Homework 3 report

112062524 詹博允

1. Implementation

HW3-1

- **Which algoritm do you choose in hw3-1 ?**

  我使用了標準的 Floyed-Warshall algorithm 來完成.

- **How do you divide your data in hw3-1 ?**

  因為用 openMP 將回圈 collapse, 又因為這是 cpu 版本, 不會有 memory access efficiency 的問題, 所以概念上是將 adjacency matrix 做等分切割.

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

  在一開始實作時，我選擇使用 spec 提供的 Blocked Floyed-Warshall 來實作，並用 pthread 來平行化，但是後來發現整體需要花費的時間過高，因此只有寫出 sequential 版本的 Blocked Floyed-Warshall, 並在跟同學討論過後決定將原始版本的 Floyed-Warshall 平行化 (因為 hw3-1 並沒有要求要比較 performance)

  最後提交的版本是用原版的 Floyed-Warshall 並用 openMP 做平行，主要平行化的地方是：

  1. 初始化 adjacency matrix 時用 collapse 把 loop 攤開
  2. 執行 Floyed-Warshall 時用 collapse 把 loop 攤開

HW3-2

- **What is your configuration in hw3-2 ? And why ? (e.g. blocking factor, #blocks, #threads)**
  - **Blocking factor :**

一開始實作時我選擇用 blocking factor = 8 來實作, 但是在後續的跑分發現這樣只能通過 correctness, performance 則會全都 TLE. 因此後續將 blocking factor 改成 16, 32, 並發現在改成 64 時 share memory 會不夠但是即使改成了 32 performance 還是有好幾筆 TLE, 後續跟同學討論後發現是因為我宣告了過多的 thread, 導致了 share memory 在 phase 2 時不夠, 所以去查找了 GPU 的 system spec 得到了以下的算式：

49152 (總 share memory 大小) / 4 (size of int) / 3 (在 phase 2 時每個thread 會需要三個 matrix) 開根號 得到 64

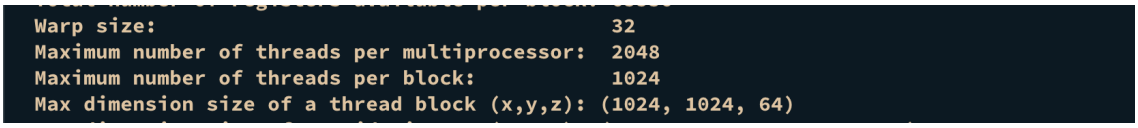
因此最後便將 blocking factor 訂為 64, 並調整其他的 parameter 讓 blocking factor = 64 可以運作

- **#blocks :**

  在每個 phase 時的 block 數量會不同, 基本上是依照 Block Floyed-Warshall 的設計來定義 block size, 因此在 phase 1 時會有 1 個 block, phase 2 時則會有 2 * blocking factor 個 block, phase 3 則會有 blocking factor * blocking factor 個 block.

- **#threads :**

  因為 GTX1080 的 maximum threads per block 是 1024, 但是因為 64 * 64 = 4096 (Block Floyed-Warshall 的每個 block 會被分配到 4096 個 edge), 因此需要讓每個 thread 計算四個 edge 才能符合 system spec. (但是為了讓每個 thread 計算四個 edge, 所以在 coding 上我的寫法是用 blocking factor / 4 作為 `threadIdx.y` value, 因此在 block factor 小於 64 時會有 thread redundancy 發生, 但是因為多數測資都大於 64, 所以整體的 performance 並不會有太大的影響)



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

  這裡我使用了 spec 所提供的 blocked Floyed-Warshall algorithm 作為實作的方法, 並以助教提供的 code 當作原始架構去做更改, 在架構上比較大的更改是為了最大化 block gpu 的 shared memory 使用, 設計上每個 thread 需要處裡四個 edge 的計算.

  - **Padding**

為了 coding 的方便性與 host memory to share memory 的搬移不會有 segmentation fault, 我將 adjacency matrix 的 column 與 row 增加成 block factor 的倍數.

