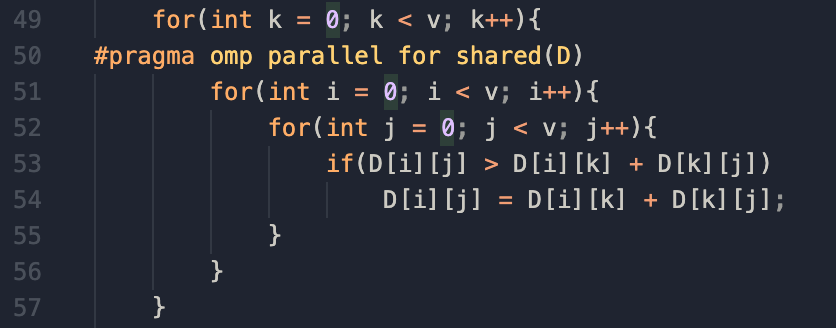
CS542200 Parallel Programming

Homework 3: All-Pairs Shortest Path

107062103 王依婷

1. **Implementation**
2. **Algorithm of hw3-1**

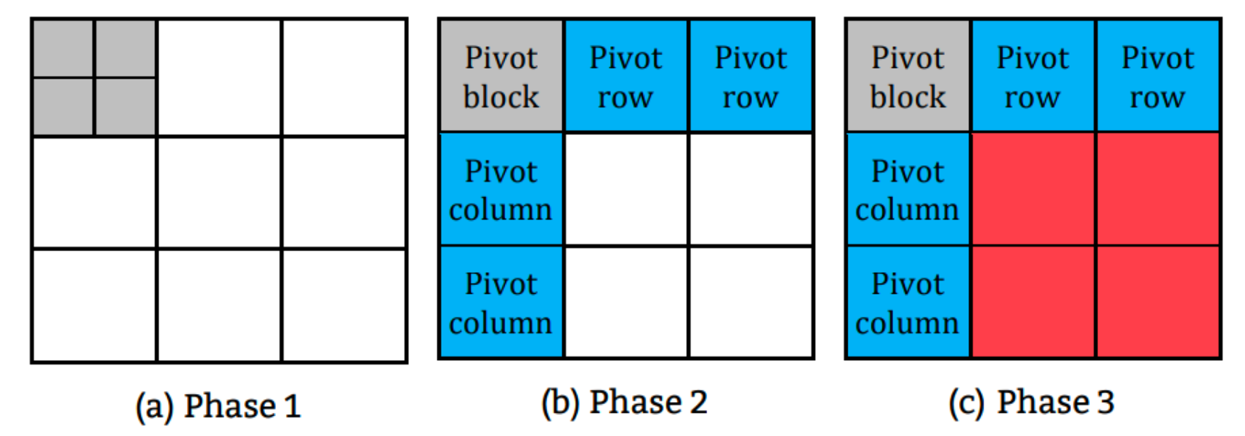
在hw3-1部分我使用的是基本的Floyd-Warshall algorithm，再加上OpenMp來做平行。因為最外層的k for-loop每個迴圈之間會有data dependency，因此只對裡面兩層for loop進行平行化。



1. **Division of data in hw3-2, hw3-3**

在hw3-2及hw3-3中，使用的是spec中介紹的blocked Floyd-Warshall algorithm，會將整個尋找最短路徑的過程分成一塊一塊的block來進行，根據phase的不同，需要進行計算的block數量不同，也會需要不同block的值來對當下的block做計算。

我讓每個GPU中的thread block的大小對到一個block，也就是一個thread一次會負責處理一個vertex。在每一輪中計算中，會依序選定左上到右下對角線上的block作為該輪的pivot block。在Phase1時只需要計算Pivot block，不需其他block的數值；Phase2時要計算和Pivot同一行及列的其他block，需要自己以及在Phase1算好的Pivot block；而Phases3時則計算剩餘的block，需要Phase2中對應到自己row index和column index的兩個block。

* **hw3-2**

Single GPU有整個distance matrix，只要在計算block時拿取所需對應block的數值就可以了。

* **hw3-3**

hw3-3有兩個GPU，在計算量最大的Phase3時，我會將所有block分配到兩個GPU上計算，但要注意的是，若原本的block數量（也就是round）是奇數，為了計算時取index方便以及正確性，會指定由第二個GPU多拿一個block。

1. **Configuration in hw3-2, hw3-3**

* **hw3-2**

GPU block數量和thread數量的部分，對應上一點divide data的內容，在Phase1時只要計算一個Pivot block；Phase2時共有round \*2-1個block（round為在行/列方向上block的數量） ，為了計算時取index方便直接開2\*round個blocks；而Phase3要計算(round-1)\*(round-1)個block，一樣為了方便直接開round \* round個blocks。每個block中thread的數量也一樣對照到algorithm中每個block裡面的vertex數量，為B\*B。

