**Homework 3: All-Pairs Shortest Path**

108062313 黃允暘

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

我在hw3-1用的是Blocked Floyd-Warshall algorithm，是根據助教提供的template去優化。考慮到matrix block之間有dependency的問題，在這個部分就沒有寫得很複雜，只有單純的用OpenMP對cal( )裡面的for迴圈做平行化。一張含有 文字 的圖片

自動產生的描述

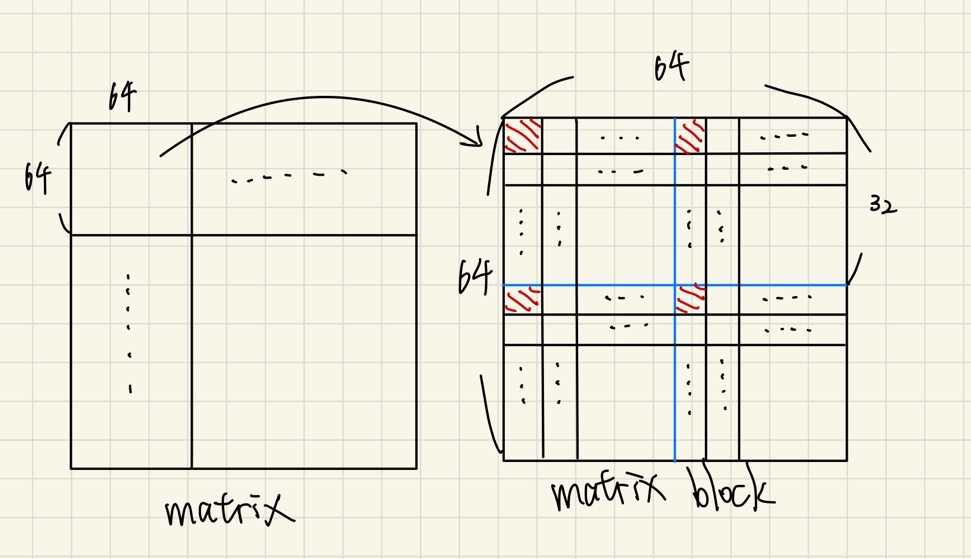
1. **How do you divide your data in hw3-2, hw3-3?**

我在這部分會講解我如何分配資料的，詳細的原因會在接下來的c, d講解。

**hw3-2:**

在hw3-2中使用的是Blocked Floyd-Warshall algorithm(BFW)，而剛好這個演算法會把matrix分成很多個block(matrix block)，因此我就一個cuda block負責一個matrix block的計算。且因為在BFS中每個phase要計算的matrix數量都不一樣，我會根據不同的phase給予不同數量的cuda blocks，至於詳細的原因及做法在下面會講到。

我的一個matrix block size為64\*64，而一個cuda block當中的threads數量我開(32, 32)，因此一個thread要負責4個點的計算。假設x=threadIdx.x, y=threadIdx.y，那該thread需要計算的點就是dist[y][x], dist[y+32][x], dist[y][x+32], dist[y+32][x+32]。如下圖所示：紅色的格子就是thread(0, 0)要計算的點。



**hw3-3:**

分配data的方式基本上跟hw3-2是一樣的，不過在hw3-2是由一張GPU負責全部的matrix blocks，在這邊因為有兩張GPU，因此在phase3的時候，GPU0只負責上半部的matrix blocks，而GPU1負責下半部的matrix blocks。

1. **What’s your configuration in hw3-2, hw3-3? And why?**

**hw3-2:**

* blocking factor:

在實作BFS的時候，越大的matrix block size越能達到加速的效果，但因為我有用share memory做optimization，因此block的size就不能無限上綱。根據上課所教，一個block內的share memory約為49512 bytes，約為12288個int的大小。而因為在phase3需要用到儲存三個matrix block的資料，因此一個block最多可以有4096個int(12288/3=4096)，也就是64\*64。綜合上述，我就將block factor設為64。

* #blocks

不同phase會擁有不同數量的cuda blocks，而一個cuda block負責一個matrix block。(matrix的大小為V\*V)

phase1: 因為只需要計算一個pivot，就給他1個cuda block。

phase2: 根據BFW，在這個階段需要計算pivot-row與pivot-col，因此我會開(V/64, 2)個cuda blocks。

phase3: 在phase需要計算剩餘的matrix blocks，因此我會開(V/64, V/64)個cuda blocks。

* #threads

因為一個blocks擁有1024個threads，因此threads的數量我開(32,32)。不過在這邊可以發現，一個cuda block負責一個matrix block，也就是說一個cuda block要負責64\*64個點的計算，那也就代表一個thread要負責4個點的計算。