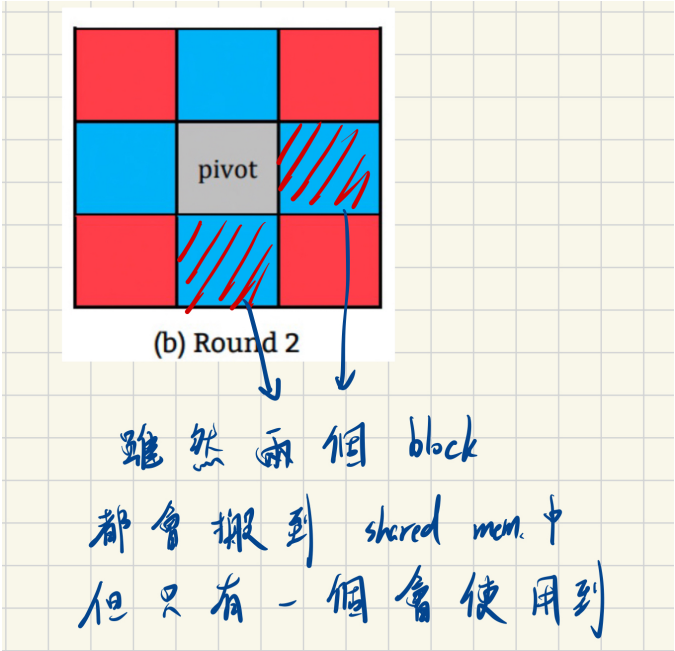
e.g. 如果 vertex = 5000, block factor = 64 → vertex 會被增加到 ceil(5000/64)*64 = 5056

- **Phase 1**

  如同 spec 提供的演算法所描述, phase 1 會對 pivot block 做 Floyed-Warshall 因為在我的邏輯裡 gpu 的 block 數量會對應到需要處理的 adjacency matrix block 數量, 所以只宣告了 1 個 block, 為了提升 performance, 我在一開始將 data 從 global memory 搬入到 shared memory 中, 用 __syncthreads 確保 shared memory 的資料寫入結束, 對這個 block 執行 Floyed-Warshall 後再將 data 搬入回 global memory.

- **Phase 2**

  phase 2 則是對 pivot block 所在的 column 與 row 的 block 做資料更新, 這個部分我宣告了 2 * matrix block 數量的 gpu block, 每個 block 使用 3 個 matrix block 大小的 shared memory. (這樣的設計其實會導致有一半的 thread 在做 redundancy 的計算, 但是在觀測過 profiler 的結果後我認為 performance 的 bottleneck 並不在這個部分, 因此我沒有對這個部分在做進一步的優化)



一開始與 phase 1 相同, 會先將 pivot block, 對應到的 row 與 col block 從 global memory 搬入到 shared memory, 並在搬移完成後開始做 blocked Floyed-Warshall 的更新, 再將更新完的資料從 shared memory 搬回到 global memory

- **Phase 3**

  phase 3 需要計算的則是剩餘的區域, 我開了 matrix block * matrix block 數量的 gpu block 去做計算, 每個 block 需要 2 * matrix block 的 share memory

  一開始會先將 pivot block 與自己所在的 matrix block 搬入到 shared memory 中, 並在更新 matrix block 的數值後再將資料從 shared memory 搬回到 shared memory 中.

HW3-3

- **How do you divide your data in hw3-3 ?**

  一開始我嘗試實作了將 phase1, 2, 3 都做 divide 計算後再 communicate, 會發現整體的 performance 反而下降了, 經過觀察後發現如果在 phase 1, 2 做 communication, communication cost 將會大於 calculation cost, 因此便決定只在 phase 3 做 multi GPU 的 parallel compute. 主要的做法是讓兩張卡都計算 phase 1 與 phase 2, 直到 phase 3 時將整個 matrix 平均切成兩塊(上下兩塊), 由 GPU 0 計算第一塊 (上方那塊), GPU 1 計算第二塊(下方那塊), 並在計算完後將兩張卡的結果做同步, 避免下一個 round 的 data 不同步.

