hw3_109062320

109062320 朱季葳

Implementation

hw3-1

在這裡我使用Floyd Warshall algorithm，並連續的分配data給每一個thread，還有在每一回合結束之前都加上 `pthread_barrier_wait` 來確保正確性。

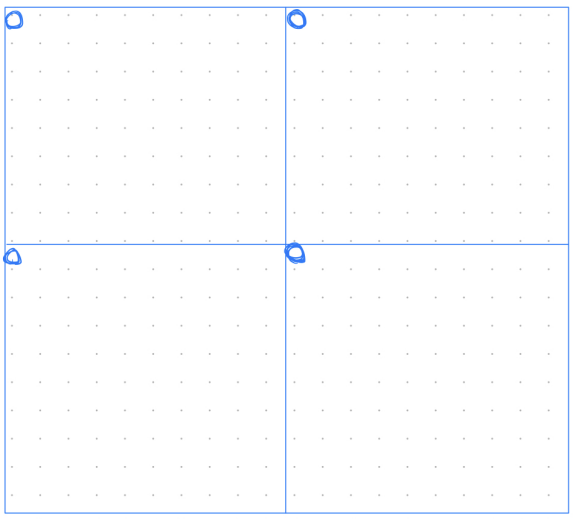
hw3-2 && hw3-3

- 注：因為hw3-2以及hw3-3最主要的差別就只有在hw3-3把phase3的部分拆成上下兩個part來做所以有額外進行處理，其餘的部分大致上都是一樣的所以放在一起做解釋。

在這裡基本上就是照著spec上面的圖去刻，因為要避免branching還有bank conflict，所以在一開始就把graph matrix進行padding，使row, column的數量為64的倍數，其中64為blocking factor的值(同時也是shared memory的array的column, row數量)。
而以下將一一解釋各個configuration的數值意義：

1. 把blocking factor設為64:因為在phase2, phase3的時候都會用到三個block，所以為了配合shared memory的size(49152)，我們可以透過簡單計算 `(49152/ 4(integer size)/ 3 = 4096, (4096)^(1/2) = 2^6 = 64)` 得到當blocking factor為64的時候可以完整使用shared memory的空間。

2. 每個block裡面都設置為32x32threads:因為每個block最多能有1024個thread，而每個thread都負責四個data，如下圖所示：



thread(0,0)負責block切成的四個小block裡面左上的那個data，同時因為一個warp為32個thread，所以在這裡可以ensure memory coalescing，在Optimization-hw3-2會有更仔細的說明。

3. Number of round : `(# of vertex after padding) / 64(aka blocking factor)` ，在這邊padding的部分因為上面已經解釋過所以就不再贅述了，而因為graph matrix是一個square, 再加上有經過padding，所以我們可以透過padding過後的graph的column or row數量除以blocking factor去得知我們需要幾個回合。

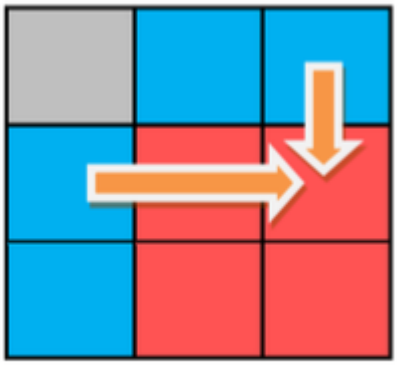
4. implementation of each phase

    - 因為在每個Phase裡面都有把global memory的data寫到share再寫回去的部分, 所以下面只會講述是處理了哪一些部分的data。
    - Phase1:在這個Phase裡面因為只需要進行pivot block的計算，所以就只有開一個shared memory array去進行計算，而這個部分基本上和普通的Floyd Warshall非常相似，而且因為會有data dependency的問題，所以要在每次回圈的最後都加上 `__syncthreads()` 來確保正確性。
    - Phase2:在這個phase是基於該pivot block, 還有同個column, 同個row的block去進行處理的部分，所以在這個部分每個grid是(# of round)個block所組成的(因為另一個是fix住的，所以只需要一個變因)，為的就是讓每個輪到的pivot block可以分別去更新fix住的column/row所對應到的block上的data。另外在這裡如果遇到輪到的blockIdx與當前pivot相同則return(不然會做額外的計算)。

