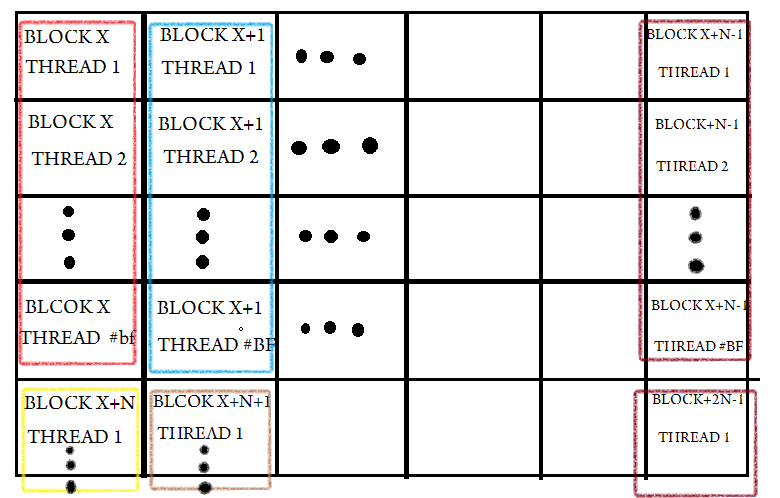
Parallel Programming HW4 Report

102062209 邱政凱

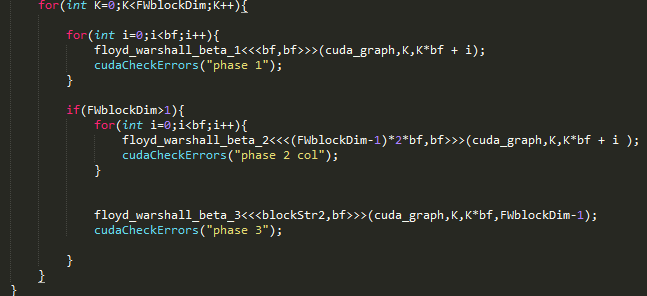
Implementation

實作Cuda的第一個要件就在於如何將Data之間區分成彼此不互相Dependent的單位，充分利用GPU的效能。在這次作業中，我們要計算的是一個N\*N的Shortest path graph，因此我基本的概念就是讓每個GPU裡的thread負責更新一個對應的(i,j)距離。然而cuda最高的thread上限數是1024，因此我們勢必要切割出許多的BLOCK。最直覺的做法應該是讓每一個Blocked APSP algorithm裡面的block對應到一個cuda的block，然而在Blocked APSP裡面，同一個tile block在同個phase裏會需要使用到相依或相同的資料，因此這種作法只支援Block factor <=32的實作方法(32\*32 = 1024,單一block thread數上限)。因此我使用了另一種方法：(前面那種方法我仍然有實作，並在後面的optimize section會分析實驗結果)



我的作法是將每個block都設定跟BF一樣的Thread數，接著如上圖一樣，block id隨著column id增加而增加，thread id隨著row id增加而增加，而每BF個ROW之後又是另一排新的Block。這種作法在大測資的狀況下會需要多維度的block dim3，因為cuda的kernel block數單一維度上限約莫六萬，不過就算放入多維度的block dim3，我仍然會用轉址的方式把其視為單一維度來處理。

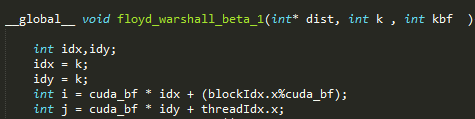
而在CPU的部分，我將三個Phase分別分成三個Kernel來Launch，原因就在於三個Phase需要的block數不同，轉址的方式也不同，data dependency也不同。

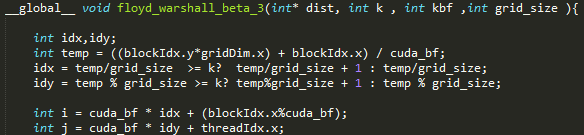
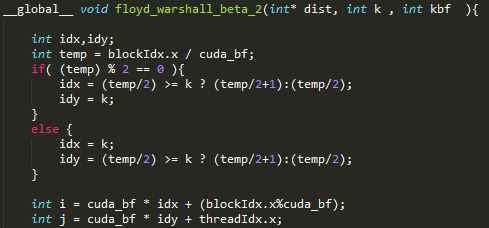


可以看到，我在Phase 1或Phase 2裏，會分成BF數個階段來Launch kernel，這部分是因為在Phase1和2中，每個iteration都會depend on上一Iteration自身block的計算結果，而因為跨block的thread間無法synchronize，因此藉由launch新的kernel確保所有的thread抓到的資料都是最新的。