- **What is your configuration in hw3-3 ? And why ? (e.g. blocking factor, #blocks, #threads)**

  整體的 blocking factor 與 #blocks 還有 #threads 基本上與 single GPU 的架構相同, 主要的差異是在 phase 3 的 #blocks, 因為在設計上分成了上下兩個部分, 為了要達到這樣的設計 phase 3 的 #blocks 需要除以 2. 因為剩下的內容並沒有太多的差異, 這裡就不贅述了.

- **How do you implement the communication in hw3-3**

  在溝通的部分, 為了避免在 phase 3 計算完後同步到錯誤的資料, 我使用了 `cudaDeviceSynchronize` 以及 `barrier` 來確保 GPU 的計算已經完成.

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

phase 1, 2, 3 我沿用了 hw3-2 的設計, 本來想要嘗試其他的設計方法, 但是礙於時間壓力最後並沒有更多的時間去想出新的方法, 因此最後還是使用了 hw3-2 的設計, 這裡就不再對 phase 1, 2, 3 的部分多做贅述

在 padding 的部分, 因為每個 gpu 的 blocking factor 都是 64, 為了避免分配時出現兩邊的 matrix size 無法被 64 整除的情況發生, 這裡的 padding 我選擇以 128 的倍數來做 padding, 舉例來說：

如果是原本的 padding : 如果 vertex = 5000, block factor = 64 → vertex 會被增加到 ceil(5000/64)*64 = 5056 → 5056/(2*64) = 39.5

如果是更新後的 padding : 如果 vertex = 5000, block factor = 64 → vertex 會被增加到 ceil(5000/128) * 128 = 5120 → 5120/(2*64) = 40

因為剩餘的設計部分都已經在上面的幾點描述, 這裡便不再贅述.

2. Profiling Results (hw3-2)

我跑了三筆測資的 phase3 profiling results, 分別是 p11k1, p15k1 與 p20k1, 想要看到在不同的測資下整體的 performance 是否會有差異

- p11k1

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "NVIDIA GeForce GTX 1080 (0)"					
Kernel: phase3(int*, int)					
172	achieved_occupancy	Achieved Occupancy	0.491875	0.492445	0.492228
172	sm_efficiency	Multiprocessor Activity	98.27%	98.40%	98.35%
172	shared_load_throughput	Shared Memory Load Throughput	2327.0GB/s	2562.5GB/s	2499.8GB/s
172	shared_store_throughput	Shared Memory Store Throughput	96.960GB/s	106.77GB/s	104.16GB/s
172	gld_throughput	Global Load Throughput	145.44GB/s	160.16GB/s	156.23GB/s
172	gst_throughput	Global Store Throughput	48.480GB/s	53.386GB/s	52.078GB/s
V: 11000, E: 505586					

- p15k1

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "NVIDIA GeForce GTX 1080 (0)"					
Kernel: phase3(int*, int)					
235	achieved_occupancy	Achieved Occupancy	0.492343	0.492665	0.492483
235	sm_efficiency	Multiprocessor Activity	98.40%	98.47%	98.43%
235	shared_load_throughput	Shared Memory Load Throughput	2473.5GB/s	2559.6GB/s	2524.5GB/s
235	shared_store_throughput	Shared Memory Store Throughput	103.06GB/s	106.65GB/s	105.19GB/s
235	gld_throughput	Global Load Throughput	154.59GB/s	159.97GB/s	157.78GB/s
235	gst_throughput	Global Store Throughput	51.531GB/s	53.324GB/s	52.595GB/s

