Homework 3: All-Pairs Shortest Path

107062120 姚潔恩

Implementation

I. hw3-1

在 hw3-1 中我使用普通的 Floyd-Warshall 演算法，其中第一層迴圈是中間點 k，第二、三層是跑過每一個組合(i, j)。平行的部分是使用 pthread，對於第二層 Floyd-Warshall 的迴圈平行化。因為計算所需的 data 有 dependency，所以必須在第二層迴圈結束後做一個同步化的動作，實作上使用 pthread 的 barrier。

II. hw3-2

A. Configuration

hw3-2 中使用 Blocked Floyd-Warshall (以下簡稱 BFW)演算法，每個 phase 就對應到一個 kernel function，每個 GPU block 就是負責計算 BFW 中的一個 block。

- Blocking Factor

因為我有使用 shared memory 加速，因此 BFW 的 block 大小就必須受到限制。根據計算：49152(總 shared memory 大小)除以 4(int 為 4byte)再除以 3(下面會說到計算 phase2 時需要存三個 block 的資料)的開根號剛好為 64，因此我的 Blocking Factor 就設定為 64。

- #Thread

thread 的數量為(blocking factor, 1024/blocking factor)=(64, 16)。因此可以發現每個 thread 要負責計算一個 BFW block 中的四個資料。我分資料的方法是讓 i=threadIdx.y*4、j=threadIdx.x 其中(i, j)分別代表每個 thread 要負責計算的 row 起始位置以及 column 位置。而每個 thread 負責計算的四個 element 就是(i, j)、(i+1, j)、(i+2, j)、(i+3, j)。

B. Implementation

- Padding

為了讓 graph matrix 的 row 大小和記憶體有 alignment 以及之後寫 kernel function 比較方便(不用考慮一些 index 超出範圍的問題)，因此我讓 graph matrix 的 column 數 pad 到 128 的倍數，放入的值為 INF 因此不會影響結果。

- Phase1

Phase1 只要計算 pivot block，因此只需要一個 GPU block。我有使用 shared memory 加速，方法就是開大小為 Blocking Factor * Blocking Factor 的二維陣列。在進行 Floyd-Warshall 之前會先把 pivot block 從 global memory load 到 shared memory 裡

面。每個 thread 各自 load 自己負責的四個 element (如上#threads 中所述)，並在最後做__syncthreads 確保 block 中的所有 thread 都已經將資料正確存在 shared memory。接下來在跑 Floyd-Warshall 的迴圈時就可以使用 shared memory 中的 data，避免 access global memory。在做 Floyd-Warshall 的過程中因為有 data dependency 所以也要使用__syncthreads 同步。最後完成 Floyd-Warshall 後再把 shared memory 的資料 copy 回 global memory。

- Phase2

Phase2 為計算 pivot-row 和 pivot-column，因此我是開(V/Blocking Factor)個 GPU blocks (blockIdx.x=Round 的 block 是 pivot block，會直接 return)，其中每個 GPU block 會負責計算 pivot-row 和 pivot-column 中的一個 block。也就是每個 GPU block 負責的就是(Round, blockIdx.x)以及(blockIdx.x, Round)兩個 block。因為要計算 pivot-row 和 pivot-column，再加上需要 pivot block 的資訊，所以需要三塊 shared memory 分別為 pivot-row、pivot-column、pivot block (因此最大的 Blocking Factor 為 64)。接下來就和 phase1 一樣，會先 load shared memory 然後再進行 Floyd-Warshall 然後再把 shared memory load 回 global memory。

- Phase3

最後 phase3 要計算剩下的區塊，我是開(V/Blocking Factor, V/Blocking Factor)個 GPU blocks (blockIdx.x=Round 或 blockIdx.y=Round 的 block 是 pivot block 或 pivot-row 或 pivot-column，會直接 return)，其中每個 GPU block 負責的就是(blockIdx.y, blockIdx.x)這個 block。需要的 shared memory 分別為 pivot-row、pivot-column 兩塊。Phase3 比較特別的是因為 data 沒有 dependency(只會用到 pivot-row 和 pivot-column)，所以在 Floyd-Warshall 的迴圈不需要加__syncthreads。

III. hw3-3

hw3-3 我使用 OpenMP 完成 multi GPU 的實作，其中 BFW 演算法的 kernel function 基本上都和 hw3-2 一樣，因此以下只說明我如何分配工作以及溝通的方式。