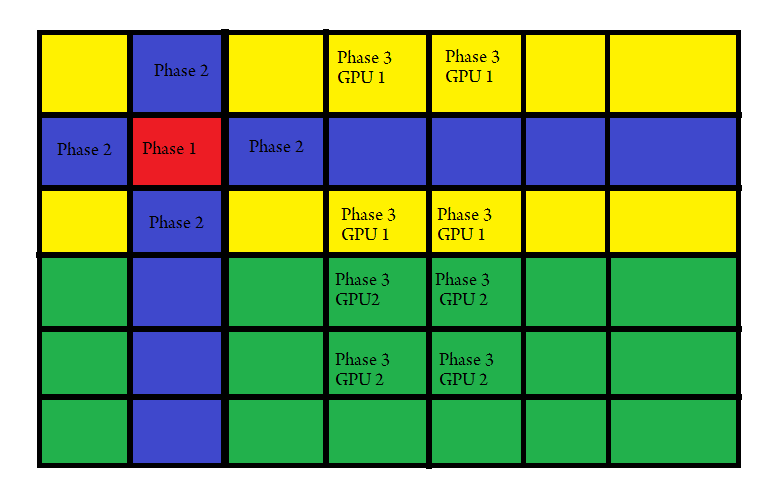
而Phase 3因為每次iteration都不會相依於上一輪自身計算的結果，而只相依於phase 2中的結果，因此可以直接用一個kernel launch，在裏面直接進行K個iteration，而不用擔心Synchronization的問題。

至於關鍵的轉址方式簡單附上如下，大致上就是確保所有block的所有thread不會計算到相同的(I,j)。





在Multi GPU的版本中，我實作的是可以針對任意數量的GPU做計算分配的演算法。因為考量到Communication和Memcpy的overhead，我只針對佔據大部分計算量的Phase3做跨GPU的計算。也就是說，所有GPU都在每個回合中自行計算完整的Phase1和Phase2，然後將Phase 3的工作量均分。



以兩顆GPU的狀況為例，如上圖的graph所示，如果目前是K=1，也就是整個Blocked APSP algorithm的第二個iteration，那上圖紅色和藍色的部分就是Phase 1/2要計算的block，而這是所有GPU都要計算的部分。接著在Phase 3，我會用row index當作參考，將扣除Phase1/2的block後剩下的blocks中所有的row分成(#GPU)個Section，如果有無法整除的餘數Index的row都丟給ID最大的GPU計算。

計算個別完成之後，在openmp的版本中我使用cudaMemcpy(…,Device to Device)的設定，直接交換所有GPU裡面的資料。以上圖為例，GPU1就是把自己的黃色部分的BLOCK的距離計算結果傳給GPU2，而GPU2則把自己的綠色部分傳給GPU1。交換完彼此的資料之後，設定K=K+1，繼續Blocked APSP的下個Iteration。在MPI的版本中則是每個Process先用cudaMemcpy(…,Device to Host)把自己計算的部分copy回CPU，接著用MPI\_Send傳給其他所有的GPU，並同時接收所有其他GPU送來的計算結果，每個Process再個別用cudaMemcpy(…,Host to Device)把其他GPU的結果更新到對應於自己的GPU中。

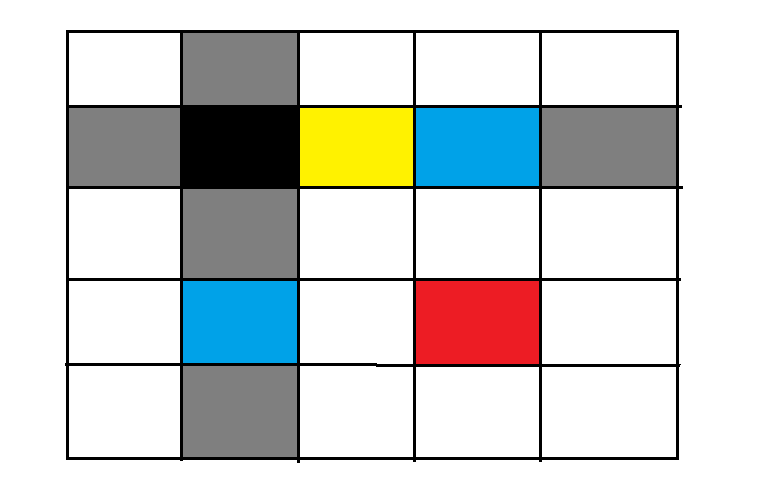
最後一點就是，因為graph的節點數不一定能被Block Factor整除，因此我的作法是把多補上一些不可能走道的假節點值到N可以被BF整除為止。

整體而言，這種演算法的好處在於Block Factor數大的時候仍可以充分利用GPU資源，經實驗結果會發現在BF = 256時會達到最好的效果。然而缺點是，因為每個BLOCK中只會有BF數個Thread，所以如果BF數較小時可能會有太多的Block，造成GPU資源硬體的Context switch的overhead太多。

Optimization Techniques:

以下則是我另外實作的，加入Optimization Techniques的版本。因為為了使用Shared Memory，這部分的實作是跟上面使用不同的架構。這部分我的架構設計使得Block Factor一定要小於等於32才行。然而後續測試發現，跟原本的架構比起來，在Block Factor <=32時確實會有效能的改善，但是在原本的版本中的BF=256時的效能又比這種優化版本的效能還好(但是這個版本BF數上限是32 QQ)。因此後面的實驗章節我使用的還是以上面的架構為主。(不過在相同的Configuration(Block factor)時這種架構速度是有超過上面的原本架構，所以還是有優化成功的)。

Shared Memory:

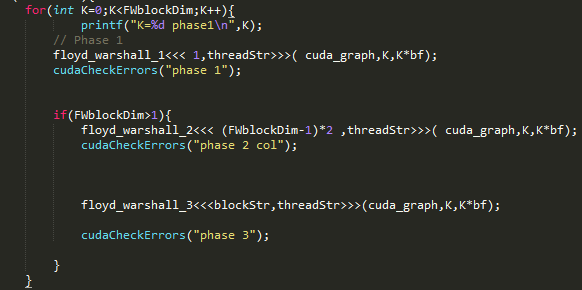


圖(一)