- p20k1

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3(int*, int)
        140                         achieved_occupancy                        Achieved Occupancy    0.492206    0.492745    0.492505
        140                               sm_efficiency                   Multiprocessor Activity      98.37%      98.50%      98.45%
        140                      shared_load_throughput             Shared Memory Load Throughput  2273.7GB/s  2541.5GB/s  2468.3GB/s
        140                     shared_store_throughput            Shared Memory Store Throughput  94.738GB/s  105.89GB/s  102.85GB/s
        140                               gld_throughput                    Global Load Throughput  142.11GB/s  158.84GB/s  154.27GB/s
        140                               gst_throughput                   Global Store Throughput  47.369GB/s  52.947GB/s  51.423GB/s
```

可以看到整體的 shared memory 使用量非常的高, 但是無論在哪個情況下 shared memory 都有使用到 98% 以上.

另外可以看到其他的數據的跑分也十分的一致, 都不會超過 5% 的差異 , 代表我的程式在各個情況下的執行情形都很一致

3. Experiment & Analysis

System Spec

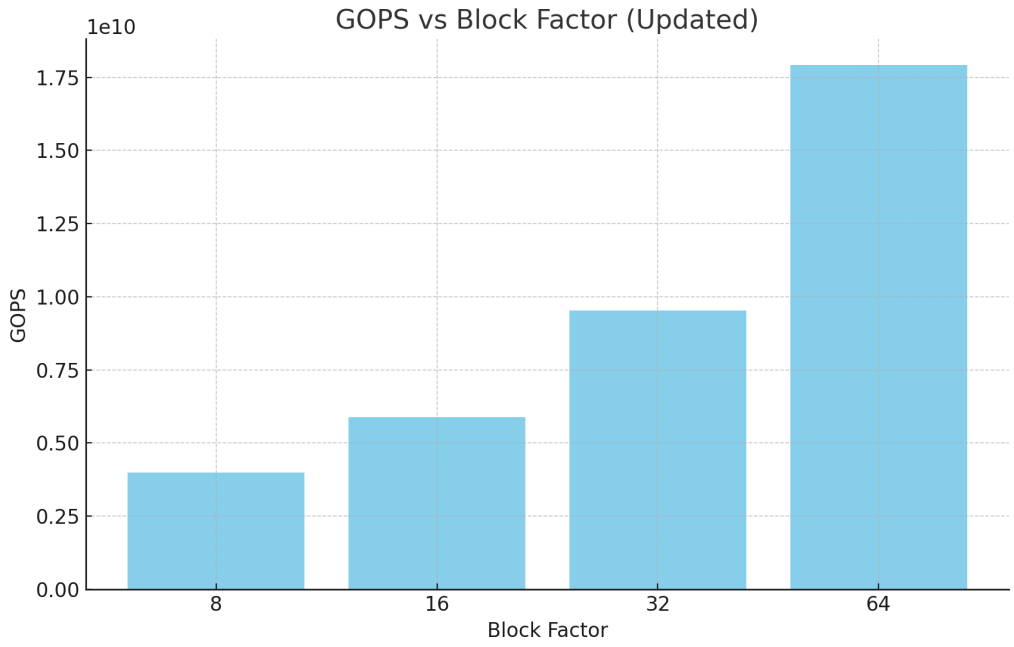
主要的實驗是執行在國網中心的 server 以及 hades cluster (based on hades 當下有多卡), 不過同一個 experiment 只會在同一個 cluster 上跑.

Blocking Factor (hw3-2)

因為 profiler 花費的時間比想像的還多, 所以這裡使用 c20.1 來做實驗, 我們並將實驗限縮在 phase 3 (比較好去計算時間), blocking factor 的部分則是用 8, 16, 32, 64 去觀測差異

- Integer GOPS 的部分, 我先取得 phase3 的 inst_integer 後, 再用除以運算時間來取得

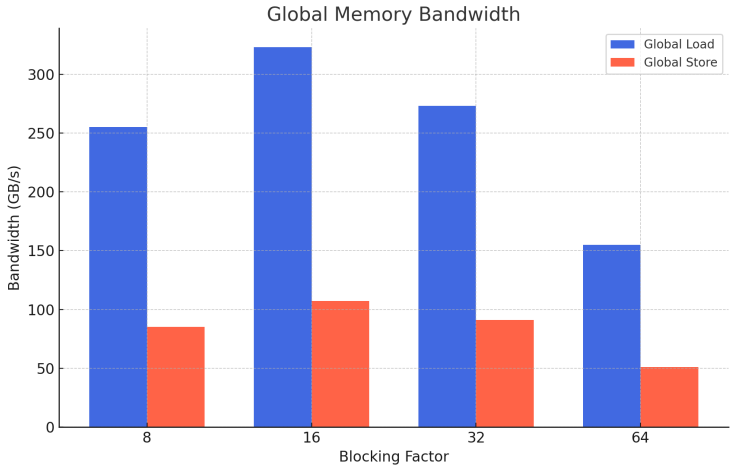
  以下是實驗結果：



可以從圖中觀測到 blocking factor 跟 GOPS 高度正相關, 幾乎是線性的在成長, 會有這麼高的相關性我的理解為隨著 blocking factor 越大, gpu 花費在 initialization 的時間與 switching 的時間會越來越少, 所以 performance 也會越好, 因此才會有更高的 GOPS.

- global memory bandwidth

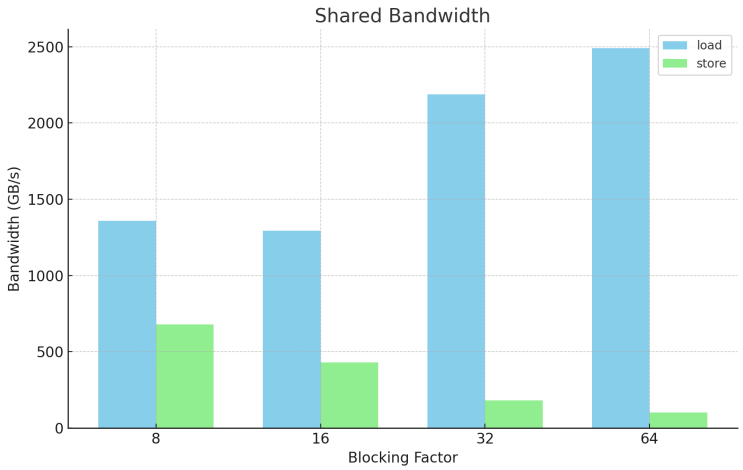
  以下是實驗結果：



可以看到 store 的數量整體而言很接近, 整體的 load throughput 也沒有很大的分歧, 這也跟程式的設計邏輯相符, 在設計上我們會希望 global memory 的 load 與 store 的數量越少越好, 最好只有在一開始時 load 進 share memory, 並在一且都計算完成後再 store 回 global memory, 因此這部分我認為我有好好優化了 global memory 的使用.

- shared memory bandwidth

  以下是實驗結果：



  可以看到, shared memory store operation throughput 會隨著 blocking factory 的上升而逐漸下降. 而我的 load 卻是隨著 blocking factor 而上升. 會有這個現象我認為十分的特殊, 因為隨著 load 與 store 應該要是正相關才對, 這也顯示了我的程式在優化上其實做的不是很理想, 才導致了我的 load throughput 與 store throughput 會有這麼大的分歧. 應該是之後可以在優化的點. (我後來思考會不會是因為我的程式撰寫是以 blocking factor = 64 為主要的優化方向, 反而造成了 shared memory store 的這樣的奇特現象? 如果未來有時間我應該會再去研究這裡的主要變因是什麼)

- 以下是跑分的實際數據 (只是為了記錄, 助教批改時可以直接跳過)

  64