A. Divide workload & Communicate

為了減少兩張 GPU 溝通以及同步的次數，我只有在 phase3 作 multi GPU 的平行計算，phase1 和 phase2 每個 GPU 還是各自算自己的。分工作的方法是在 phase3 時 GPU0 只負責上半部的 blocks，而 GPU1 只負責下半部的 blocks。phase3 結束以後會需要溝通，因為下一個 round 的 pivot block 和 pivot-row 可能是由另一張 GPU 所計算，這時就需要從另一張 GPU 複製下一輪的 pivot-row 的正確資訊，以確保下一個 round 可以被正確計算。複製資料的方式我使用 cudaMemcpy 的 cudaMemcpyDeviceToDevice。另外在複製資料前我會用 cudaDeviceSynchronize

搭配 OpenMP 的 barrier 確保兩個 GPU 都完成計算以後才進行資料複製，以避免複製到錯誤的資料。

Profiling Results

這次作業中最大的 kernel 是第三個 phase 的 cal_phase3，以下是 cal_phase3 的 profiling results，使用的資料是 p15k1。

```
(venv) pp21s17@hades01 ~/hw3-2> sh sh/profiling_results.sh                                                    master?
==49682== NVPROF is profiling process 49682, command: ./hw3-2 /home/pp21/share/hw3-2/cases/p15k1 p15k1.out
==49682== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==49682== Profiling application: ./hw3-2 /home/pp21/share/hw3-2/cases/p15k1 p15k1.out
==49682== Profiling result:
==49682== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: cal_phase3(int*, int, int)
        236                        achieved_occupancy                        Achieved Occupancy    0.941660    0.943976    0.942972
        236                                sm_efficiency                   Multiprocessor Activity      99.93%      99.98%      99.96%
        236                    shared_load_throughput             Shared Memory Load Throughput  3415.7GB/s  3457.1GB/s  3435.4GB/s
        236                   shared_store_throughput            Shared Memory Store Throughput  142.32GB/s  144.05GB/s  143.14GB/s
        236                            gld_throughput                    Global Load Throughput  213.48GB/s  216.07GB/s  214.71GB/s
        236                            gst_throughput                   Global Store Throughput  71.160GB/s  72.024GB/s  71.570GB/s
(venv) pp21s17@hades01 ~/hw3-2>                                                                               master?
```

Experiment & Analysis

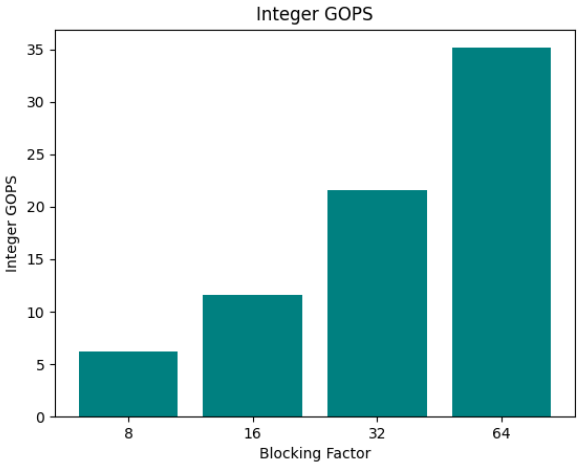
A. System Spec

實驗環境使用課程提供的 server hades。

B. Blocking Factor

因為 shared memory 的大小有限，因此測量 Blocking Factor 的上限只能到 64。另外因為使用 nvprof 跑 metric 會需要大量時間，因此 GOPS 的部分使用的資料是比較小的 c18.1，bandwidth 的部分則是使用 c20.1。測量的對象都是最大的 kernel cal_phase3。

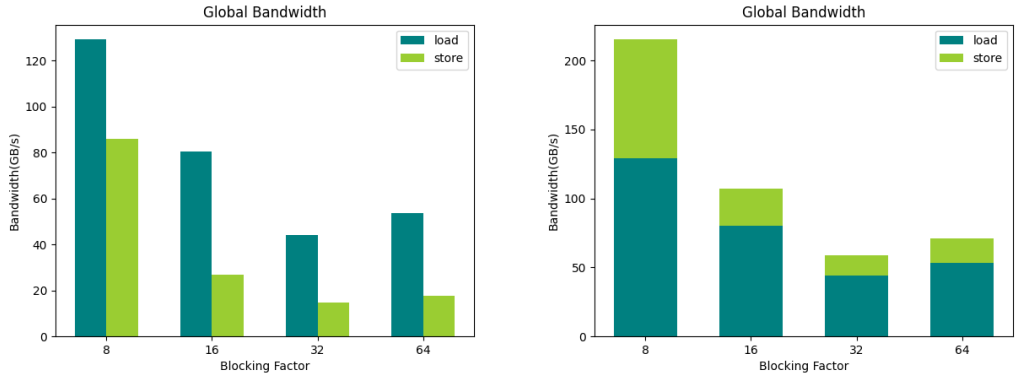
- Integer GOPS



首先我使用 inst_integer 這個 metric 測量 phase3 有多少 integer 的指令，接著再用 nvprof 計算 phase3 執行的時間。最後用指令數量除執行時間就可以得到 GOPS。可以看到 Blocking Factor 和 integer GOPS 呈現正相關，我想原因是 Blocking Factor