使用shared memory的關鍵在於把彼此會使用相同資料的節點(以及對應的threads)放到同一個block中，讓它們可以共享資源。因此針對Blocked APSP演算法，三個PHASE各自有不同的方式。且因為單一block的thread數上限是1024，所以block的大小最多只能是32\*32。

想像上面的圖每一格APSP的BLOCK都是剛好對應到一個Cuda block，而裏面都各自有32\*32個thread，而每個thread各自都負責更新一個對應的(I,j)距離。而因為shared memory的Life time跟block一樣，因此我們不能像前面一樣每個iteration都重新launch一個kernel(在這個版本中也沒必要，因為同一個APSP Block的點都在同一個cuda block之中，因此可以使用syncthreads())。

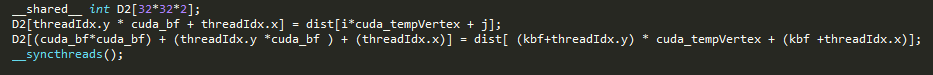
下圖就是shared memory架構launch kernel的方式，跟原本的架構比起來，phase 1跟phase 2並不會分成BF次來LAUNCH，而是只launch一次。



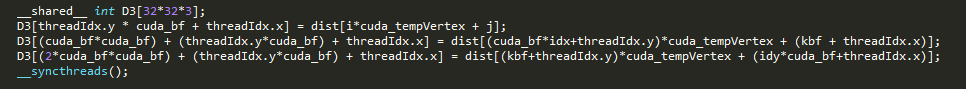
在Phase 1中，每次更新只會使用到同屬同一個APSP block的距離，因此在Phase 1中，block中每個thread都各自去Global memory抓對應於自己(I,j)的那個距離放在shared memory中。在Phase 1的距離更新也就直接寫到shared memory裏，離開kernel的時候再寫回global memory就行了。(以下三段code，cuda\_bf代表block factor，cuda\_tempVertex代表總節點數，I,j的計算方法就跟原本的架構一樣)



在Phase 2中，假設以圖(一)為例，黑色的block是phase 1時更新的block，而黃色的block則是目前要更新的，那麼同屬這個block的每個thread就個別去黑色的block和黃色的block各自抓取一個對應的距離放到shared memory裏來使用和更新。(每個thread負責兩個值)



在Phase 3中，以圖(一)為例，如果當前要更新的block是紅色的block，則每個thread就分別去兩個藍色的block及紅色的block中抓取對應的距離放進shared memory中來使用和更新。(每個thread負責三個值)。

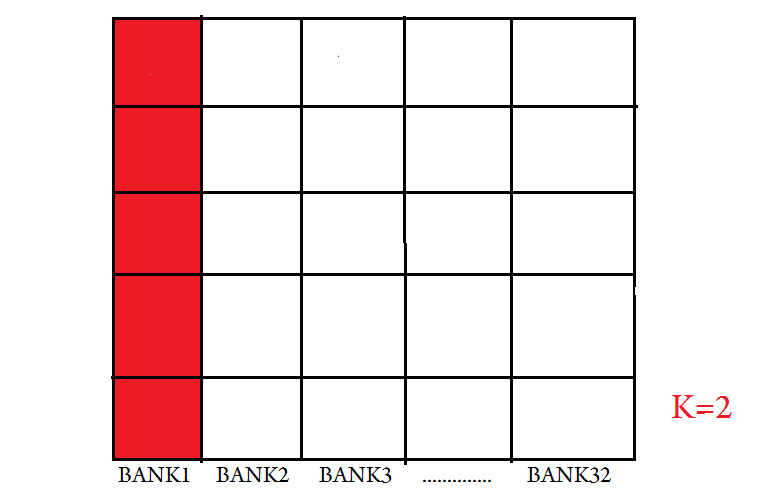


Bank Conflict Avoidance

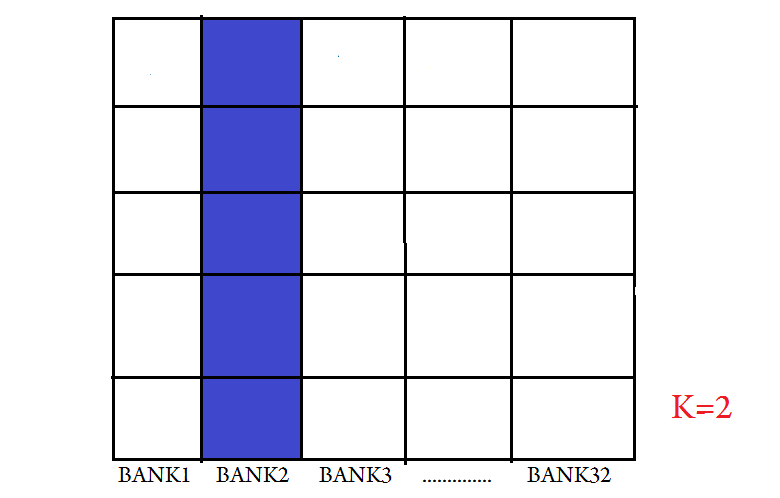
使用Shared Memory的另一個關鍵就在於要避免bank conflict。Bank conflict發生於同一個warp的threads存取同一個bank不同位址的資料。而在APSP演算法中，每回合我們需要去抓三個shared memory裡面的值：Dij,Dik,Dkj。以我的狀況來說，因為我一開始的Indexing方式如下：



