Parallel Programming Homework 3: All-Pairs Shortest Path

107060002 曹立元

Part I、Implementation

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

   在 hw3-1 中我選擇的演算法是 Floyd-Warshall，因為老師上課時有說過這是一個很適合被平行的演算法。

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

   (1) hw3-2

   基本上我切割 data 的方式就是以 Block_FW 中的大 block 來分，在 hw3-2 中，我把 blocking factor 設為 64（原本是設成 32，但發現太慢了），blockDim 設為 (32, 32)，代表新的 submatrix 裡的每個 block 可以被切成 2x2 個小 block。

   而因為 blockDim 為 (32, 32)，所以如果每個 thread 對應到一個 pixel 的話剛好可以分配完一個小的 block 的運算，因為現在有 2x2 個小的 blocks，所以每個 thread 除了負責他原本對應到其中一個 block 的某個 pixel 外，也負責在其他小 block 中相對應 pixel 的運算，在這樣每個 thread 負責 4 個 pixel 的運算之下，便可以將 kernel 中的每個 block 對應到新的 BxB submatrix 的每個 block。

   (2) hw3-3

   在 hw3-3 中每個 phase 的操作基本上和 hw3-2 一樣，唯一新增的部分是在 phase3 時我把資料上下切一半，分別給兩張卡去做運算，而只在 phase3 做 partition 的原因是因為我把 single GPU 版本的 code 拿去做 profiling 之後發現 phase1、phase2 的運算時間佔比非常小，沒必要再做 partition 去承擔 device 之間溝通的 overhead。

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

   (1) hw3-2

   我在 hw3-2 中的 blocking factor 為 64，因為實作完 blocking factor 為 32 的版本後發現太慢了，所以決定增加一倍試試看，而且在 blocking factor 為 64 時，一個 2*B*B*sizeof(int)大小的 shared memory size 為 32768 bytes，更加接近 shared memory 上限的 49152 bytes，所以 64 也是以 2^n 來設定 blocking factor 大小時的最大值，最後決定選擇 64 當作 blocking factor。

而#blocks 在三個 phase 中分別為 1, (round – 1, 2), (round – 1, round – 1)，因為在第一個 phase 中只需要處理 pivot block 一個 block 就好所以把 block size 設為 1。

而在 phase2 中因為有 pivot row/col 要處理，所以總共需要(round – 1) * 2 個 blocks (因為新的 submatrix 為 round x round 的 matrix，所以每個 row/col 有 round 個 block，再扣掉 pivot block 所以 pivot row/col 各需要 round – 1 個 blocks)。

在 phase3 中因為要處理 pivot block、pivot row/col blocks 以外的所有 block，所以共需要 (round – 1) * (round – 1) 個 blocks。

#threads 為 (32, 32)，因為 Maximum number of threads per block 為 1024，而且化為 (32, 32) 時可以很容易的對應到 block 中的位置去分配計算工作。

(2) hw3-3

在 hw3-3 中 blocking factor 和 #thread 都跟 hw3-2 一樣，原因也相同，不一樣的是 phase3 的 #blocks 數量，我把新的 submatrix 上下切一半分別分給兩張卡，所以每張卡基本會有 round / 2 個 block 要處理，如果 round 為奇數時，則 GPU1 會再多拿一個 row 的 blocks。

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

我是在每一輪的 3 個 phase 做完之後，在下一輪開始進行之前利用 cudaMemcpy 把前一輪算好的資料 copy 給 Host，再將資料分別 copy 給兩張卡，以達到 synchronize 的效果。

一開始我的做法是兩張卡都把自己上一輪算好的 part 傳給 Host，在各自從 Host 中 copy 別人上一輪算好的東西到自己這裡，等於是擁有整個 matrix 最新的資訊，但發現這樣子會非常慢。後來我發現因為兩張卡只要算各自的那半邊，而 pivot column 最新的資訊都已經存在自己這了，只需要再把 pivot row 的資訊 copy 過來便可以完成在自己那半邊計算 3 個 phase 所需的所有資訊，多餘的資料是不需要的，所以改成在每一輪做完之後，只要將上一輪算好的 pivot row 資訊 copy 給兩張卡就好。

5. **Briefly describe your implementations in diagrams, figures or sentences.**

I/O 的部分我用範例 code 給的方法去做，各個 phase 的演算法我也是照著 spec 中給的資訊去刻的，其他關於 blocking factor 的設定、怎麼分配計算、溝通等等方法都已經在前面有提到了。

Part II、Profiling Results (hw3-2 kernel3 with testcase p11k1)

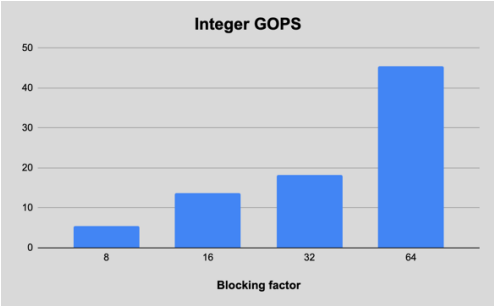
Profiling result (p11k1)	Min	Max	Avg
Occupancy	0.907852	0.912083	0.908777
sm efficiency	99.76%	99.93%	99.90%
shared memory load throughput	3093.4GB/s	3453.6GB/s	3394.8GB/s
shared memory store throughput	136.03GB/s	138.16GB/s	137.54GB/s
global load throughput	16.574GB/s	18.935GB/s	18.826GB/s
global store throughput	64.983GB/s	71.571GB/s	70.580GB/s

Part III、Experiments / Analysis

1. Blocking Factor

第一個是 Integer GOPS，可以看到隨著 Blocking factor 的增加 GOPS 也有顯著的上升。

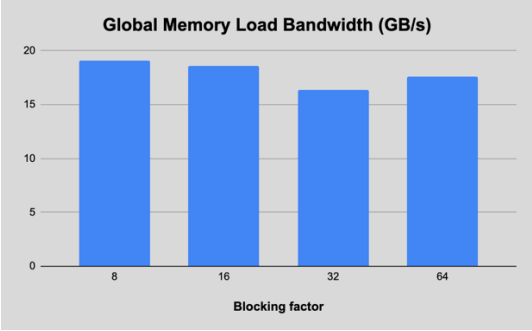
(1) Integer GOPS

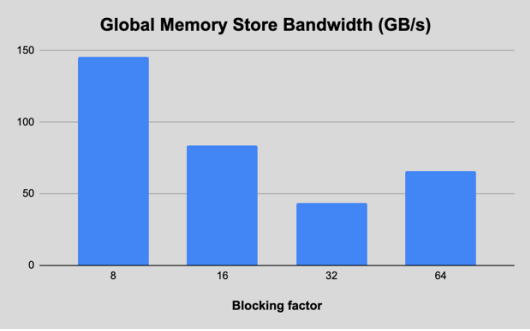


再來是 global / shared memory 的 load / store bandwidth，首先在 global / shared memory load bandwidth 的部分，可能因為只是要把 input file 存到 distance matrix 中，所以大家要處理的資料量差不多，在 global memory load bandwidth 上並沒有顯著的差異。在 shared memory load bandwidth 方面，便可以明顯地看出隨著 blocking factor 增加，意味著 allocate 的 shared memory size 更大，在 bandwidth 上有明顯的成長。

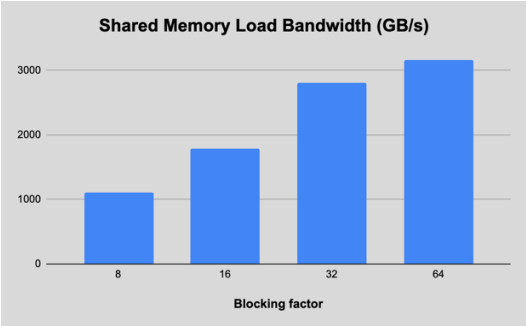
再來是 global / shared memory store bandwidth 的部分，bandwidth 隨著 blocking factor 增加到 32 時遞減，而在 blocking factor 為 64 時又回升，我也不太確定為什麼會這樣，也許是 threads 與 pixel 的 mapping 以及 block 數量多寡的差異所造成。以下為實驗結果的圖表：

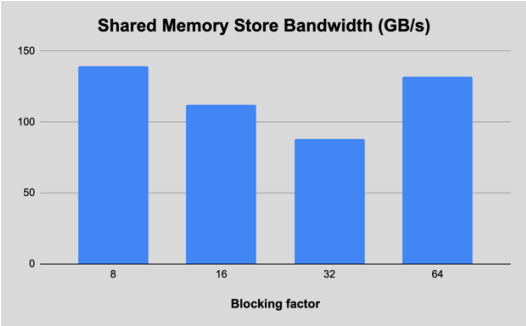
(2) Global memory load / store bandwidth





(3) Shared memory load / store bandwidth





2. Optimization

在 optimization 的部分，我是一次直接做到一個段落，沒有像 spec 中拆的那麼細或是加太多優化，所以只分成加上 shared memory 以及加大 blocking factor 兩個部分，由於 GPU baseline 幾乎跑不動比較大的測資，所以並沒有辦法畫出適合比較的圖表，我就直接放上實驗數據。

在比較小的測資時，可以看到無論加上 shared memory 或再把 blocking factor 加大，都可以在非常小的時間內完成。在比較大的測資 (這裡是拿 p20k1) 時，可以發現把 blocking factor 從 32 加大變成 64 後，速度又可以再快將近 2.3 倍，證明他是有用的。

hw3-2 Optimization			
	p20k1	c14.1	c16.1
GPU Baseline	算不出來	126.930817	748.308254
Add shared memory	23.841897	0.189292	0.302797
scaled up B from 32 to 64	10.54191	0.197763	0.287316

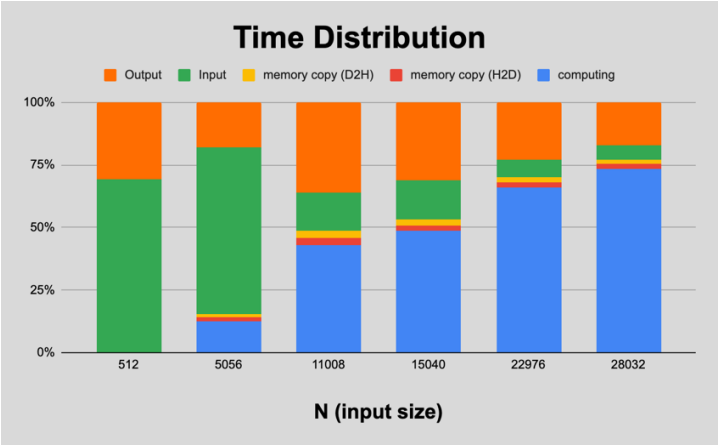
3. Weak scalability of multi-GPU

我拿了幾筆比較大的測資分別跑在單一 GPU 和雙 GPU 上，發現大概都只能 speedup 1.38 ~ 1.5 倍左右，並沒辦法因為兩張卡就加速到兩倍，證實了 multi-GPU 的 weak scalability。

hw3-3 Weak scalability				
testcase	p30k1	p31k1	p32k1	p33k1
single GPU	30.123712	35.8874	38.76323	41.597257
multi GPU	21.721846	24.47982	25.78958	27.372058
SpeedUp	1.386793369	1.465999341	1.503057824	1.519697825

4. Time distribution

從圖中可以發現，對於較小的測資來說 I/O 是他的 bottleneck，而隨著 input size 逐漸加大，program 執行的時間也逐漸被 computing time dominate。而 memory copy time 的部分雖然與 input size 有正相關，但對於整個程式來說所佔的時間比並不大。



Part IV、Experiences & Conclusion

這次作業真的是寫了好久，因為對於 cuda code 還不太熟悉，除了不太知道怎麼寫之外，也不太會 debug，但確實也是學到了很多。實際操作之後才能更深刻的體會原來使用 global / shared memory 時 access 的速度會差那麼多，再來是如果有做好 pattern 的安排也可以讓速度大幅提升，總而言之寫完之後給我的感覺就是，code 真的不能亂寫 XD