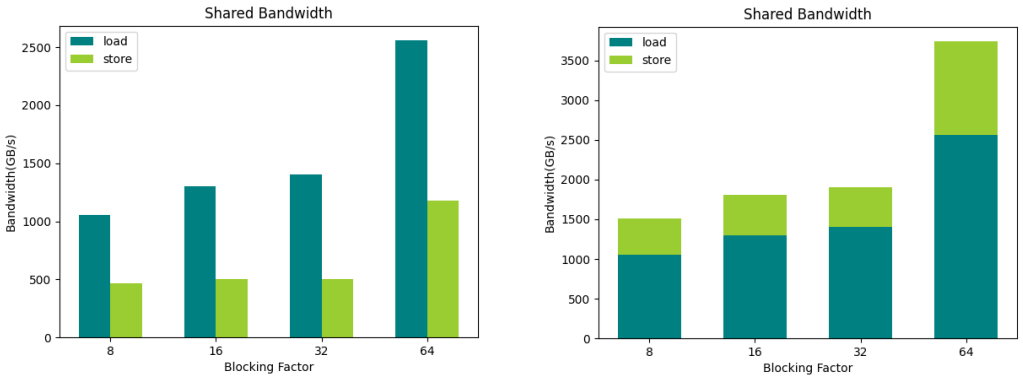
越大則 access memory 會比較有效率，因為 Blocking Factor 如果很小就需要頻繁的搬動 memory 以及增加 access global memory 的次數，也會增加許多同步化的時間。

- global memory bandwidth



測量 global memory bandwidth 我使用 nvprof 中的 gld_throughput 和 gst_throughput 來量測。可以看到 Blocking Factor 越大則 global memory bandwidth 越小，原因為使用較大的 Blocking Factor 可以降低 access global memory 的次數。

- shared memory bandwidth



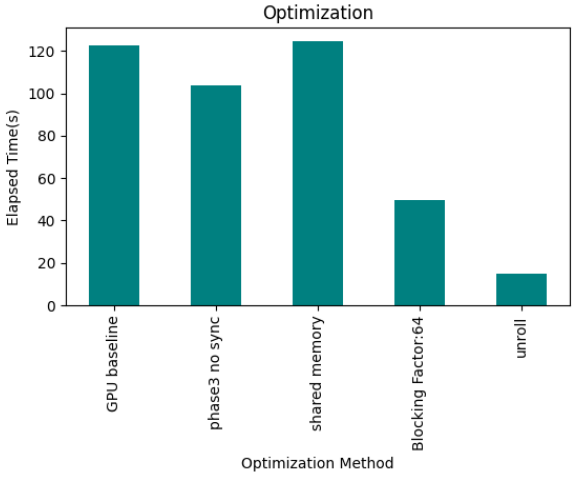
測量 shared memory bandwidth 我使用 nvprof 中的 shared_load_throughput 和 shared_store_throughput 來量測。可以看到 shared memory bandwidth 和 Blocking Factor 呈正向關係，顯示使用較大的 Blocking Factor 可以增加使用 shared memory 的效率。

C. Optimization

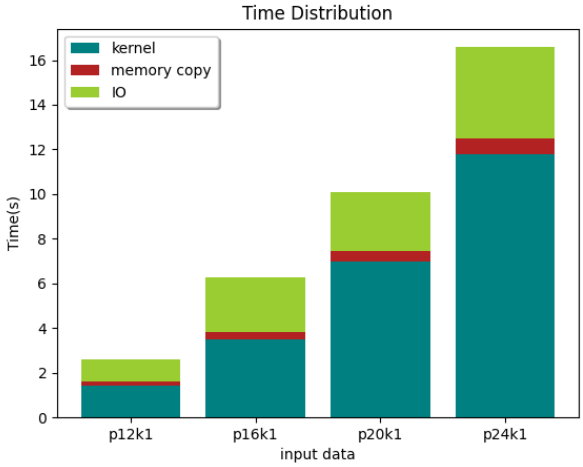
Optimization 的部分我使用較大的測資 p24k1 測試，使用 clock_gettime 測量 block_FW 的時間(input、output 的 IO 時間就不會被計入)。以下是四個優化：

- No synchronize in phase3: phase3 跑 Floyd-Warshall 迴圈時不用 __syncthreads。
- Shared memory: 使用 shared memory 加速，減少 access global memory。
- Bigger Blocking Factor: 從 32 提升到 64 以完全利用 shared memory。