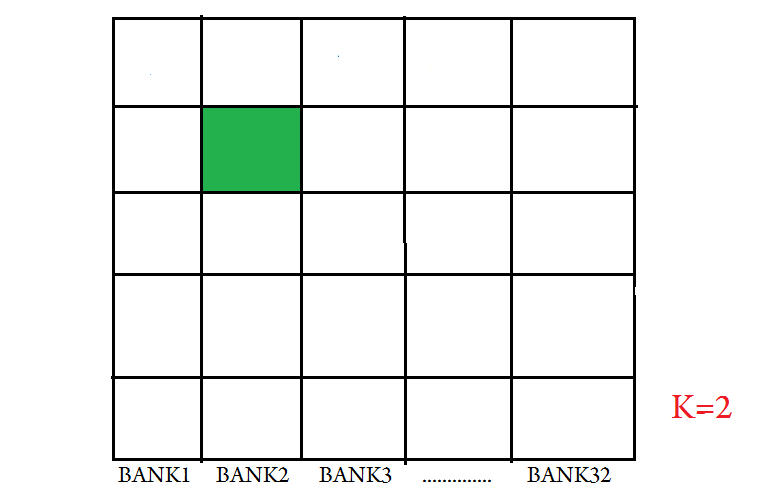
因為cuda裏用2d dim的結構來做thread的indexing時，似乎連續的threadIdx.x會被當成連續的thread而被排在同一個warp中。而這樣子的indexing方式會導致我的某個warp在shared memory裡的取值變成如下 (shared memory為32\*32的2D int array，而在我測試的gpu中bank數為32，i是row index，j是column index，假設目前是K=2的回合 ) :



Dij 取值



Dik取值

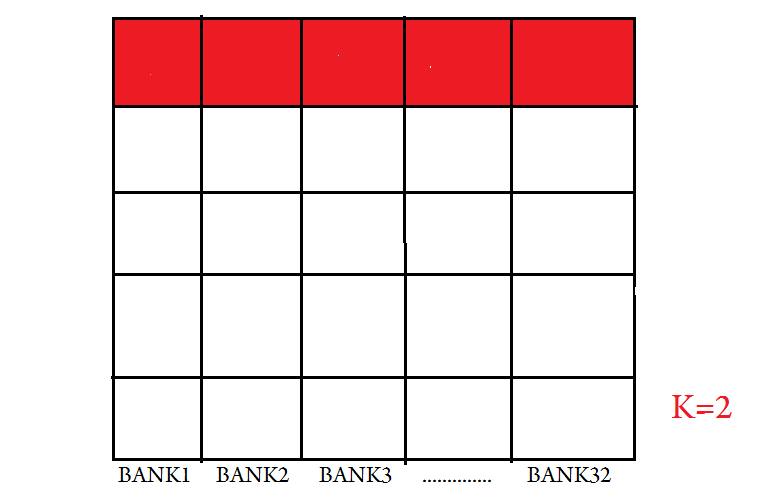


Dkj取值

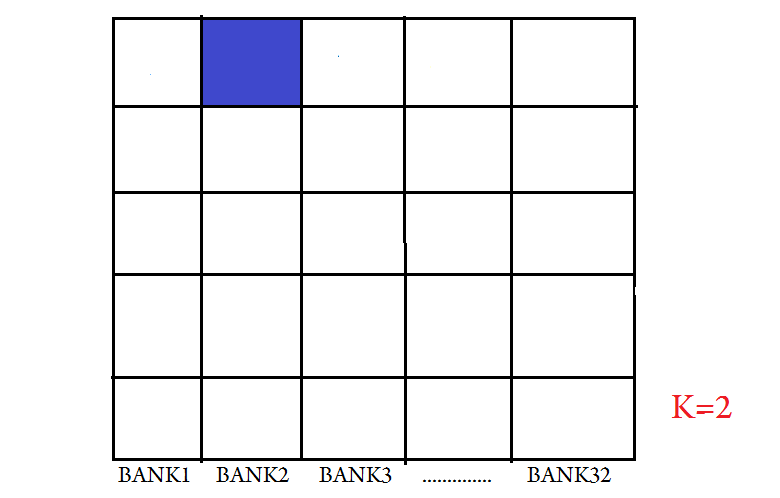
我們可以看到，在Dij和Dik的取值時，都會發生bank conflict(Dkj因為全部都是同一個位置所以不會)。而如果把thread跟(I,j) pair的indexing方式改成：(也就是把kernel中所有threadIdx的X/Y互換)