**hw3-3:**

基本上架構跟hw3-3差不多，有差的地方在於在phase3中，開的cuda blocks數量。因為在這個phase中，是由兩張GPU各自負責一半的matrix blocks，因此只會開(V/64, V/128)個cuda blocks。

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

我是使用OpenMP完成multi GPU的操作。因為我只有在phase3的時候才做multi GPU的平行計算，因此只有在phase3結束的時候才需要溝通。需要溝通的原因是因為下一個round的pivot與pivot-row可能是由另外一張GPU所計算得出，所以在下一個round開始前，需要先從另一張GPU拿到它的pivot-row。另外需要注意的是，在傳輸資料前，需要確保兩張GPU都完成phase3，因此我會用omp barrier確保兩張GPU都完成計算。至於，傳輸的方式，我是用cudaMemcpy當中的cudaMemcpyDeviceToDevice，以達到peer-to-peer的溝通方式。

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

細節的部分在上面已經提過了，因此在這個部分只會帶過整體流程。

**hw3-2:**

首先在讀去input file之後，我會先執行padding，目的是為了避免matrix size無法整除blocking factor的問題。因此會先將matrix size補零擴大到blocking factor的整數倍。並將擴充後的matrix複製到GPU裏面。接下來就會不斷經過3個phase:

**phase1:**

在phase1當中，只需要計算pivot block。另外，因為我有用shared memory進行加速，所以需要開一個大小為blocking factor\*blocking factor的2D shared memory，並將要計算的matrix block複製到shared memory當中。另外也會呼叫\_\_syncthreads()，確保資料都已經存到shared memory。接著就可以進到迴圈當中，以shared memory進行實際的BFW phase1操作。最後，因為在phase1當中有dependency的關係，所以在for迴圈當中也會呼叫\_\_syncthreads()，確保同步。全部結束之後，會把資料從shared memory搬回GPU memory。

**phase2:**

在phase2當中，要計算pivot-row或pivot-col，需要pivot的資料。因此我會開兩個大小為blocking factor\*blocking factor的2D shared memory，其中一個儲存pivot的資料，另一個儲存row-pivot或col-pivot的資料。將所需要的資料都複製到shared memory之後，呼叫\_\_syncthreads()，確保資料都已經存到shared memory後，就進行BFW phase2得實際操作。我在這個部分是開(V/64, 2)個cuda blocks，因此blockIdx.y為0的cuda block就負責pivot-col的計算，而blockIdx.y為1的cuda block就負責pivot-row的計算。全部結束之後，會把資料從shared memory搬回GPU memory。

**phase3:**

在phase3需要計算剩下的區域。因為除了需要目標matrix的資料之外，也需要pivot-col以及pivot-row的資料。因此我在這邊會開三個大小為blocking factor\*blocking factor的2D shared memory，並在資料複製完之後呼叫\_\_syncthreads()。而每個cuda blocks透過threadIdx與blockIdx找到要計算的matrix block。另外，在這個部分的data並沒有dependency，所以在for迴圈當中並不用呼叫\_\_syncthreads()。全部結束之後，會把資料從shared memory搬回GPU memory。

**hw3-3:**

在這個部分我是使用OpenMP完成multi GPU的操作的，演算法細節基本上跟hw3-2一樣，因此我在這邊專注講解如何使兩張GPU平行運算。

* Divide data

就像上面提過的，我只有在phase3進行GPU的平行計算。在phase3當中，GPU0負責上半部的matrix blocks，而GPU1負責下半部的matrix blocks。而因為下一個round的pivot與pivot-row可能是由另外一張GPU所計算得出，所以在下一個round開始前，需要先從另一張GPU拿到它的pivot-row。在透過cudaMemcpy當中的cudaMemcpyDeviceToDevice傳輸完資料之前，我會呼叫omp barrier確保兩張GPU都完成運算了。

1. Profiling Results

在我的實作當中，最大的kernal是cal3，也就是計算phase3的kernal funciton。以下是利用NVIDIA profiling tools所測出來的cal3的profiling result。我所使用的測資是p20k1。一張含有 文字 的圖片

