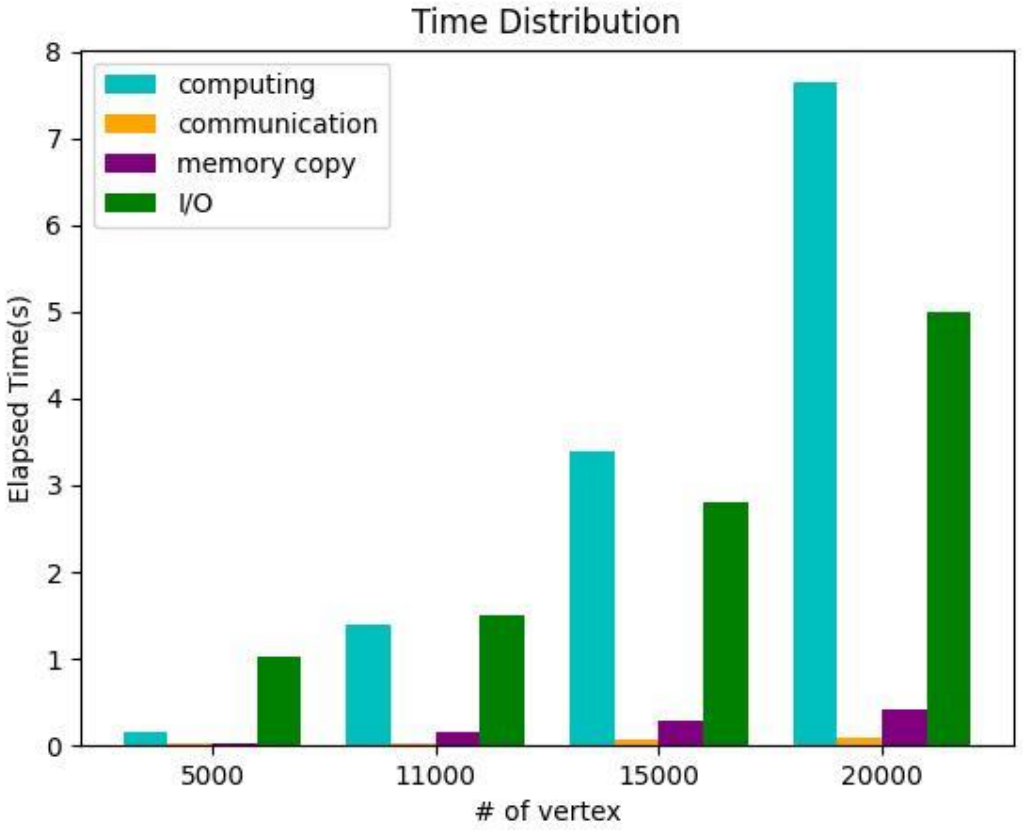
實驗結果如下:



由於執行時間很短，因此放在圖表中看起來兩者差異有點大，但如果看實際數字其實兩張 gpu 所花的時間跟 1 張 gpu 是差不多的，因此我認為結果還算不錯。

e. Time Distribution:

- Computing time: 使用 nvprof 加總所有 phases 花費時間。
- Communication: runtime 扣掉其他三個時間取得。
- Memory copy (H2D, D2H): 使用 nvprof 取得。
- I/O time: 使用 clock_gettime(CLOCK_MONOTONIC…)在 input 及 output 前計算時間。



從圖中可以看出，computing time 所花費的時間最長，且會隨著 input size 增加而顯著提升，是整個程式執行的 bottleneck，如果要減少程式執行時間，就要對計算量最大的 phase 3 進行特殊優化才會有比較明顯的效果。另外，由於沒有對 i/o 進行平行化，他所花費的時間也很長，且也會隨著 input size 增加而上升。Communication time 的部分非常少，而 memory copy 雖然也會隨 input size 增加，不過並沒有花費太長的時間。

4. Experiences / Conclusion

- 這次花費非常長的時間才完成作業，從一開始要了解每個 phase 的行為，決定如何讀取資料，就花了很長時間才搞清楚，尤其在 gpu 的 code 中無法像平常用 cpu code 一樣印出 debug 訊息來觀察，而且在 gpu code 中也無法讀取沒有事先定義好的 global variable，因此初期常常發生 segmentation fault 的情況。
  好不容易完成一個可以產出正確結果的版本，卻發現無法通過最後幾筆測資，為

了將 blocking factor 改為 64，整個程式必須做大幅度的調整，因為一個 thread 所要處理的資料數量跟位置都不一樣了。

- 透過這次的經驗學到了許多優化 gpu code 的方法，也理解到上課教的優化概念如何實作在程式中，其中最印象深刻的是，stream 的使用並沒有想像中那麼困難，不過得到的效果在這個程式中好像也沒有太明顯，可能是沒有用在最適合他的地方。