Homework 3: All-Pairs Shortest Path

108062313 黃允暘

1. Implementation

a. 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。

```
#pragma omp parallel num_threads(ncpus)
{
    #pragma omp for schedule(static)
    for (int b_i = block_start_x; b_i < block_end_x; ++b_i) {
        for (int b_j = block_start_y; b_j < block_end_y; ++b_j) {
            for (int k = Round_B; k < Round_1_B && k < n; ++k) {…
        }
    }
}
```

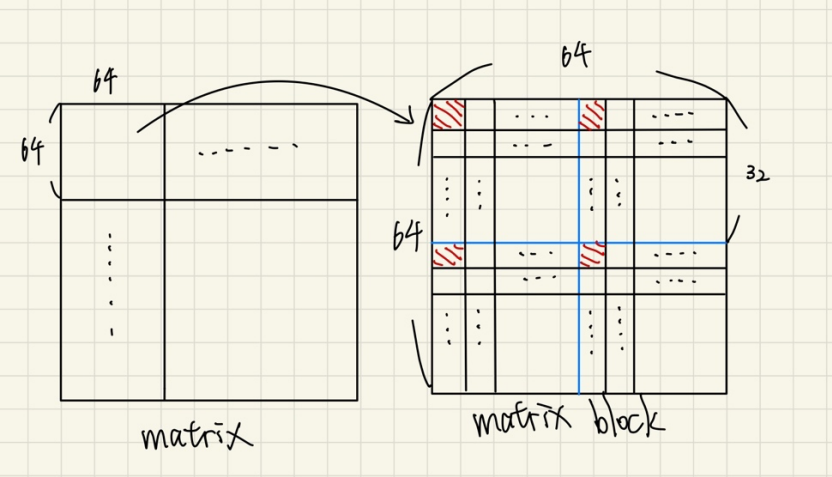
b. How do you divide your data in hw3-2, hw3-3?

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。

c. 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。

d. 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。

e. 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。

2. 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。

```
==411428== NVPROF is profiling process 411428, command: ./hw3-2 cases/p20k1 out.out
==411428== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==411428== Profiling application: ./hw3-2 cases/p20k1 out.out
==411428== Profiling result:
==411428== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal3(int*, int, int, int)
        313                        achieved_occupancy                        Achieved Occupancy    0.936442    0.947273    0.945431
        313                                  sm_efficiency                   Multiprocessor Activity      99.91%      99.99%      99.98%
        313                          shared_load_throughput             Shared Memory Load Throughput  3175.3GB/s  3472.6GB/s  3359.4GB/s
        313                         shared_store_throughput            Shared Memory Store Throughput  259.21GB/s  283.48GB/s  274.24GB/s
        313                                 gld_throughput                  Global Load Throughput  194.41GB/s  212.61GB/s  205.68GB/s
        313                                 gst_throughput                 Global Store Throughput  64.803GB/s  70.869GB/s  68.560GB/s
```