自動產生的描述

1. Experiment & Analysis
2. **System Spec**

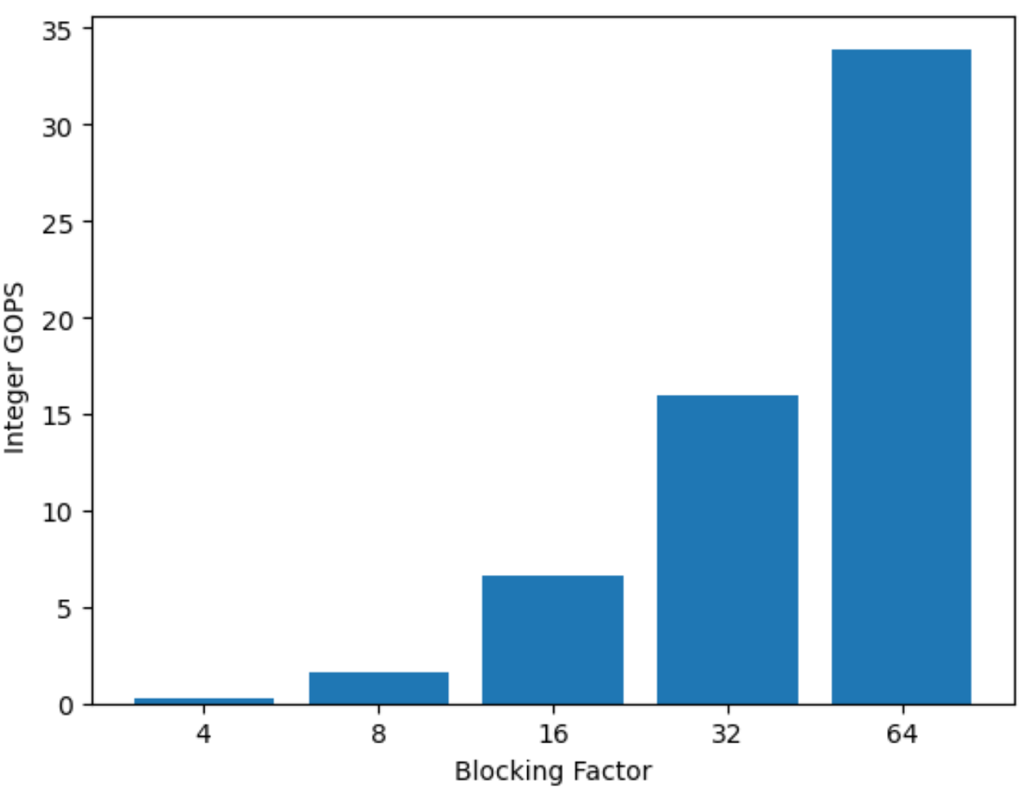
課程提供的hades server。

1. **Block Factor**

因為我有使用shared memory進行加速，因此Block Factor最多只能開到64。

* Integer GOPS

我首先用nvprof當中的inst\_integerf去測試kernal funcion執行過程中總共會有多少個指令，接著再計算執行時間，相除之後就可以得到GOPS。而我是用c18.c做測試的。

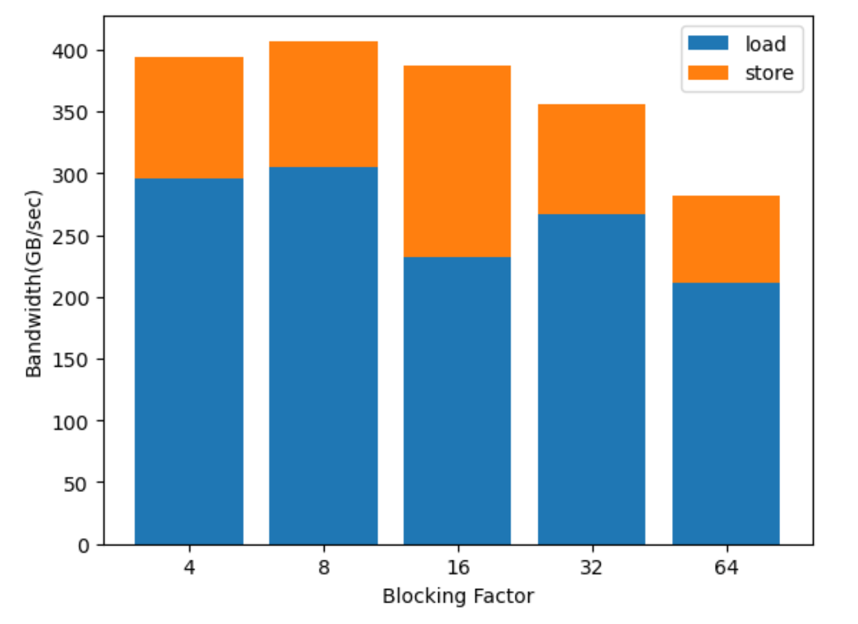


可以從上圖中看到，隨著Blocking Factor上升，Integer GOPS也會隨之上升。我推測這是因為Blocking Factor越大，越能展現Blocked Floyd-Warshall algorithm的優勢，也就是access memory更有效率，就不用一直等待搬移資料的時間，而且也不用一直等待synchronization。