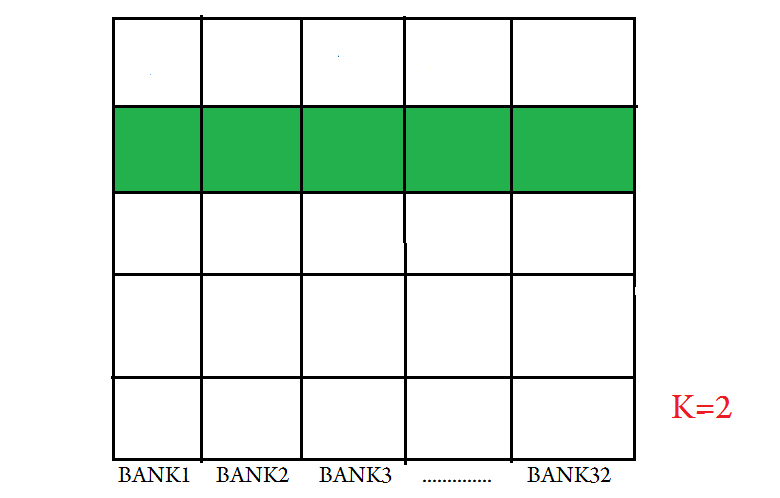
在一個回合中，同一個warp的取值就變成如下：



Dij取值



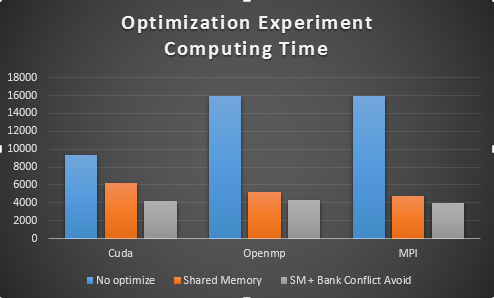
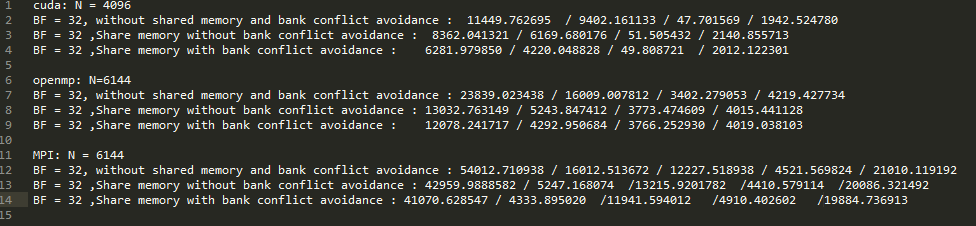
Dik取值



Dkj取值

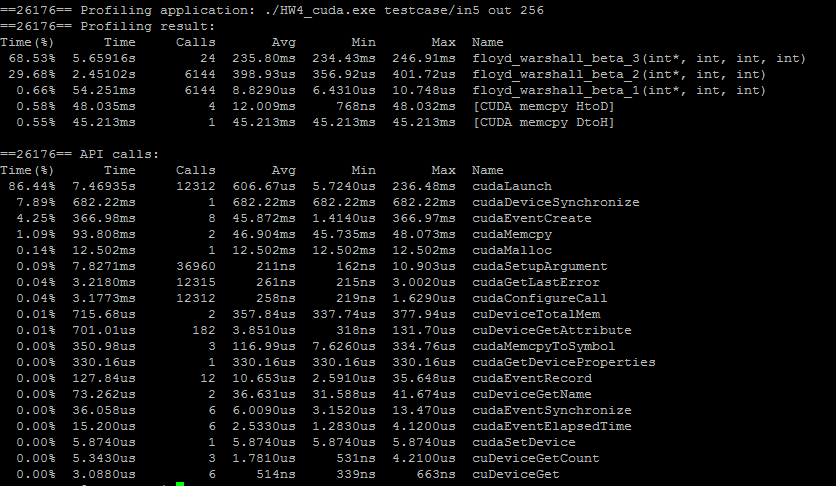
可以發現只要把x/y的index互換，bank conflict就解決了，在以上三次存取中都沒有發生任何bank conflict，效率因此獲得提升。

簡單的實驗證明optimization的效果如下：

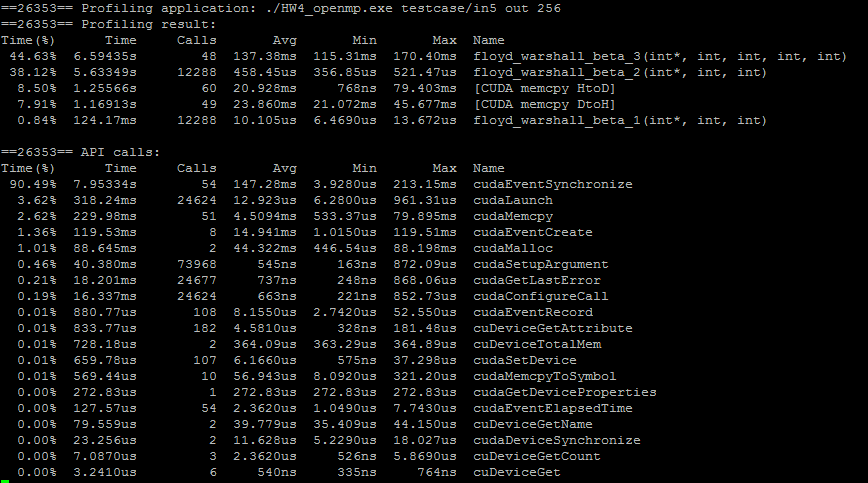


這是Cuda版本使用N=4096，Multi GPU版本使用N=6144，在BF=32的情況下，三種版本對應的computing time。可以發現shared memory會對整體速度有明顯的改善，而bank conflict avoidance可以再微幅改善。而在Multi GPU的版本中，改善的效過又優於SINGLE GPU的版本。