```
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3(int*, int)
         79                             inst_integer                    Integer Instructions  2741528576  2741528576  2741528576
         79                    shared_load_throughput          Shared Memory Load Throughput  2436.0GB/s  2549.2GB/s  2491.4GB/s
         79                    shared_store_throughput         Shared Memory Store Throughput  101.50GB/s  106.22GB/s  103.81GB/s
         79                             gld_throughput                  Global Load Throughput  152.25GB/s  159.32GB/s  155.71GB/s
         79                             gst_throughput                 Global Store Throughput  50.750GB/s  53.108GB/s  51.904GB/s
```

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   80.42%  153.65ms       157  978.64us  964.46us  1.0027ms  phase3(int*, int)
```

32

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3(int*, int)
        157                            inst_integer                    Integer Instructions  1545204224  1545204224  1545204224
        157                    shared_load_throughput             Shared Memory Load Throughput  2148.8GB/s  2217.9GB/s  2189.1GB/s
        157                   shared_store_throughput            Shared Memory Store Throughput  179.07GB/s  184.83GB/s  182.42GB/s
        157                             gld_throughput                   Global Load Throughput  268.60GB/s  277.24GB/s  273.64GB/s
        157                             gst_throughput                  Global Store Throughput  89.533GB/s  92.414GB/s  91.212GB/s
```

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   79.91%  162.32ms        79  2.0547ms  1.9640ms  2.1657ms  phase3(int*, int)
```