      ```
      // 因為用講的有點抽象所以在這裡附上index的code
      // real index in d
      int idx_x = i + round * blocksize;
      int idx_y = j + round * blocksize;
      int idx_x_row = i + blockIdx.y * blocksize; //y is fixed, only x c
      int idx_y_col = j + blockIdx.y * blocksize; //x is fixed, only y c
      ```

    - Phase3:在這裡基本上是和Phase2反過來的，這裡是讓每個輪到的block都去與同column/row的block進行更新，所以我們需要每個grid是 `(# of round x # of round)` 才能確保每個block都輪到，而在這裡因為blockIdx.x, blockIdx.y與當前所輪到的round相同的時候就會return，所以不會有dependency的問題，因此可以把 `__syncthreads()` 拿掉。拿Spec上面的圖來說，當round是0的時候(灰色block)，如果blockIdx.x, blockIdx.y是0的話就return(如同塗藍色的那些

block)，然後去更新其餘的blocks，而其餘的block正是此時的pivot block。



(f) Phase 3

5. hw3-3：
    - Multi-thread programming:在這裡使用OpenMP並讓兩個thread分別對上兩個GPU來完成實作。
    - Data Allocation:在這裡因為Phase3佔了主要的計算量，並且為了減少溝通的時間(memory copy到對方那邊去)，所以在這裡只有Phase3有分工，而在其他Phase的部分兩個GPU都還是要算完全部的data。
    - Communication:在這裡我會在每個round要進行運算之前把這個round需要的那個row的data傳給對方(如果那個row是自己處理的話)，因為同樣可以從Phase3的圖看到，與自己相同row的block一定與自己在同一張GPU上面，但是和自己同個column的可能就會在另一張GPU上面，所以當這個狀況發生的時候就要同步更新一下data，而在這裡我使用的是DeviceToDevice的方式來進行Memory copy。
    - Synchronization:基於前面在communication所說明的，為了確保正確性，要先拿取不是自己處理的data，所以要在cudaMemcpy之後加上barrier來確保大家有同步在進行操作。

Profiling Results (hw3-2)

在這裡我使用p24k1這筆測資來進行profile, 並觀察loading最重的phase3 kernel

Metric Name	Min	Max	Avg
achieved_occupancy	0.946712	0.947737	0.947188
sm_efficiency	99.92%	99.97%	99.95%
shared_load_throughput	3305.3GB/s	3381.3GB/s	3366.2GB/s
shared_store_throughput	268.49GB/s	273.61GB/s	272.38GB/s
gld_throughput	18.534GB/s	18.754GB/s	18.572GB/s
gst_throughput	66.578GB/s	68.486GB/s	68.168GB/s

Experiment & Analysis

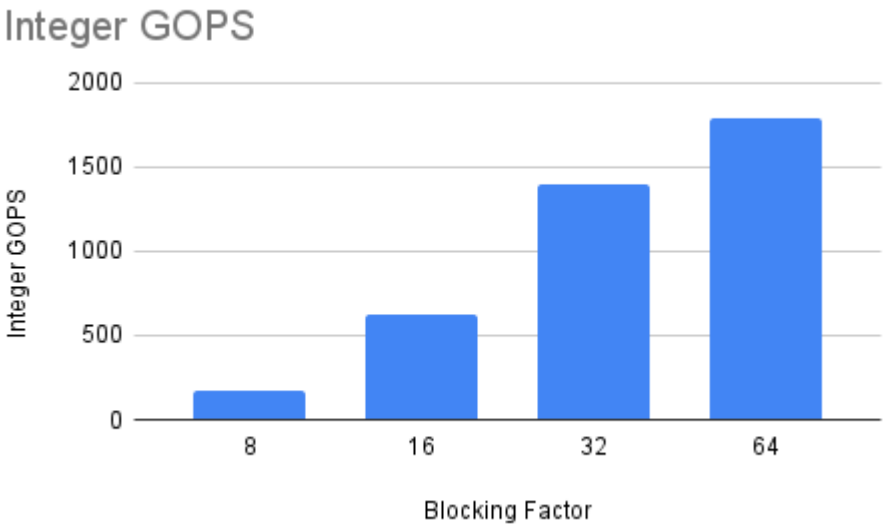
System Spec

課程提供的hades server

Blocking Factor (hw3-2)