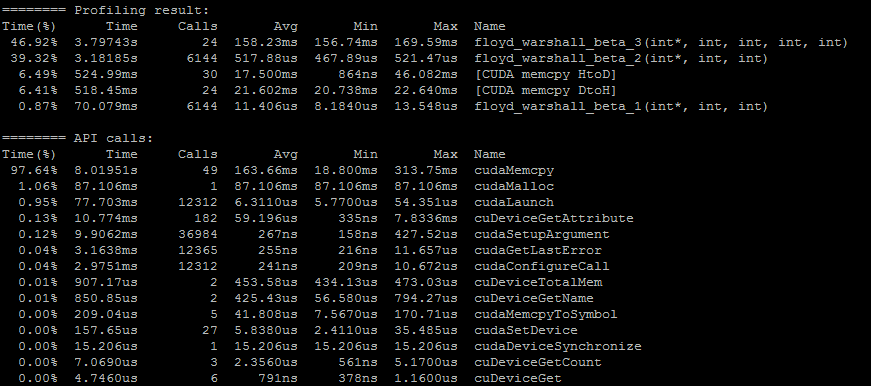
Profiling Result (N=6144, BF=256)



Cuda Profiling



Openmp Profiling



MPI (process rank=0) Profiling

從cuda版本中我們可以看到，phase 3的kernel確實是消耗最多時間((N/BF-1)\*(N/BF-1)個block)，其次是Phase 2((N/BF-1)\*2個block)，再來是Phase 1(1個BLOCK)。

另一個有趣的現象我們可以看到openMP的總時間和MPI的kernel執行總時間是差不多的；但是openMP的Kernel function時間似乎有重疊，從這部份我們可以推論不論哪個thread在使用kernel function，時間都會疊加紀錄上去，而MPI版本中因為兩個gpu分別是兩個不同的process控制，所以kernel時間加總大致就是kernel finction加總的時間。

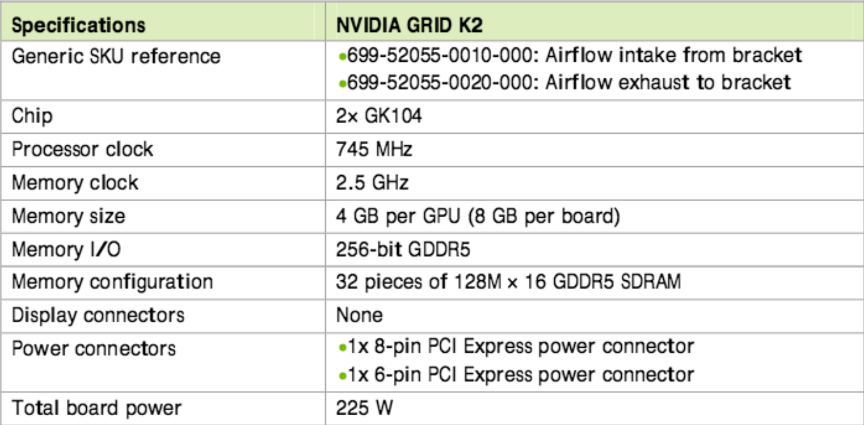
另外一點有趣的發現是，儘管openMP版本只需要每個iteration完呼叫一次Memcpy(Device to Device)就好，但是實際上可以發現cuda似乎還是會先把資料從GPU讀回Host之後再傳給另一個GPU，而不是直接傳過去，因此OpenMP版本的Memcpy時間並不比MPI分開兩次(D2H/H2D)來的快。

在CPU的方面，可以看到其實cuda大部分的API其實幾乎都不怎麼花時間的。至於我的cudaEventSynchronize在multi GPU版本會佔大部分時間主要是因為我在GPU之間的溝通都有使用cudaEventRecord來記錄時間，因此在傳送完之後會做cudaEventSynchronize，因此傳送GPU資料的時間就會全部加到cudaEventSynchronize上頭。