至於blocking factor，希望一個cuda GPU中的block能夠剛好處理一個計算時的block，使用deviceOuery.cpp查閱了細節後（+chap7講義）得知一個block最多能有1024個threads，因此將blocking factor設成32，讓block能放滿，裡面有32\*32=1024個thread。

|  |  |  |  |
| --- | --- | --- | --- |
|  | Phase 1 | Phase 2 | Phase 3 |
| blocking factor (B) | 32 | | |
| # of blocks | (1, 1) | (2, round) | (round, round) |
| # of threads | (B, B) | | |

* **hw3-3**

blocking factor同上。

Phase1和2計算量較小，且為了讓Phase3兩個GPU各算一半的時候能夠直接開始，不需要再做data copy，我讓它們也都計算Phase1和Phase2的所有block，所以GPU block數量和hw3-2相同。Phase3時也是對應到divide data所說，兩個GPU各有split \* round個blocks。

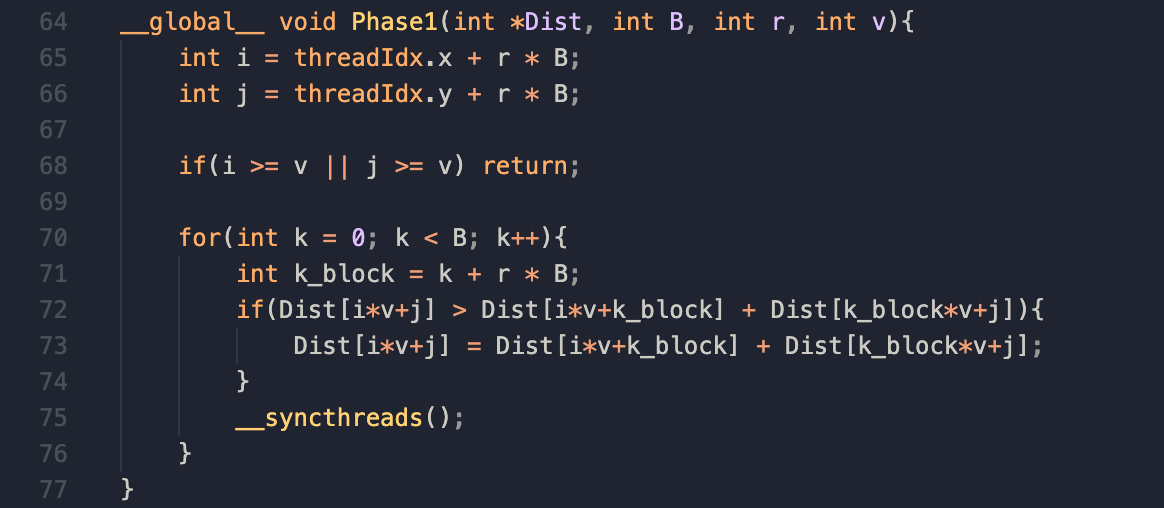
|  |  |  |  |
| --- | --- | --- | --- |
|  | Phase 1 | Phase 2 | Phase 3 |
| blocking factor (B) | 32 | | |
| # of blocks | (1, 1) | (2, round) | (round\_split, round) |
| # of threads | (B, B) | | |