16

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3(int*, int)
        313                            inst_integer                    Integer Instructions   947042432   947042432   947042432
        313                    shared_load_throughput             Shared Memory Load Throughput  1289.5GB/s  1302.3GB/s  1293.7GB/s
        313                   shared_store_throughput            Shared Memory Store Throughput  429.83GB/s  434.09GB/s  431.24GB/s
        313                             gld_throughput                   Global Load Throughput  322.37GB/s  325.57GB/s  323.43GB/s
        313                             gst_throughput                  Global Store Throughput  107.46GB/s  108.52GB/s  107.81GB/s
```

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   79.88%  161.10ms        79  2.0392ms  1.9251ms  2.1249ms  phase3(int*, int)
```

8

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase3(int*, int)
        626                            inst_integer                    Integer Instructions   650040032   650040032   650040032
        626                    shared_load_throughput             Shared Memory Load Throughput  1311.0GB/s  1368.4GB/s  1360.2GB/s
        626                   shared_store_throughput            Shared Memory Store Throughput  655.52GB/s  684.19GB/s  680.09GB/s
        626                             gld_throughput                   Global Load Throughput  245.82GB/s  256.57GB/s  255.04GB/s
        626                             gst_throughput                  Global Store Throughput  81.940GB/s  85.523GB/s  85.012GB/s
V: 5000, E: 7594839
```

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   80.02%  163.10ms        79  2.0646ms  1.9985ms  2.1278ms  phase3(int*, int)
```

Optimization (hw3-2)

在 Optimization 的部分,我主要做的優化有 :

1. Coalesced memory access
2. Shared memory
3. Large blocking factor
4. Reduce communication
5. Unroll

因為時間的緣故, 沒有辦法寫出這麼多個版本去跑 profiler, 所以這裡僅用文字描述在 coding 時的發現：

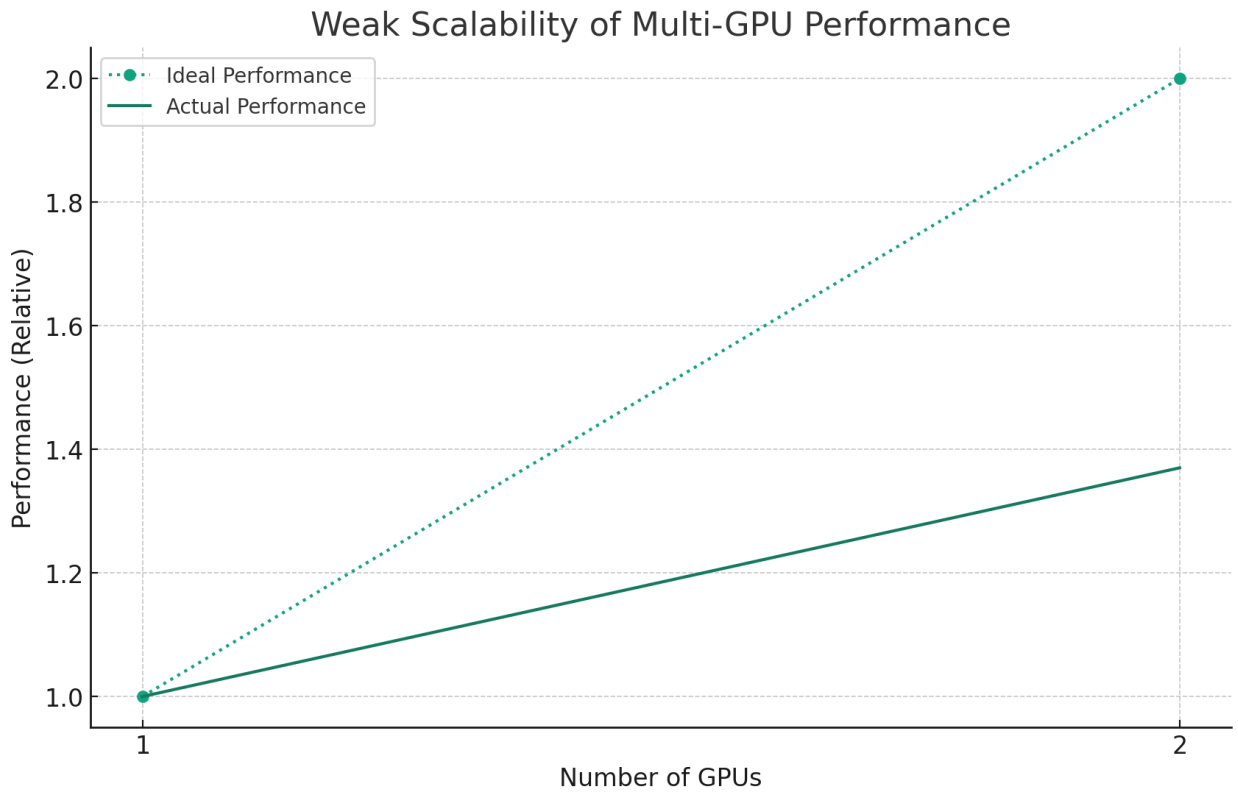
第一個版本時只有 Shared memory, 第一版的 code 只能通過 hw3-2 的 correctness