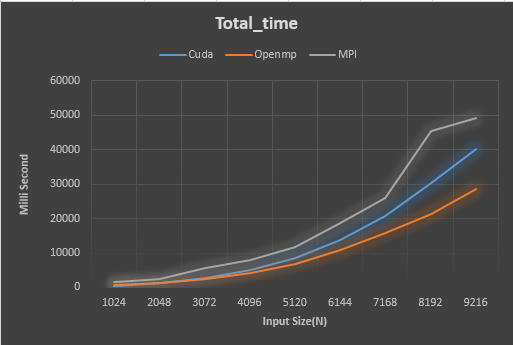
Experiment & Analysis

Weak scalability的實驗我使用的Block Factor固定為256。MPI的設定為Node=2:PPN=1

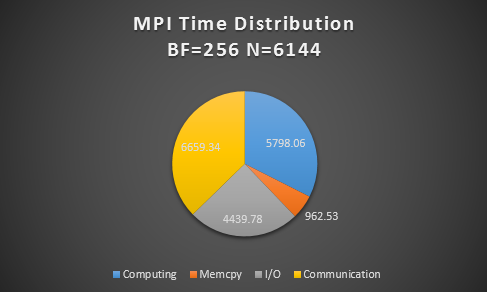
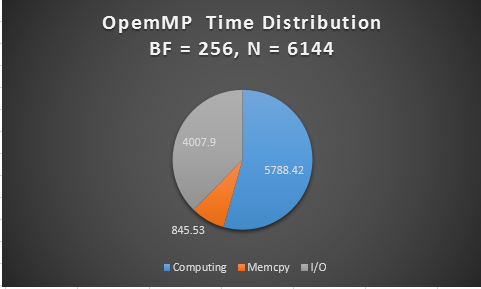
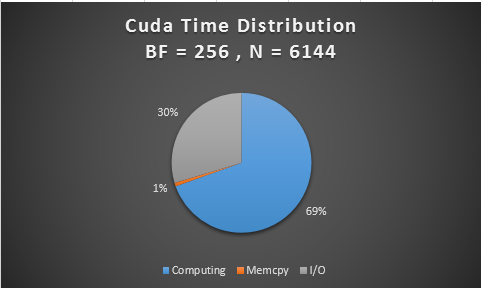
為了避免課程的cluster阻塞的問題，我使用的機器則是自己的機器，規格如下：



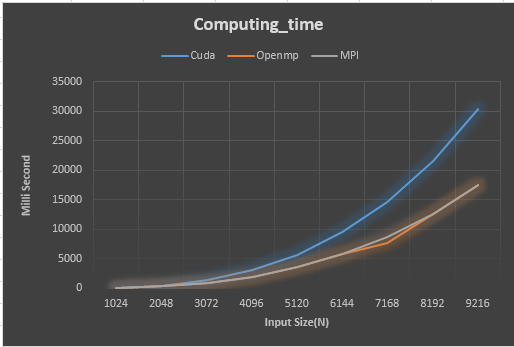
Weak Scalability & Time Distribution：



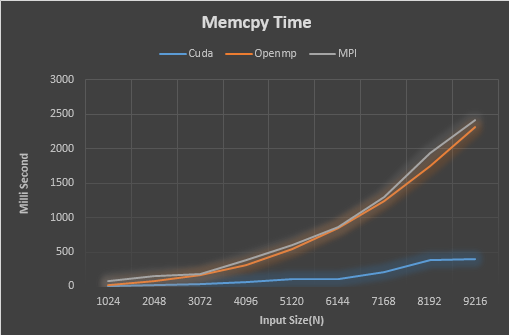
首先是總時間，我們可以看到在我的實作中，OpenMP的版本是最快的，而Single GPU版本次之，最慢的反而是MPI版本。這部分的原因後續會分析，主要是因為MPI的Communication時間太長導致。而OpenMP大約可以達到Single GPU時60~70%的時間，這部分主要是因為Phase 3的計算拆開由兩塊GPU同時執行，但是又因為有Memcpy的Overhead導致無法達到近50%的時間。觀察上圖不難發現時間隨著input size的增加呈現凹形曲線，可以與Floyd Warshall Time Complexity(N^3)互相驗證。



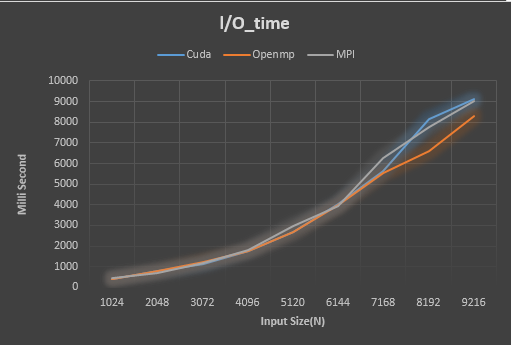
從Time Distribution圓餅圖中可以看出，在Cuda版本中Computing Time佔了大多數的時間且I/O也有一定分量；而到了OpenMP，在I/O時間基本上一樣的狀態下，儘管Computing Time有減少(兩塊GPU同時處理Phase 3)，Memcpy佔的比例卻開始變得不容忽視，但是整體還是以Computing Time為主。不過在MPI版本中，我們可以看到Communication反而成了最大的Bottleneck。其實在MPI版本中，Communication會佔這麼大的比例一部份原因我認為是只有兩塊GPU可以用。在Multi GPU的實作中，只要有牽涉到資料的傳遞，不論GPU是幾塊，總傳輸量是一樣的(有三塊的話每個GPU要處理的量就變成原本的1/3)。所以如果每個Process可以使用的GPU超過兩塊的話，傳輸總量不變，每個Process的計算資源變多，整體而言MPI的總時間應該會逐漸呈線性減少。



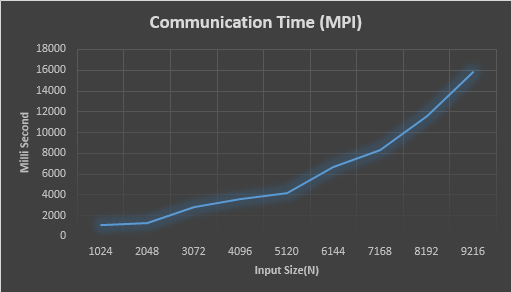
看到Computing Time的部分(Kernel runtime)。這部分就幾乎是跟理論值一樣的狀況，在GPU有兩塊的狀態下，儘管有分開來計算的只有Phase 3，但是實際上Phase3佔了總計算量的九成以上，所以實際上幾乎使可以達到近兩倍的計算效率。而實際上觀察Multi GPU和Single GPU的Computing Time，確實也達到了幾乎兩倍的速度。而OpenMP和MPI版本的速度幾乎是一模一樣的，這部分也可以驗證前面OpenMP和MPI版本的總時間差異是來自於MPI的Communication的假設。