3. Experiment & Analysis

a. System Spec

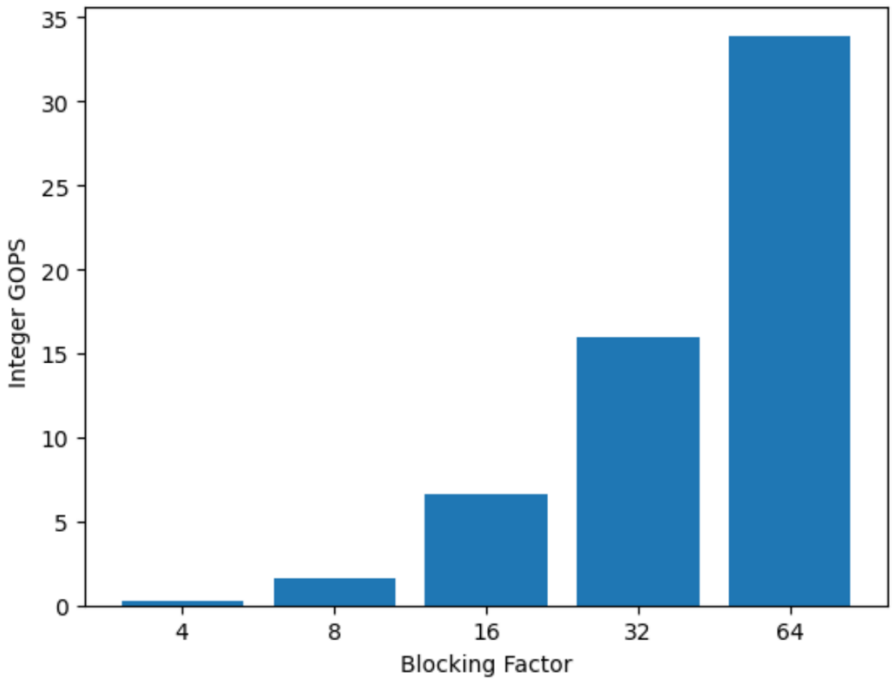
課程提供的 hades server。

b. 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。

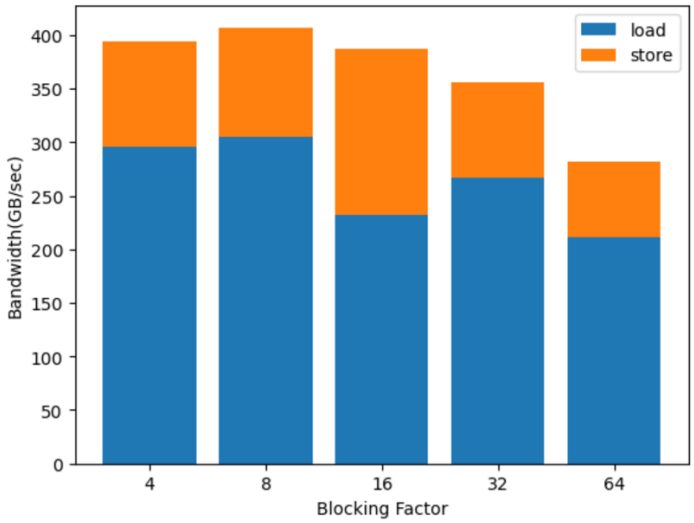


可以從上圖中看到，隨著 Blocking Factor 上升，Integer GOPS 也會隨之上升。我推測這是因為 Blocking Factor 越大，越能展現 Blocked Floyd-Warshall algorithm 的優勢，也就是 access memory 更有效率，就不用一直等待搬移資料的時間，而且也不用一直等待 synchronization。