1. **e. Implementation & Communication of hw3-3**

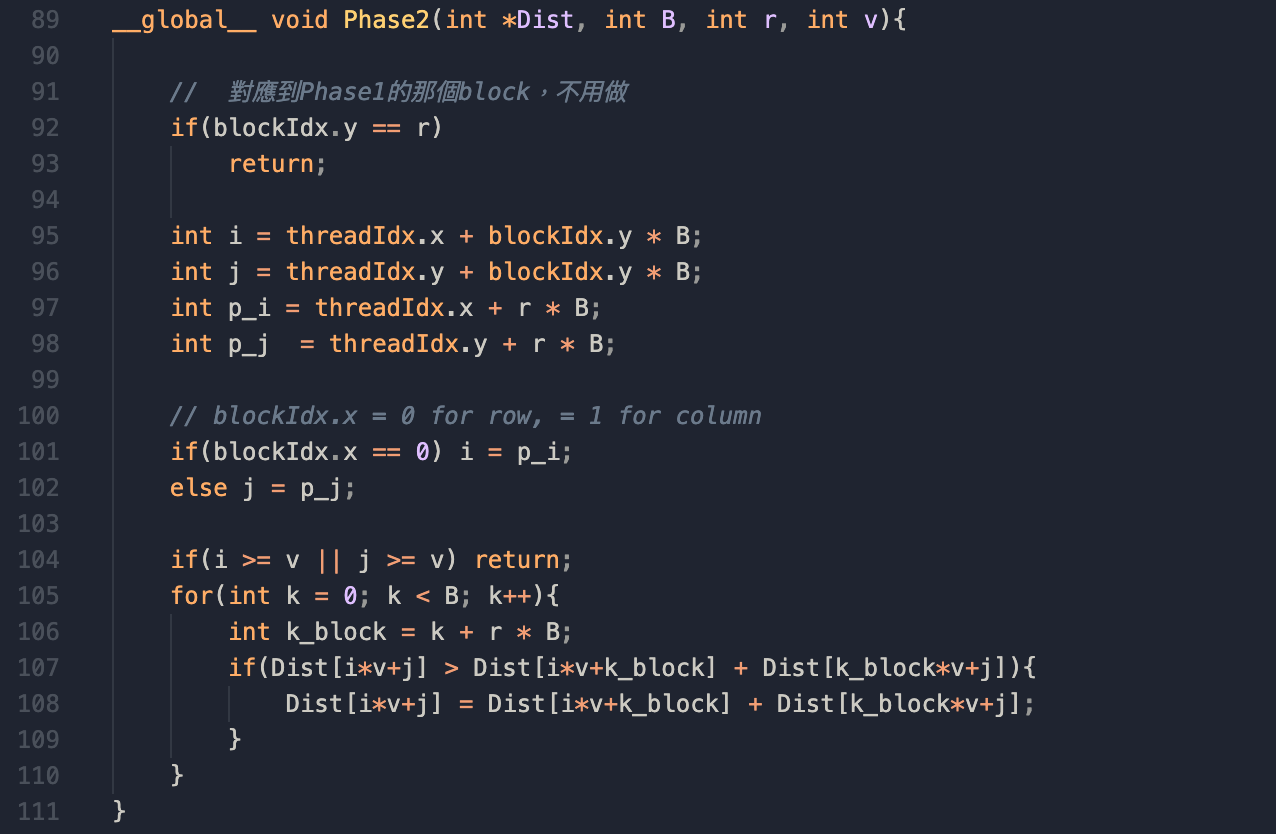
* **hw3-2**

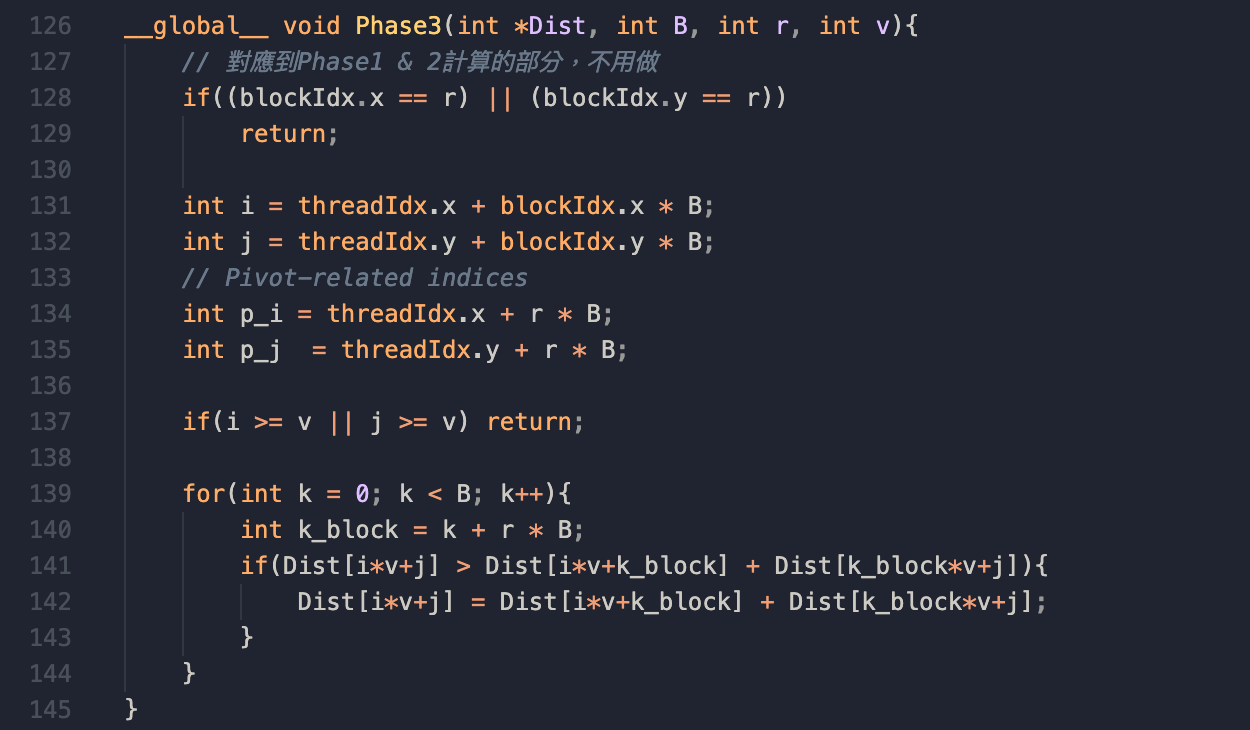
按照作業spec上的block Floyd-Warshall algorithm方法實作（以下所附圖片都是baseline方法的程式碼），讀進資料時存成一為的陣列D，使用cudaMemcpy把Ｄ複製到device上，接著為Phase1、2、3各別實作不同的kernel function來計算。因為case中的vertex數量有可能不能整除於設定好的blocking factor，那這樣在以block為單位計算的時候會遇到取到超過陣列範圍的狀況，所以在一開始讀進vertex數量時，就會計算出若要湊成B的倍數，vertex還需要再padding多少，並將資料陣列開成Ｂ的倍數，多的padding部份補成230 – 1，讓計算能夠順利進行，如以下程式碼所示。