第二個版本則加入了 coalesced memory access 後, 整體的 performance 沒有太大程度的上升, 還是只能通過 correctness, 整體沒有太大的差異

第二個版本則加入了 Large blocking factor, 將 blocking factor 從 8 增加到 64, 整體的 performance 增加了非常多, 幾乎所有的測資都通過了, 但是也遇到了有一筆 correctness 突然就過不了了, 我認為應該是在 blocking factor 增加的過程中發生了 race condition, 但是最後還是找不到哪裡發生這個事件, 因此

Weak scalability(hw3-3)

在 scalability 的部分, 我做的實驗檢視了在相同的測資下 hw3-2 與 hw3-3 的 performance difference. 在理想的情況下理論上 hw3-3 的 phase 3 的計算速度應該要是 hw3-2 的兩倍, 但是在實驗過後發現當加入了 barrier 後整體的 communication overhead 會讓 performance 掉到大約 1.3 - 1.4 (1.37) 倍, 也可以看到為什麼 communication 會是 distributed learning 中一大研究部分.



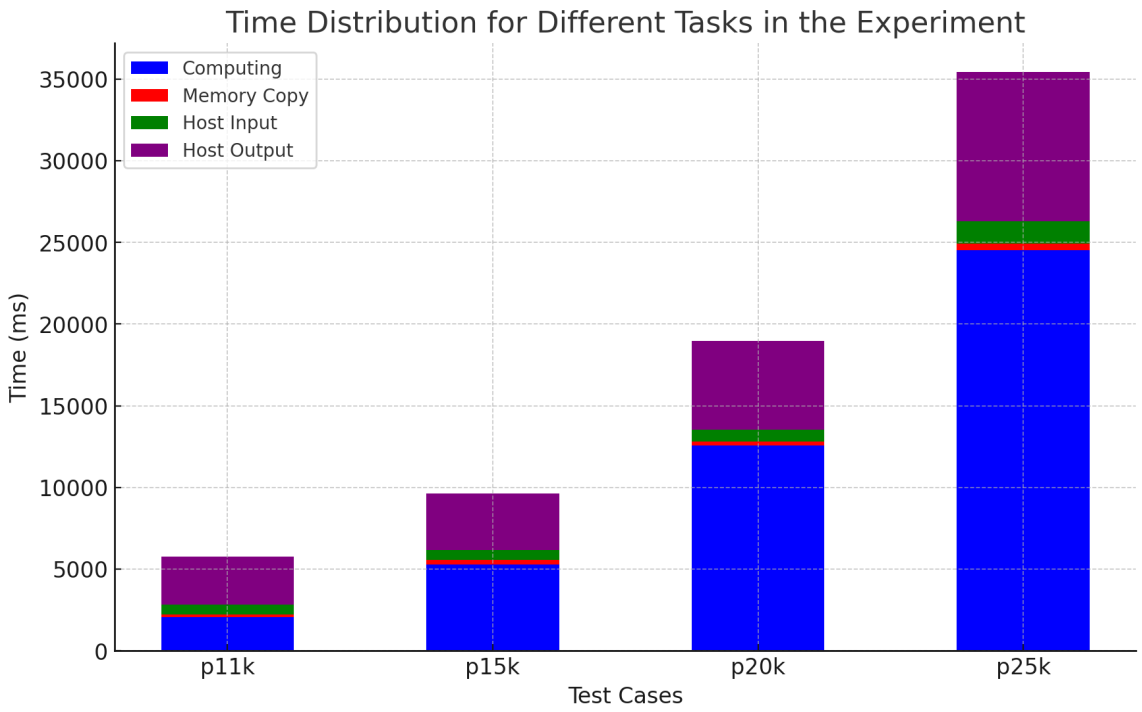
Time Distribution (hw3-2)