- Bandwidth

  我在這部分都是用 nvprof 的 matrix 去做測試的。

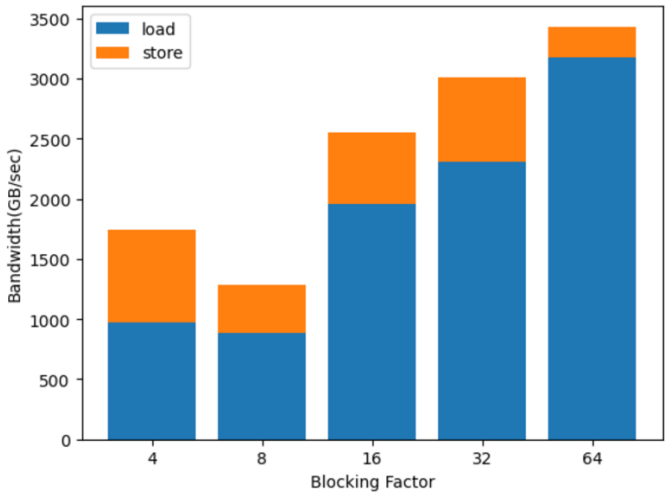
  (1) global memory bandwidth



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

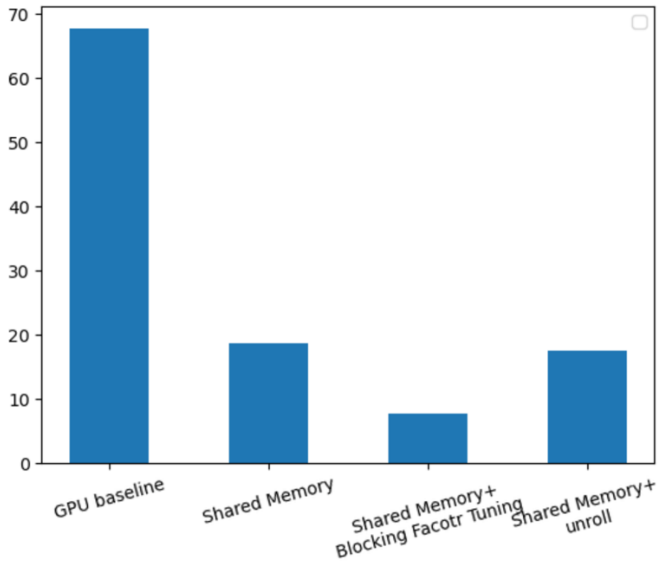
bandwidth 的值大於 store memory bandwidth，推測是因為在 phase2 及 phase3 中，分別會開出兩塊及三塊的 shared memory，並從 global memory load 進資料。但最後存會 global memory 的只有一塊 shared memory，因此 store memory bandwidth 的值會小於 load memory bandwidth。

(2) 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。

c. Optimization

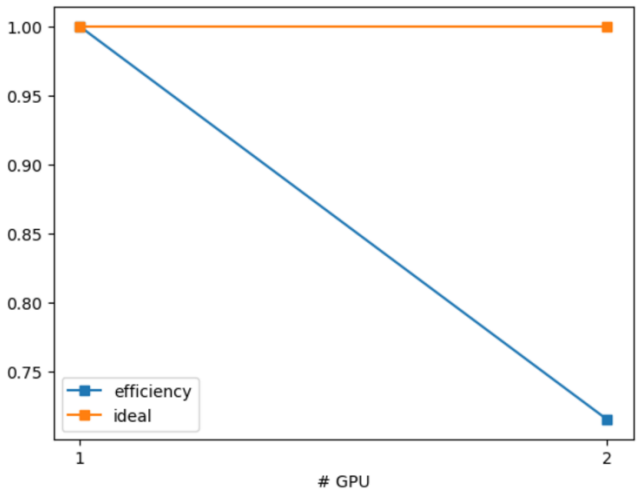


我主要有進行三種 optimization:

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

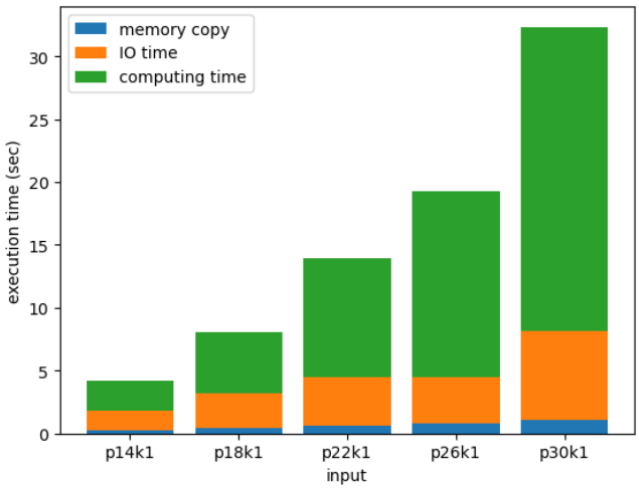
在這當中，利用 shared memory 可以最有效的降低執行時機，其次是 Bigger Blocking Factor(Blocking Factor 為 64)，最後是 pragma unroll。因為 shared memory 可以最有效地提高 performance，因此不惜把 Blocking Factor 限制在 64，也要使用 shared memory。

d. 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。

e. 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。

4. 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。