Phase1計算一個Pivot block，根據目前的輪次r找到thread對應到的block中數字，跑Floyd-Warshall的三層for迴圈計算。而因為不同k之間所計算的值有dependency，所以要加上\_\_syncthreads()。以下是Phase1部分的code。

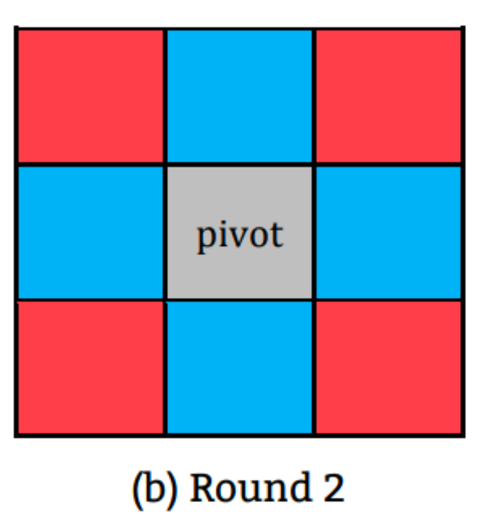
Phase2計算和Pivot block同row或是同column的blocks。我設定當blockIdx.x=0時計算row方向的blocks，=1時計算column方向的，根據這兩種情況找到對應的Pivot block index，取值來計算來計算。以下是Phase2部分的code。

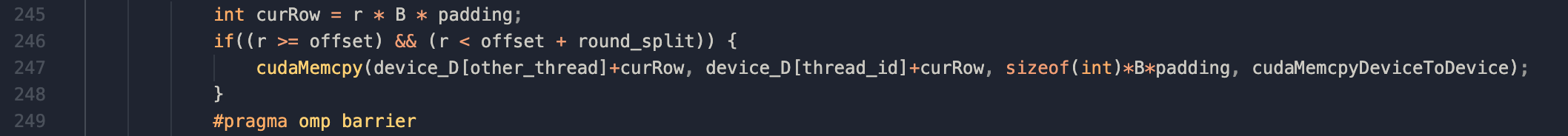


最後Phase3的部分計算所有剩餘的block，一樣會按照自己的blockIdx和threadIdx取得現在要計算的數值位置、還有其他所需要的、Phase2計算好的數值。

* **hw3-3**

在hw3-3中，按照設定總共使用兩個core、兩個GPU 來實作multi-GPU，也就是一個core 對到一個GPU，而我也是使用OpenMP來做平行。



在Phase1和2時兩個thread都會呼叫相同的kernel function，做一樣的計算，這邊都和hw3-2相同。Phase3如同上面所說，將所有要計算的block分成一半來處理，為了下一輪的正確性，兩個GPU之間需要做一些data的複製及溝通。如左圖所示，橘線上方交給device0，下半由device1負責，經過第一輪計算後，他們會各自擁有一半的正確計算結果，當進到第二輪Pivot block移動到device1那半的位置，若在Phase3時device0要計算右上block的話，需要device1所計算好的row方向的block，所以需要把他們複製到device0中。統整之後可以發現，所需要的column方向block上一輪已經計算完畢，不需要複製；只有row方向的block會因為Pivot block移動到了另一個device那半的位置，而要做複製的動作。

1. **Profile Results (hw3-2)**

我使用了c20.1來做profiling，取最大的Phase3 kernel，下表格為結果：

* **occupancy**：在一個SM中active warp和最大active warp數量的比值，這邊使用nvprof中的achieved\_occupancy，它是取所有SM在任何時刻的比值平均值。
* **SM efficiency**：平均在GPU的所有multiprocessor中，至少有一個warp是active的比例。用sm\_efficiency來查詢。

|  |  |  |  |
| --- | --- | --- | --- |
|  | Min | Max | Avg |
| occupancy | 0.894175 | 0.897251 | 0.896442 |
| SM efficiency | 99.74% | 99.82% | 99.77% |
| shared memory load throughput | 2696.6GB/s | 3048.8GB/s | 3029.9GB/s |
| shared memory store throughput | 112.36GB/s | 127.04GB/s | 126.24GB/s |
| global memory load throughput | 168.54GB/s | 190.55GB/s | 189.37GB/s |
| global memory store throughput | 56.179GB/s | 63.518GB/s | 63.122GB/s |

可以看到因為做了使用shared memory的優化，他的throughput比global memory的高出許多。而在這次作業較常需要把資料從memory中取出來做比較，值較大才做更新，所以load throughput都大於store throughput。