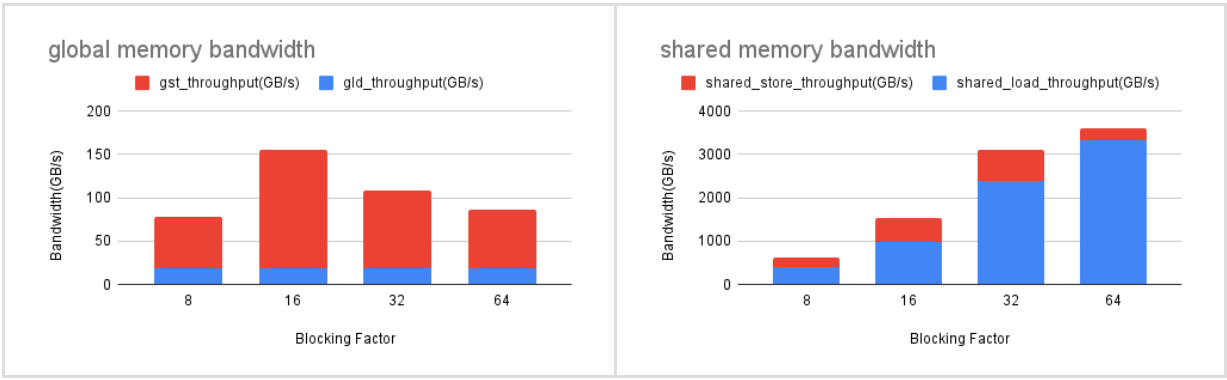
在這裡因為要避免profiling timeout, 所以使用c21.1這筆測資來進行profile，並同樣觀察loading 最重的phase3 kernel

- integer GOPS



在這裡我使用 `inst_integer` 這個metric去進行profile然後得到phase3的integer instruction count，並除去phase3的elapsed time來得到integer GOPS。在這裡可以看到Blocking Factor以及GOPs是呈現正相關的，而在這個部分我會在下面的部分與global/shared memory bandwidth一同做分析。

- global/shared memory bandwidth

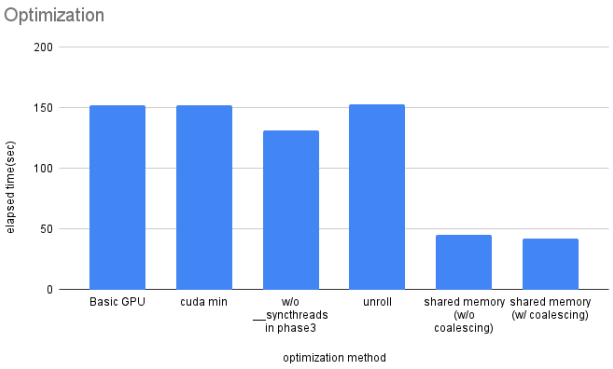


在這裡我分別透過 `gld_throughput`， `gst_throughput`， `shared_load_throughput`， `shared_store_throughput` 這些metrics去進行profile得到global memory, shared memory bandwidth(把load_throughput以及store_throughput相加)。而透過圖表我們可以進行以下分析。