◆ Unroll: 手動將迴圈 unroll，變成寫死四次運算，減少 for-loop 的 overhead。從結果可以看到把 phase3 的__syncthreads 拿掉以後因為減少需要同步化的 overhead 因此速度有明顯提升。比較奇怪的是使用 shared memory 優化且 Blocking Factor 設定為 32 的實驗竟然比較慢。我覺得原因可能是使用 shared memory 以後會多了 load 資料時需要同步的時間，而 Blocking Factor 比較小的情況下還是需要頻繁的 access global memory，所以 shared memory 可以提供的加速也比較有限。然而可以看到一但把 Blocking Factor 提升到 64 就可以完全展現 shared memory 提供的加速。最後因為固定 Blocking Factor 為 64 所以可以確定一個 thread 就是負責做 4 個 element，因此可以直接把迴圈 unroll 成四行程式。很特別的是這樣的改寫竟然也提供了很大的優化，讓我體會到 for-loop 所造成的 overhead。



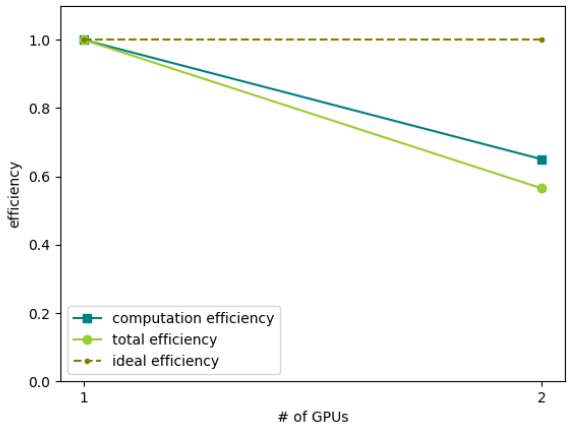
D. Time Distribution



我使用四種不同大小的 input 測量 time distribution，kernel 以及 memory copy (H2D, D2H)是使用 nvprof 測量，IO 時間則是使用 clock_gettime。可以看到基本上三項時間都和 input 大小呈正相關。

E. Weak scalability

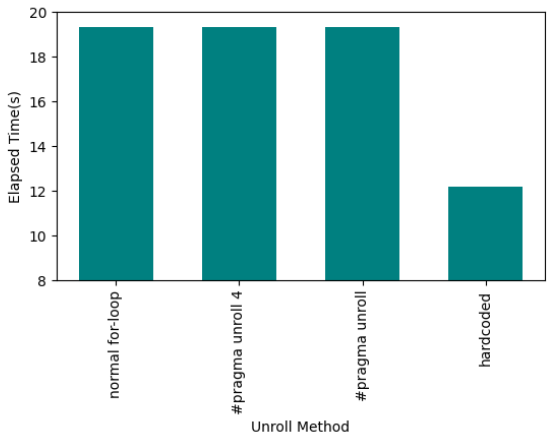
因為 Floyd-Warshall 會需要一個 V*V 的 matrix 作為 graph，因此我將 problem size 定義為 V^2。如果 problem size 要兩倍，測資的 vertex 數就要是根號二倍。我使用 hw3-2 的 p17k1 和 p24k1，兩個測資的的 vertex 數量分別為 17000 和 24000，相差約 1.412 倍。其中我用 single gpu 跑 p17k1 再用 multi gpu 跑 p24k1。可以看到 multiple GPU 的效果真的沒有很好。如果單看 kernel computation 的時間還比較好(computation efficiency)，但如果再加上 memory copy、synchronization 的時間效果就會變得更差(total efficiency)。



F. Others

- unroll

在我的設定下一個 thread 需要負責計算四個 element，因此在 Floyd-Warshall 迴圈內會需要跑一個次數四的 for-loop。我試過用#pragma unroll 去 unroll 迴圈，但是如下圖，和最左邊原本的 for-loop 相比加上#pragma unroll 根本沒有幫助。反而是手動將迴圈展開寫死成四個運算可以有顯著的進步。然而這樣的缺點就是一但寫死了就無法隨意調整其他設定如 Blocking Factor。



Conclusion

這一次 CUDA programming 的作業實在是讓我體驗到 CUDA 的難。雖然隨便寫一份 CUDA program 似乎沒有到非常困難，然而要去優化的時候就非常有挑戰。我覺得最困難的地方是在寫程式的過程中要不僅要確保自己程式的邏輯正確，同時也要不斷地去對應 GPU 硬體的架構，因為許多的優化都和硬體架構有關。總結來說就是 CUDA 不像以往寫純軟體的感覺，而是需要從軟體到硬體都有清楚的理解。