1. **Experiment & Analysis**

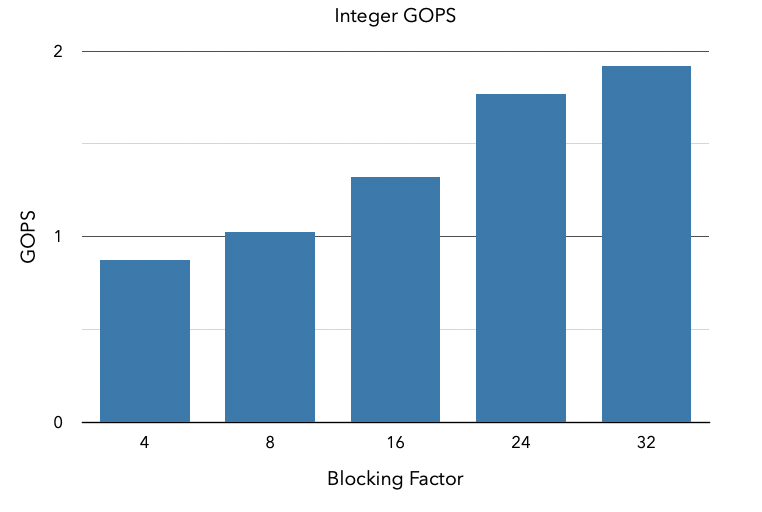
Blocking factor和optimization兩部分也使用c20.1來測量；time distribution再加上其他三個不同input size的case來做。

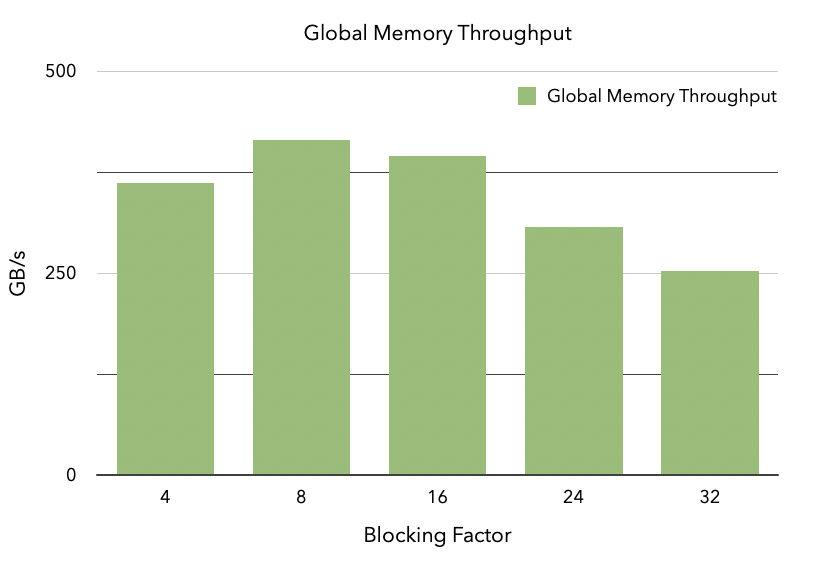
1. **System spec**

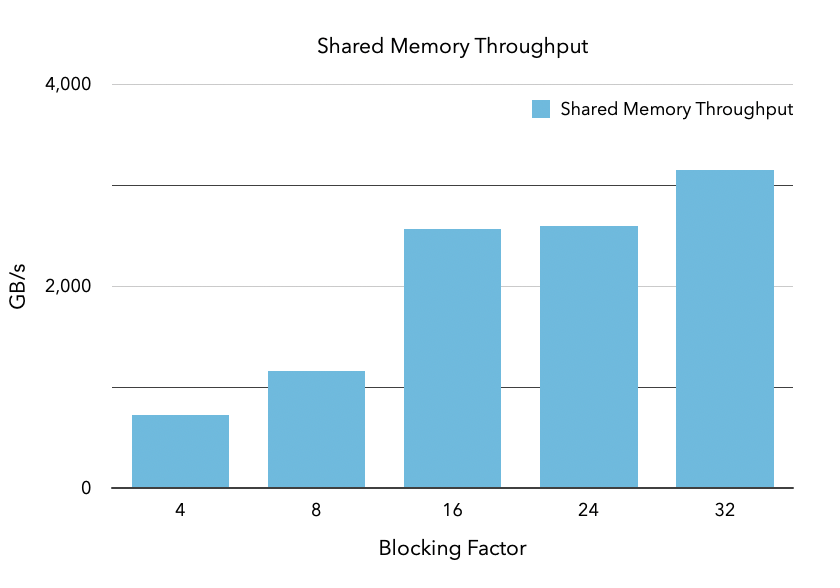
使用課程所提供的hades，沒有用額外的設備。

1. **Blocking factor (hw3-2)**

以下是調整不同blocking factor所測得的數據柱狀圖，：

使用inst\_integer來取得數據。從integer GOPS的數據中可以看出，隨著blocking factor的增加，執行的指令數量也增加，因為使用了更多的thread，也能夠一次取出更大量的資料來運用、計算，增加效率。