我執行了四筆測資 (p11k1, p15k1, p20k1, p25k1)來觀測時間分佈的變化, 實驗的 blocking factor 都是 64.

首先是各個測資的 vertex 與 edge 數：

	#vertex	#edge
p11k1	11000	505586
p15k1	15000	5591272
p20k1	20000	264275
p25k1	25000	5780158

以下是實驗結果：



可以看到整體而言 computing time 的花費時間佔比是最高的, Output 的時間也有明顯的上升, 但是不像 compute time 的上升程度那麼明顯.

另外可以看到 vertex 數量與整體的計算數量呈正相關(跟理論相符, 因為 floyed-warshall 的計算本身並不會考量到 edge 的數量), 我們以 p11k 與 p25k 為例, 雖然 vertex 數量只增加了兩倍, 但是整體運算量增加了 12 倍.

Others

在寫 hw3-2 時有發現一件有趣的事, 如果下了 `#pragma unroll` 其實對整體的 performance 不會有太多的改善, 以下是我把 phase 3 的 for loop 的 `#pragma unroll` 刪除後得到的結果：

```
Time(%)      Time     Calls       Avg       Min       Max  Name
 97.20%  19.6656s       391  50.296ms  47.982ms  51.906ms  phase3(int*, int)
```

以下是當我把所有的 `#pragma unroll` 刪除之後所得到的結果：

```
Time(%)      Time     Calls       Avg       Min       Max  Name
 95.50%  20.1052s       391  51.420ms  49.201ms  52.872ms  phase3(int*, int)
```

可以看到整體的表現幾乎沒有差異, 可能的原因為這次作業的程式的 for 回圈本身相對簡單, 所以 compiler 在一開始就將其優化掉了, 因此才會有此現象.

4. Experience & conclusion

What have you learned from this homework?

這份作業主要學到了 cuda code 的實作, 雖然一開始就知道 cuda code 的難度很高, 所以已經比以往更提前來撰寫這份作業, 但是完成這份作業的時間還是超過我所預期的時間, 在撰寫作業中比較令我印象深刻的事為一開始我以為 shared memory 只能宣告一維, 所以花費了大量的時間在做 index translation.

另一件讓我印象深刻的事情則是在 hw3-2 的部分, 當我把 block factor 提高到 64 後, 我的 C17 就會變成 wrong answer, 因為只有這筆測資會 wrong answer, 又 wrong answer 只會發生在 block factor 提高到 64 的時候, 我認為有兩種可能 :

1. 這筆測資不知怎麼的會導致 race condition, 導致了其 output 與預期不相符
2. 這筆測資會使用過多的 shared memory 導致當資料搬入回 global memory 時會有錯誤.

在我跑了 cuda-memcheck 後我確定這筆測資沒有 shared memory error, 但是在 race condition 的部分因為其他比測資都沒有發生問題, 我單獨跑了好幾次這比測資也沒有新的變化發生, 因此我最終還是沒有找到問題的答案, 這也讓我體驗了 cuda code 的 debug 過程有多絕望.

b. Feedback (optional)

spec 在分數的解釋好像需要更新了, 像是 performance 的部分提到的測資還是 40 筆, 但是 scoreboard 上其實沒有這麼多筆測資.