- global memory bandwidth:在這裡因為load的部分可以說是幾乎沒有影響到所以只討論store。我們可以發現在blocking factor=16的時候，store達到最大值之後就開始隨著blocking factor的數值遞減，而針對這個部分我以下猜測。
  - 因為Blocking Factor越大的時候，要處理的data就越多，而在傳輸大量memory的時候blocks可能就會需要排隊，所以造成throughput下降。
  - 因為每個block需要處理的資料量變多，同時透過GOPs的實驗數據也可以看到當blocking factor的值上升，每秒的instruction count也隨之增加，所以就要花費比較多的時間在計算。因此相較於計算量較小的情況，每次store可能就會相隔更久的時間。
- shared memory bandwidth:在這裡因為在我的實作當中，每個shared memory的array都是 `# of blocking factor x # of blocking factor` 的2D array，所以在load的部分就會與blocking factor呈現正相關。至於store的部分我覺得和global memory bandwidth的部分很類似，因為隨著blocking factor的數量增加，每個block裡面我所分配的thread數量也隨之上升(因為總block的數量上升，而在我的實作裡面，每個thread都負責四筆資料，所以thread數量會與block數量呈正比)，因此在這裡就不再贅述了。

Optimization (hw3-2)

在這裡我使用p24k1這筆測資來對每項optimization所測量出來的elapsed time進行比較，而我在這裡使用clock_gettime()並在處理完input data後埋下start的timestamp，以及在輸出output前埋下end的timestamp，再將兩者做相減得到Elapsed time。



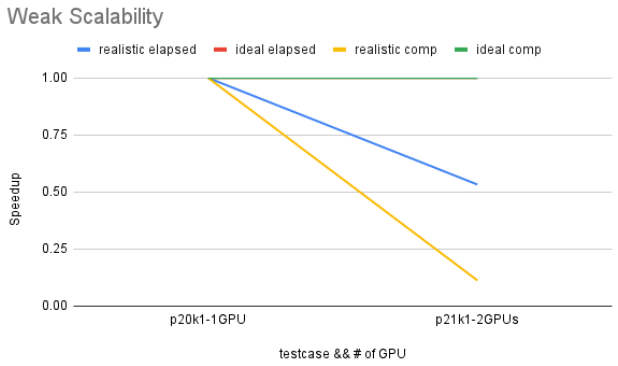
在optimization的部分我做了以下幾項優化，而我將針對他們一一進行說明，而這邊有一點執得注意的部分是我在Basic GPU的部分就有進行padding了，所以在這裡是基於已經解決掉邊界問題的實作來進行比較。

1. cuda min:在這裡因為如果有branching的話會有penalty，而在cuda有支援一些operation的API來幫助programmer減少branching的部分。但是在這裡我想可能是因為測資的關係所以才沒有看到很明顯的助益，不過在總體上來講還是對performance有一定提升的。
2. w/o `_syncthreads()` in phase3:在這裡因為phase3的計算量是最大的，而且在每個block裡的每個thread所depend on的data之間都沒有dependency的問題，所以在這裡把 `_syncthreads()` 拿掉也不會影響正確性，而依據實驗結果也可以看到這個方法是對於performance有助益的。

3. `unroll` :在這裡可以看到unroll對performance的助益其實很小，我想是可能是因為compiler已經對loop進行優化了，所以unroll可以帶來的助益就會很有限。
4. `shared memory(w/o coalescing)` :在這裡我加上了shared memory的實作，但是每個thread是負責連續的四筆data(e.g. thread0負責d[0], d[1], d[2], d[3])，而雖然因為存取shared memory的速度會比較快而使得performance有大幅上升，但是在這裡對global memory的存取是uncoalesed的，所以還是有優化的空間。
5. `shared memory(w/ coalescing)` :在這裡shared memory部分的實作跟最終版的一樣，每個thread負責該block所切成的四塊小block中的某個位置，而因為每個warp是32個thread為一組，所以可以確定每次存取的時候，每個warp所同時存取到的部分是連續的，而透過實驗出來的數據也驗證了這個方法對於performance是有助益的。

Weak scalability (hw3-3)

在這裡因為weak scalability是要觀測在每個processing unit上的工作量固定的情況之下所需要的計算時間，所以我挑選了Vertex數量分別為15000, 20959:p15k1, p21k1（`20959 ^ 2 / 15000 ^ 2 = 1.95` ）