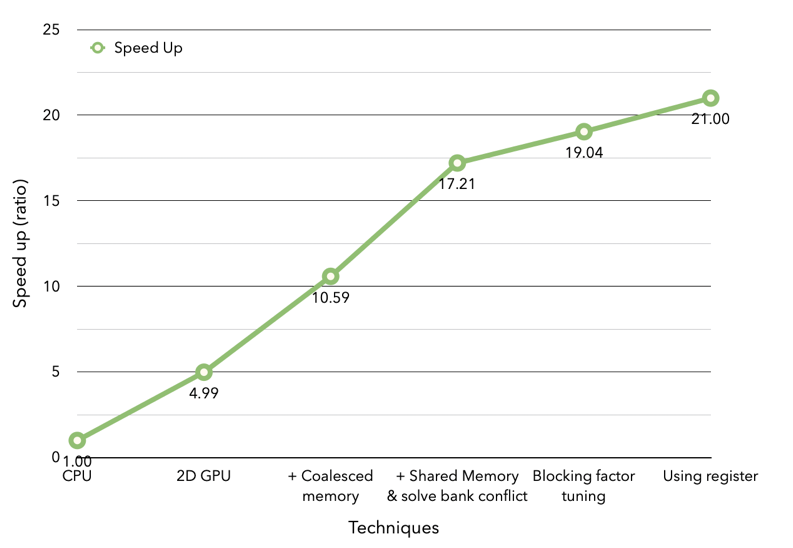
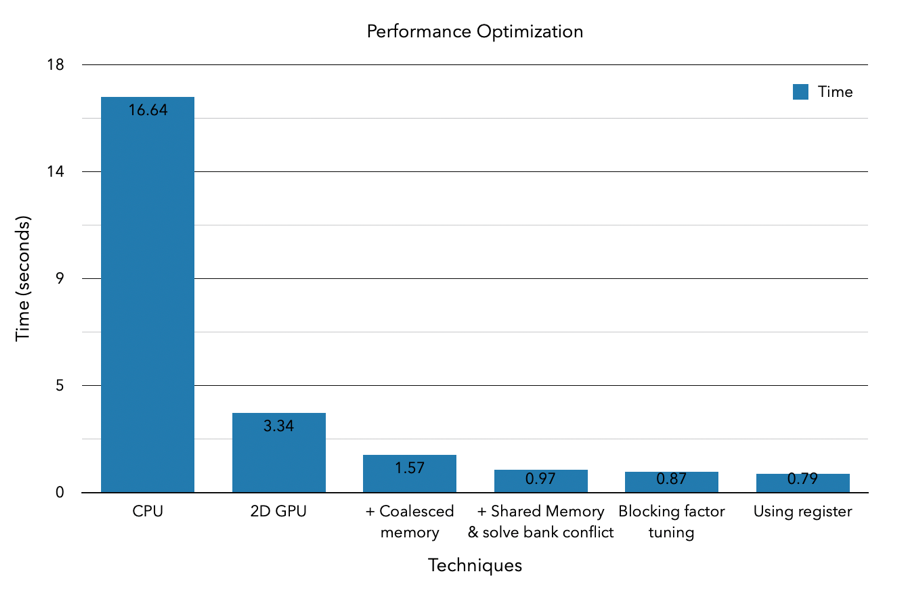
這邊我是使用throughput，也就是實際上有效的傳輸量作為評量。使用了shared memory後，減少了將access global memory的次數，只要先把global memory裡面的資料搬到shared memory就能作後續的計算工作。當blocking factor變大，這個搬移資料的次數也變少了，所以數值便下降。



從Shared memory的結果中也能看出，因為blocking factor變大，一次要搬移到shared memory的資料也變多了，因此數值隨著blocking factor變大而增加。

1. **Optimization (hw3-2)**

這邊主要先以整體時間上的減少來做比較，各部分所花的時間會展示在下面的time contribution部分。以下是各方法的執行時間和speedup，並且一一說明優化的方式：



* **2D GPU**

使用2D grid和2D block，block的大小會根據blocking factor而定。

* **Coalesced memory access**

為了能讓同一個warp中所有thread的data可以連續，一起被access，GPU thread執行的順序要和data存取的順序一致。所以我讓threadIdx.y是row方向，也就是同個warp中threadIdx.y都相同（最後blocking factor最大tune到32，代表一個block裡面的thread有32\*32個，也還是可以剛好可以fit）。

* **Shared memory**

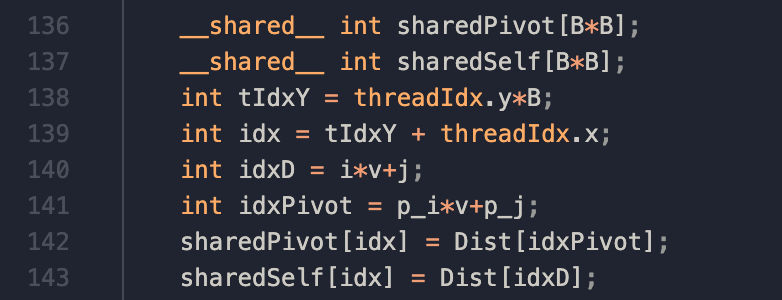
在每個phase中，都會先將計算時需要用到的相關block放到shared memory：Phase1只需要正在計算的block；Phase2還需要Pivot block，因此也load到shared memory中；Phase3需要多load Pivot row和Pivot column。這些所需要的block內數值，會由正在計算的block中每個thread幫忙load相同index位置。最後用\_\_syncthreads()來將block內thread的行爲做同步化，確保所有需要的值在計算前都到位。這時的threadIdx.x和threadIdx.y的順序也和上面相同，同時也避免了shared memory中bank conflict的可能。