* Bandwidth

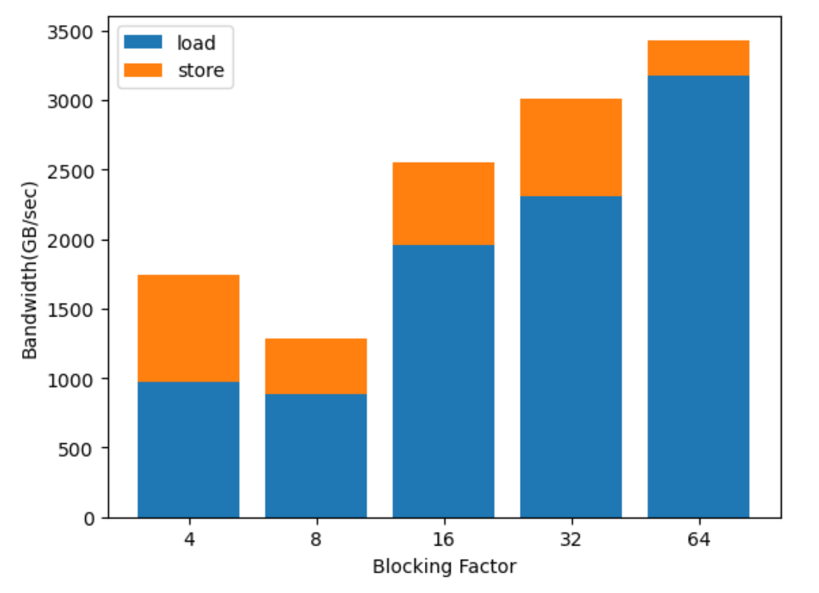
我在這部分都是用nvprof的matrix去做測試的。

1. global memory bandwidth



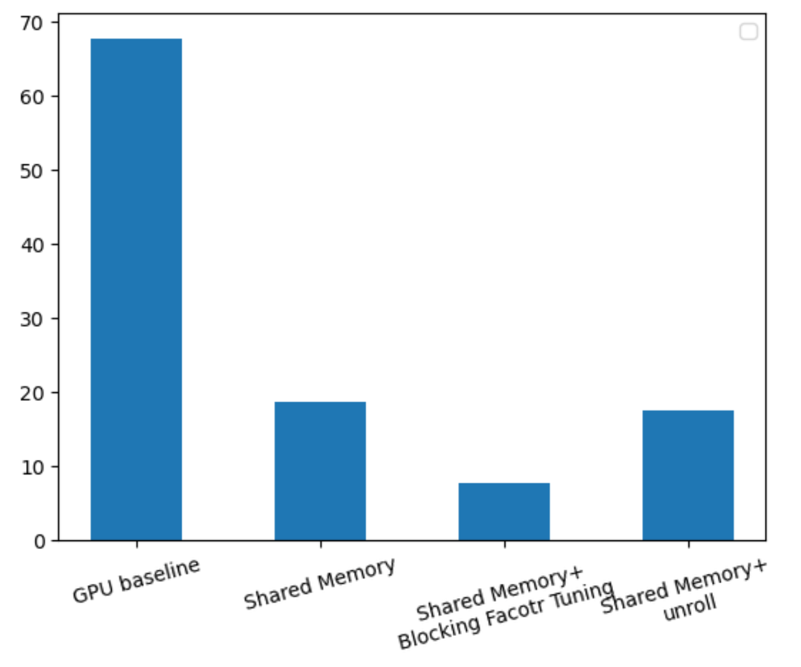
當Blocking Factor越大，global memory bandwidth會有越來越小的趨勢。這是是因為Blocking Factor越大，代表所開的shared memory也會越大，因此不用一直去global拿資料，導致global memory bandwidth也隨之下降。而load memory bandwidth的值大於store memory bandwidth，推測是因為在phase2及phase3中，分別會開出兩塊及三塊的shared memory，並從global memory load進資料。但最後存會global memory的只有一塊shared memory，因此store memory bandwidth的值會小於load memory bandwidth。