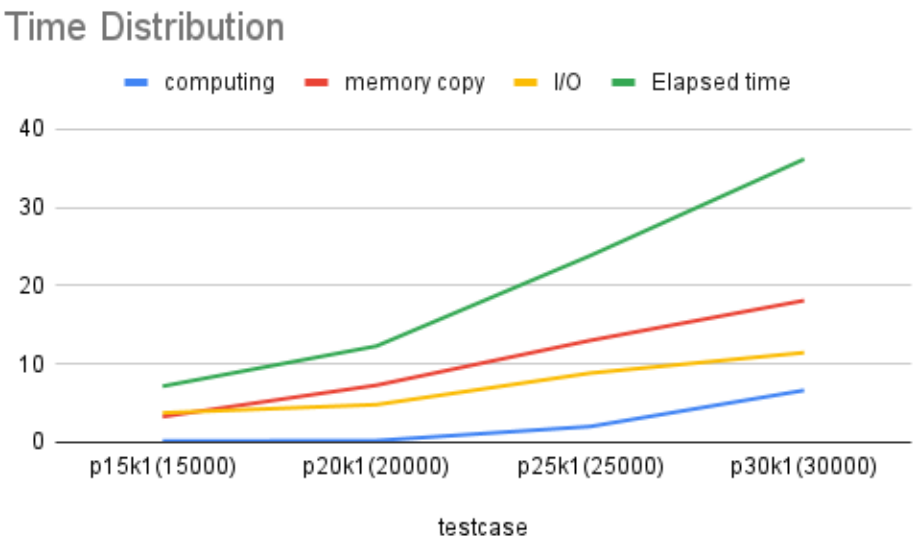


從上圖可以看到multi-GPU對於效能的提升其實沒有很明顯的助益，我想可能是因為在phase3的時候才分成兩個GPU各別去計算的關係，而在phase1, 2還是維持原樣，所以才導致效益不明顯,而在從elapsed time來看的話反而還比較scalable, 我想可能是因為把data寫回cpu的時候每個GPU都只要各寫一半就好了，所以消耗的時間就比較少。

Time Distribution (hw3-2)

以下圖表一樣是使用clock_gettime()去埋timestamp來算出的時間，算法如下：

- I/O time 為handle input/output的部分
- memory copy的部分為cudamemcpy所佔的時間
- elapsed time為handle input之前到handle output結束所佔的時間。



- 注1：在這裡因為GPU和CPU之間的溝通時間被算在memory copy那邊所以就沒有額外列出來。
- 注2：橫軸括號中的數字為vertex數量。

從上圖可以看到無論是哪個factor的時間都與vertex呈現正相關，而因為memory copy以及I/O time要優化都比較困難，所以這次只有針對computation time來進行優化。而可以透過詳細數據(在Appendix的Time Distribution experiment result)看到即使是在computatation time的speedup也非常有限，我想這可能是因為受到memory傳輸的bandwidth影響，正如同我在Blocking-Factor-hw3-2所提到的，當Blocking Factor上升時，雖然可以使得shared memory更加的fully utilized，但是在store的時候還是會受到影響，而當要處理的data增加，這樣的問題就會更加的明顯，所以這裡有很大的優化空間。

Experience & conclusion

What have you learned from this homework?

在這次作業當中我真的深深的體會到了GPU code的困難之處，而且寫法跟以往在寫CPU code的時候真的差很多，以前對於硬體Architecture對於performance的影響並沒有什麼感覺，但是在這次的作業當中真的可以看到很明顯的差距，然後也體會到了GPU code真的很難優化的事實QQ。

Appendix

Blocking Factor experiment result (https://docs.google.com/spreadsheets/d/1r_2vC3xey6X1bj2XS-1N4sNaT4-00Xixi3_ZEmSiE9Q/edit#gid=0)
Optimization experiment result
(https://docs.google.com/spreadsheets/d/16jpyruK9OHMQpYwrfWZCxz9QY1lPtHpFvyZdNv5ueik/edit?usp=sharing)
Weak Scalability experiment result
(https://docs.google.com/spreadsheets/d/1441_aMA9_S_UKmdzV_Okuo5xHoaq55DDzBrpfFpxRTY/edit?

usp=sharing)

Time Distribution experiment result (https://docs.google.com/spreadsheets/d/1jVK_-vk3rCiwKAveE3E7hheXxaSg4EAykwfazEtOhAE/edit?usp=sharing)