* **Blocking factor tuning**

Configuration的部分有提到，因為一個block中最多能有1024個thread，因為我是以blocking factor的大小來設定GPU的block size，所以將blocking factor tune到32，讓thread可以開到最滿。

* **Using registers**

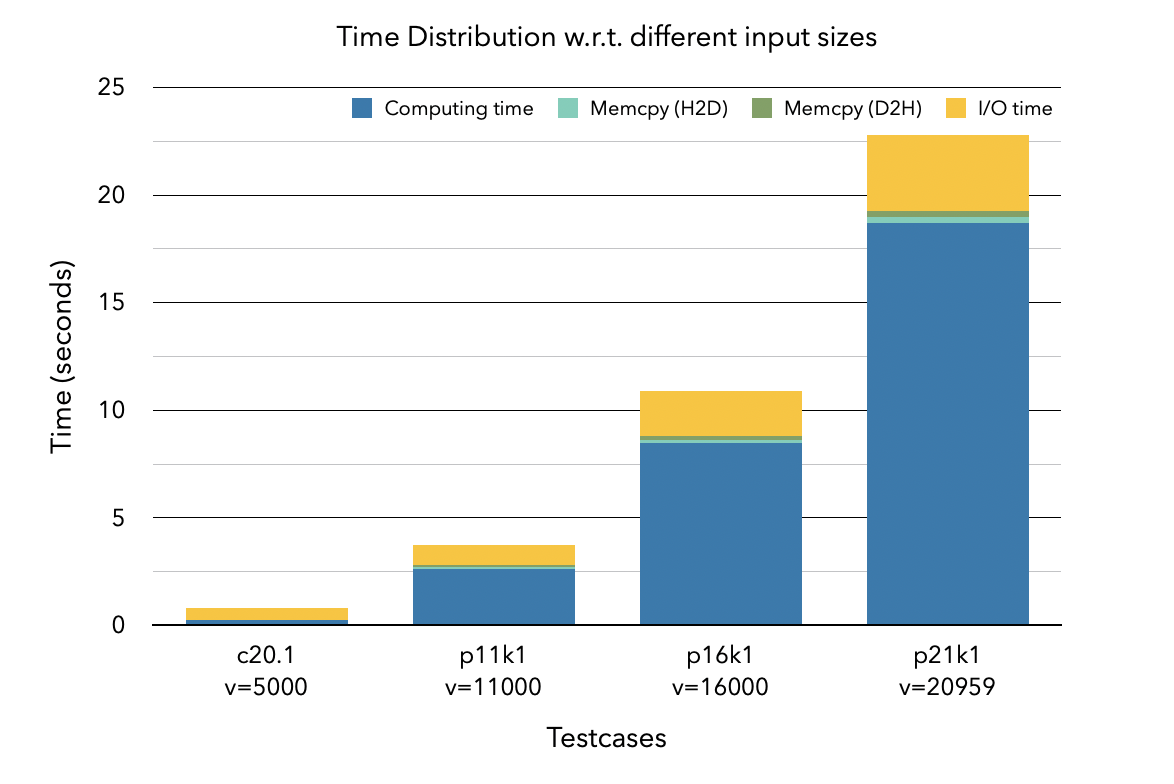
將過程中會重複計算到的數值，如各個block中的index，都用速度最快的register暫存起來，就不需要一直做加減乘除。



1. **Weak scalability (hw3-3)**

在Phase3時會將資料分成一半給兩個GPU來計算，因此兩個GPU之間還需要做溝通以及同步化，memory access也有可能變得不連續。最後兩個GPU做完一輪後都必須同步，保持整個matrix的狀態一致正確性。老師上課時有提到multi-GPU以我們目前學到的部分，加速的效果不好、很難，不過還是能夠優化，不過在這個作業中就無法做到。