1. shared memory bandwidth



當Blocking Factor越大，shared memory bandwidth也會跟著變大。這是因為當shared memory越大，程式就不用頻繁地去global memory那資料，可以直接從shared memory讀取資料就好了，也就代表著程式的效率提高了。而load memory bandwidth的變化大於store memory bandwidth，推測是因為在phase2及phase3中，分別會開出兩塊及三塊的shared memory，並各自load進資料。而在for迴圈更新的卻都只有一塊shared memory，因此load memory bandwidth的變化會大於store memory bandwidth。

1. **Optimization**

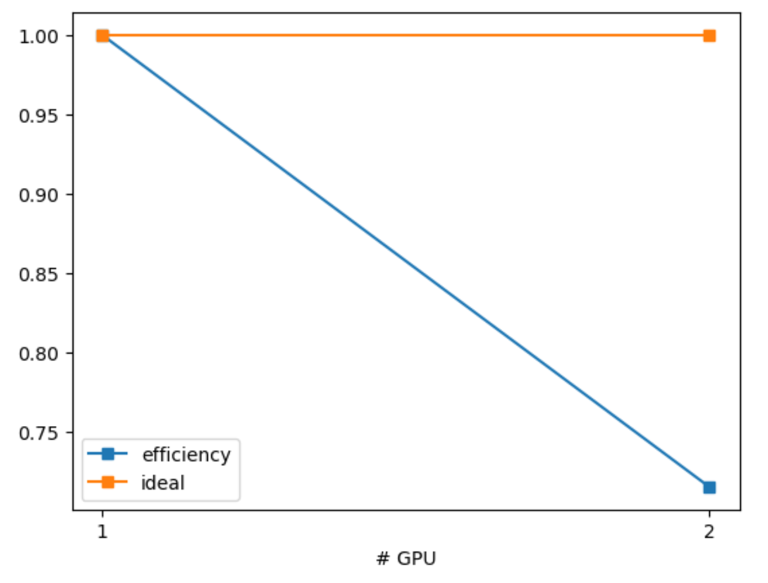


我主要有進行三種optimization:

1. Shared memory (2) Bigger Blocking Factor (3) pragma unroll

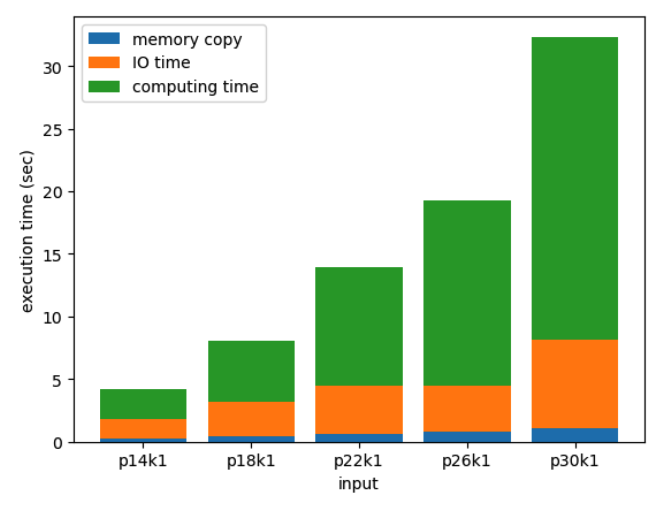
在這當中，利用shared memory可以最有效的降低執行時機，其次是Bigger Blocking Factor( Blocking Factor為64)，最後是pragma unroll。因為shared memory可以最有效地提高performance，因此不惜把Blocking Factor限制在64，也要使用shared memory。

1. **weak scalability(3-3)**



因為在這次作業當中，input是一個V\*V的matrix，因此我找了hw2 testcases當中的p12k1與p17k1來做測試，這兩筆測資各有12000與17000個點，17000^2 / 12000^2 約為2，也就是problem相差一倍。我先用single GPU跑p12k1再用2 GPU跑p17k1，以達到測試weak scalability的目的。從上圖來看，scalability並沒有到很理想。我推論是因為我只有對phase3做平行化，而且整個演算法當中有許多地方要進行memory copy與synchronization，因此就會讓程式的scalability下降。

1. **Time distribution**



我用五種不同的input data去測試程式的memory copy time, IO time和computing time，結果如上圖所示。可以看到基本上三個指標都是隨著input增大而增加。我是用nvprof測試memory copy time和computing time，然後用std::chrono::steady\_clock計算IO時間。

1. Experience & Conclusion

這一次作業真的蠻難的，光一開始的data distribution就想了很久，而之後還要做許多的優化，global memory跟shared memory的對照也要很小心。再加入第二張GPU的時候，也有發生許的synchronization的問題。cuda program真的有很多需要注意的小細節。但也透過這次的作業，使我更了解GPU的架構。