Memory copy的部分，cuda因為只需要開始跟結束各自copy一次就好，自然是時間最少的，而OpenMP跟MPI的時間相近的原因我已經在Profiling section說明了，不過也許從上圖我們可以推論，呼叫一次cudaMemcpy(D2D)還是比分開來兩次呼叫cudaMemcpy(D2H)、cudaMemcpy(H2D)有快一點，也許CUDA在這方面有額外的優化？



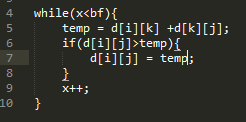
I/O時間的部分則跟預期的完全一樣，因為不論是哪種版本要讀和寫都是由CPU單獨執行，無法平行化。而時間隨著Input Size大致呈現平方倍的成長，這點則是因為input和output都是N\*N的graph。



Communication時間是我的MPI版本的Bottleneck，可以看到，因為每次的傳輸都是N\*N的資料量，整體曲線試呈大致平方倍成長的。也許是因為我使用的機器Message passing的速度本身就很慢的關係？

Blocking Factor effect

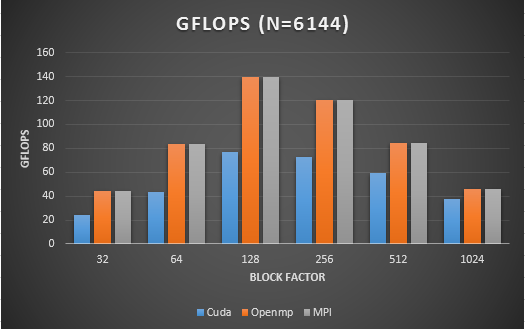
在GFLOPS的計算中，我們要計算的是每秒的總共有幾十億個運算被執行，考量到Floyd Warshall的核心計算：



是四個運算，再加上每個kernel的前置index轉換等計算，我計算GFLOPS的公式是

*GFLOPS = (10+ 4 \* (BF))\*(N/BF) \* N^2 / (10^9 \* computing Time)*

紅色是單一kernel function本身的運算次數，藍色是同一個(I,j)pair kernel會被呼叫的次數，綠色則是總共會有多少個thread會同時執行這個function。



以上則是GFLOPS的計算結果，可以發現GFLOPS一開始會隨著BF的增加而逐漸增加，這部分我認為是因為在小的BF時我的架構可能沒辦法有效榨乾GPU的運算資源(因為總Block數會太多，可能無法善用每個SM)，而BF超過128後，GFLOPS又會開始下降，我認為則是因為block裏的thread數(我的架構每個block裏的thread數是跟bf一樣的)過多，硬體的context switch會太頻繁導致效率下降。因此可以看出針對不同實作方法與切資料的方法選用適當的Block factor是很重要的。

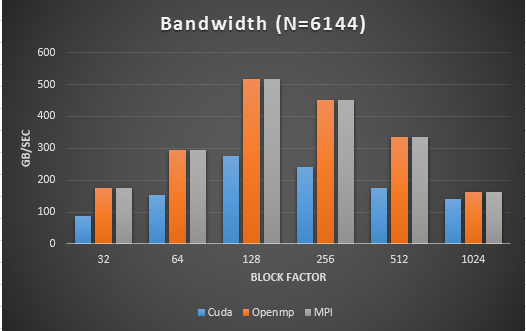
至於Bandwidth的計算，根據Nvidia官方手冊的寫法：



RB跟WB代表的則是在所有被launch過的GPU運算中讀寫的byte的總數。因為在Floyd Warshall算法中，每個回合我們都需要(1)讀取Dik (2)讀取Dkj (3)可能把結果寫回Dij中，如果假設寫回dij的機率是0.5，則當成每個回合總共有2.5次Read/write。因此，我計算Bandwidth的方法如下：

*Bandwidth = 2.5 \* 4 \* BF \* (N/BF) \* N^2 / (10^9 \* computing Time)*

其中4表示int有4個byte。紅色代表每次kernel function總共的read/write次數，藍色是同一個(I,j)pair kernel會被呼叫的次數，綠色則是總共會有多少個thread會同時執行這個function。



看的出來Bandwidth隨著Block factor呈現的趨勢其實跟GFLOPS很像，同樣都是逐漸增加，到了128時最大，其後再下降。不過另一點比較特別的是，依照我的機器規格，我算出來我的理論Bandwidth最佳值應該是200GB/SEC左右，但是上圖中許多結果都超出了理論最佳值。我認為最可能的原因大概是因為SIMD的架構下不同的threads可能很多都在某個相同的指令去存取同一個Global Memory的位置，因此可以用broadcast的方式，讀取一次再broadcast給所有需要使用該值的processor。不過這部份牽涉到很多GPU內部運作的細節，不太容易計算。

Experience & Conclusion

在這次的作業中，充分練習到了Cuda Programming以及如何設計資料結構和切割資料的方式來有效運用GPU SIMD架構的特性，以及一些優化Cuda Program的技巧。另外就是了解到了如何運用Multi GPU進一步加速運算，以及可能遇到的問題。

如果要說這次作業最困難的地方，大概就在於一開始沒有搞清楚Blocked APSP特性以及不同階段的資料相依的性質，導致一開始寫的時候怎麼寫結果都是錯的；也許可以當成借鏡，知道再要寫平行程式前，充分了解問題特性以及運算資源的特性，設計適當的演算法是非常重要的。