1. **Time distribution (hw3-2)、Others**

****

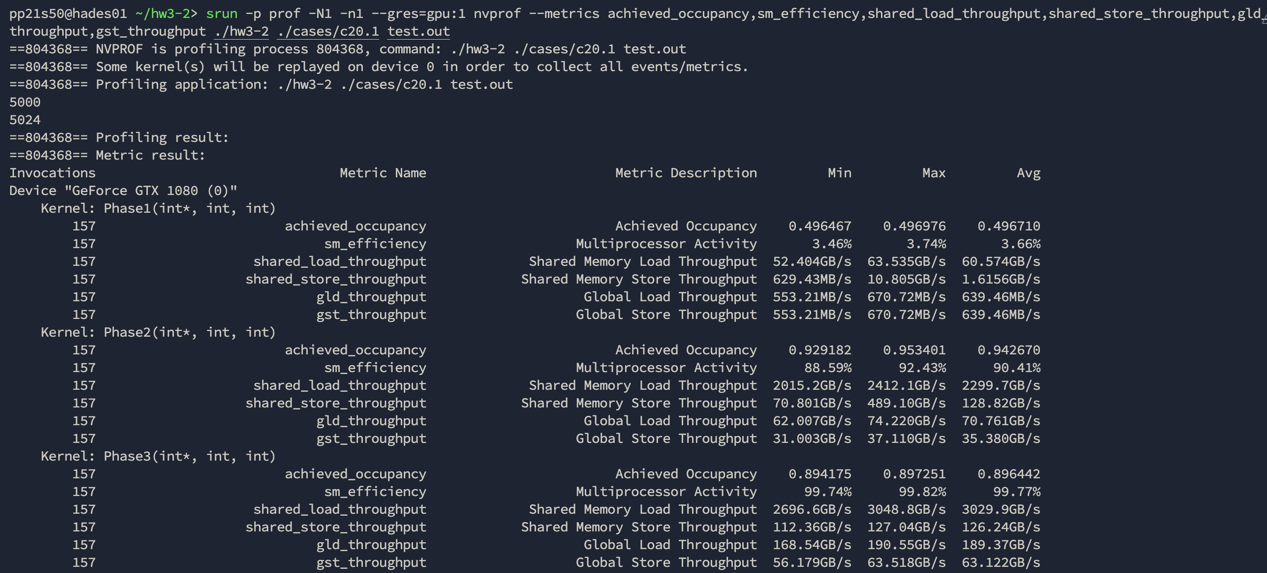
我挑選了size差異儘量差不多的一些testcase來做實驗。在size較小的case中，其實I/O所花的時間會比計算還要來得多一些，但到了較大的case，雖然I/O時間也隨之增加，但都會是計算最耗時。而不管Ｍemcpy是HostToDevice還是DeviceToHost，所花的時間差不多且並不會影響整體的performance太多，不會是問題。

在這次的作業中，其實還有更多優化的方法，但我沒有實作到那麼多，所以可以看到雖然每個testcase的size差距差不多，在計算上所多花的時間卻是越來越多，最明顯的就是從c20.1到p11k1之間的差異，vertex數量是變兩倍沒錯，但計算的時間遠超過兩倍，還有很多可以優化的空間。另外，主要都是針對比較最短路徑時的計算來做優化、改善，像是I/O的部分沒有特別更改，尤其是在size較小的case中，也造成了一定的時間花費。

1. **Conclusion**

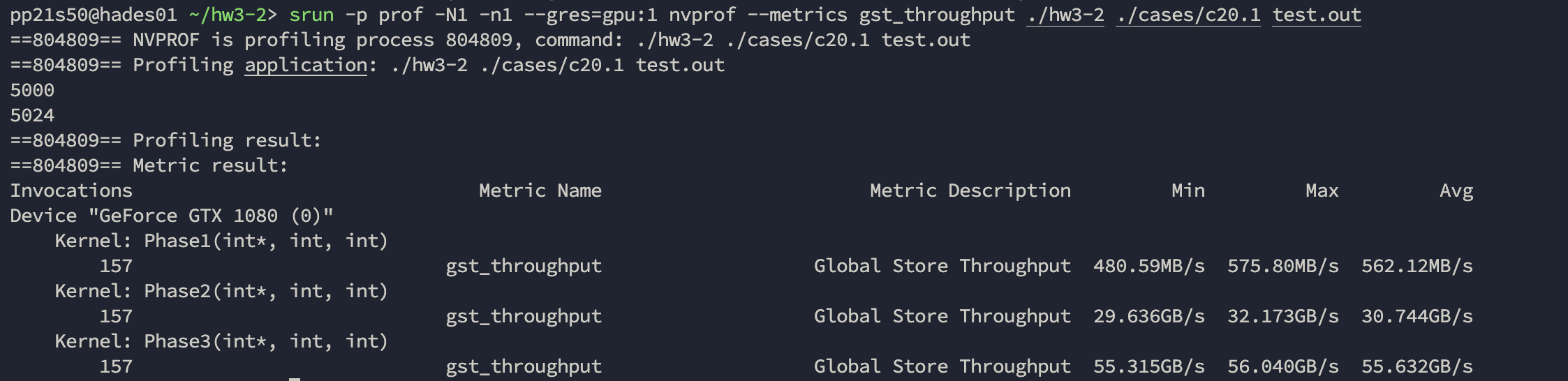
這次作業要實作的部分更多了，優化的方法也因為加入了GPU變得更加多元且powerful，最讓我驚奇的是coalesced memory的部分，程式上只是改個執行的方向，在實際memory access時卻有很大的差別，節省了不少時間！

另外，在做作業時遇到了一個小疑問，在做profiling時，當我使用--metrics [任一metric]指令，和--metrics [多個metrics]時，在同一metric上所得出的結果會是不同的，甚至差蠻多的，讓我十分疑惑！狀況如下圖所示，可以看到global store throughput的結果差異。



▲ 同時使用多個metrics

▲ 單獨使用gst\_throughput的狀況



1. **Reference**

* https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference