CS542000 Parallel Programming HW3 Report

- 學號：108022138
- 姓名：楊宗諺

Implementation

CPU Version (hw3-1)

在 CPU version 的實作上，我使用的是一般的 Floyd-Warshall algorithm 以及 OpenMP 的 threading library，在初始距離的賦值以及每個 intermediate point 的 iteration 前面加上了 omp parallel 的 compiler directive 來平行 for loop 的計算工作。在 directive 的選擇上，因為這邊可以平行的 for loop 都為兩層，所以選擇了 `collapse(2)` 以及 guided 的 scheduling method 作為 directive。

```
#pragma omp parallel for schedule(guided, 1) num_threads(thread_num) collapse(2)
```

One GPU Version - Blocked Floyd-Warshall Algorithm (hw3-2)

- Padding

  由於 input vertices 的數量並不一定會整除預設的 blocking factor，所以可能會造成之後的計算上需要多做 boundary 的檢查。為了避免該情況，這邊我利用了 padding 將 input vertices 的數量增加至可以整除 blocking factor 的數量，雖然這麼做會讓 dist 陣列變大，但相對來說節省掉了之後計算上要做 boundary check 的時間，同時也避免了 warp divergence 的發生。

- Settings of Blocking Factor and Kernels

  由於 Blocked Floyd-Warshall algorithm 有三個主要的 phases 且每個 phase 都和其前面的 phases 具有 data dependency，所以這邊我設定了三個 kernels 來分別負責這三個 phases 的計算，每個 kernel 所分配到的 thread block 數量取決於其負責的 phase 所需要計算的區塊數量。舉例來說，phase 1 只需要計算 pivot block，所以負責執行 phase 1 的 kernel 只有一個 thread block，而 phase 2 需要計算 pivot row 以及 pivot column 上所有的 blocks，所以負責執行 phase 2 的 kernel 會有 `2 * (N - 1)` 個 thread blocks，這邊 N 為 `ceil(V, B)`，而最後負責執行 phase 3 的 kernel 則會有 `(N - 1) * (N - 1)` 個 thread blocks，也就是負責剩下所有的計算區塊。

  此外，這邊我也設定了每個 thread block 的 thread 數量為 32 * 32 = 1024 個（GPU 環境的上限），同時也設定了 blocking factor 為 64，代表說每個 thread 會需要計算四個格點。

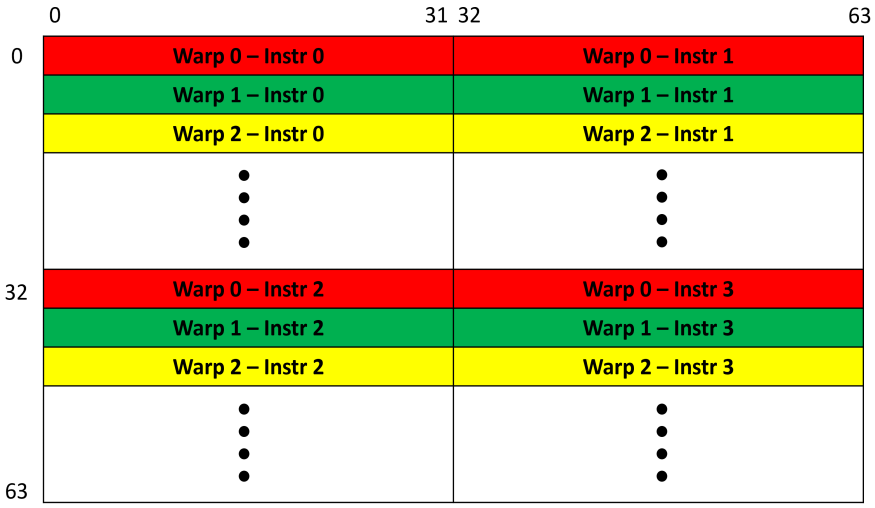
  ```
  const int B = 64;
  const int N = ceil(V, B);

  dim3 grid1(2, N - 1);
  dim3 grid2(N - 1, N - 1);
  dim3 block(32, 32);

  for (int r = 0; r < N; ++r) {
  ```

```
    phase1<<<1, block>>>(dist_d, r);
    phase2<<<grid1, block>>>(dist_d, r);
    phase3<<<grid2, block>>>(dist_d, r);
}
```

- Optimization Techniques
  - Coalesced Memory Access and Use of Shared Memory

    在 global memory access 上，這邊我有採用了 coalesced access 的 pattern 來讓處於同一個 warp 裡的 threads 在執行 global memory access 的相關操作時能夠對連續的記憶體空間打 request，一次讀出或寫入連續的 data，進而達到較好的效能。

    舉例來說，如果 blocking factor 為 64 且 thread block 的大小為 32 * 32，這代表說總共會有 32 個 warps 且每個 warp 都會執行四條 memory access 的指令，那麼 coalesced access 的 pattern 就會如下圖所示：



    此外，觀察演算法我們還可以發現不管在哪個 phase，每個 block 的計算都只會取決於特定某幾塊 blocks，而且會重複地去 access data。舉例來說，phase 1 每個 block 的計算基本上只取決於自己，phase 2 的計算取決於 pivot block 以及自己，而 phase 3 的計算則是取決於 pivot row 以及 pivot column 上對應的兩個 blocks。

    基於這樣的 access pattern，在計算工作開始之前可以將隨後計算會重複 access 到的 data 先從 global memory 搬到 shared memory，這樣之後對於 memory 的操作就可以直接 access 較快的 shared memory，而不是相對來說慢很多的 global memory。

    此外，這邊同時也體現出了設定 blocking factor 為 64 的好處是我們可以在 phase 2 和 phase 3 的時候大致上填滿 shared memory 的空間（GPU 環境的上限為 48 KB/block），進而達到較好的資源使用率。

```
// Implementation of coalesced memory access and shared memory in phase
1 for example
int x = threadIdx.y, y = threadIdx.x;
int i = x + (r << 6), j = y + (r << 6);
```

```
__shared__ int dist_shared[4096];
dist_shared[(x << 6) + y] = dist[i * nV_d + j];
dist_shared[(x << 6) + y + 32] = dist[i * nV_d + j + 32];
dist_shared[((x + 32) << 6) + y] = dist[(i + 32) * nV_d + j];
dist_shared[((x + 32) << 6) + y + 32] = dist[(i + 32) * nV_d + j + 32];
```

- Loop Unrolling

  在每個 phase 負責的 iteration 前，我有加上 loop unrolling 的 compiler directive 來去盡量減少 branch 預測錯誤所帶來的 overhead 以及提高指令級的平行度，進而提升效能。

Two GPUs Version - Blocked Floyd-Warshall Algorithm (hw3-3)

在兩顆 GPU 的控制上，這邊我是使用了 OpenMP 的 threading library 來生成兩個 worker threads 分別負責兩顆 GPU 的執行。

而演算法的部分，基本上 two GPUs version 和 one GPU version blocked Floyd-Warshall algorithm 的 implementation 大致上相同，唯一不同的點在於我讓 two GPUs version 中的兩顆 GPU 分別持有上半部以及下半部的 data，並且分別負責兩個部份的計算。然而從計算的 pattern 來看，在三個 phases 的計算上，兩顆 GPU 彼此會有 data dependency，也就是說一顆 GPU 需要的 data 可能會在另一顆 GPU 上，而這個需要的 data 基本上就是 pivot row。

因此，在每個 round 開始之前，我有先透過 device to device memory copy 的方式將該 round 所需要計算的 pivot row 從其中一顆 GPU 搬到另一顆 GPU 上。這樣在 round 開始之後，兩顆 GPU 就可以獨立且正確地完成兩個部份的計算。

Profiling Results（hw3-2）

這邊我使用的測資為 `p20k1`，以下是針對負責 phase 3 的 kernel 利用 Nvidia profiling tools 所測出來的結果。

Metrics	Min	Max	Avg
Occupancy	0.921782	0.922755	0.922174
SM efficiency	99.95%	99.96%	99.96%
Shared memory load throughput	3450.6GB/s	3479.1GB/s	3465.9GB/s
Shared memory store throughput	143.77GB/s	144.96GB/s	144.41GB/s
Global memory store throughput	215.66GB/s	217.44GB/s	216.62GB/s
Global memory store throughput	71.887GB/s	72.480GB/s	72.206GB/s

Experiment & Analysis

Methodology

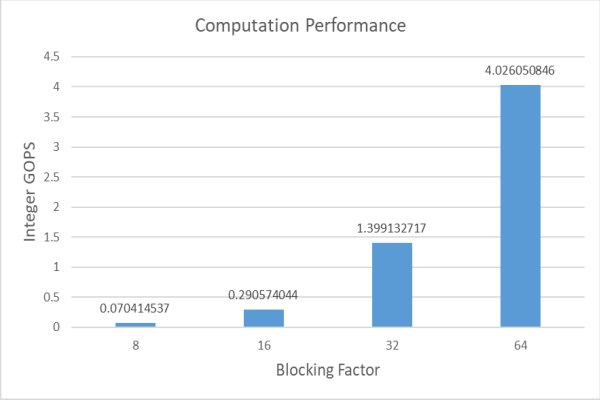
以下實驗採用的系統配置為課堂所提供的 hades server。在手動計算執行時間的部分，我主要是使用 `std::chrono::steady_clock::now()` 去夾在需要計算時間的區塊來獲得時間差並加總。

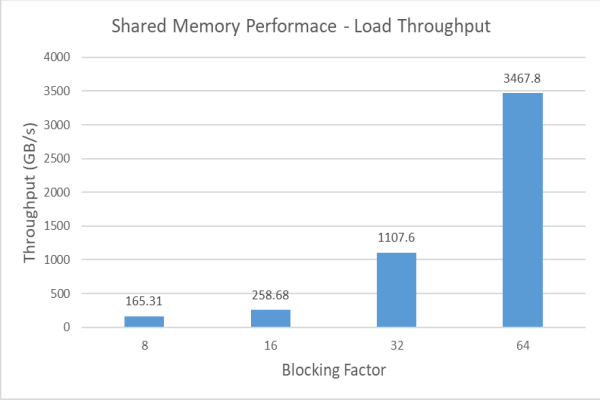
Profile

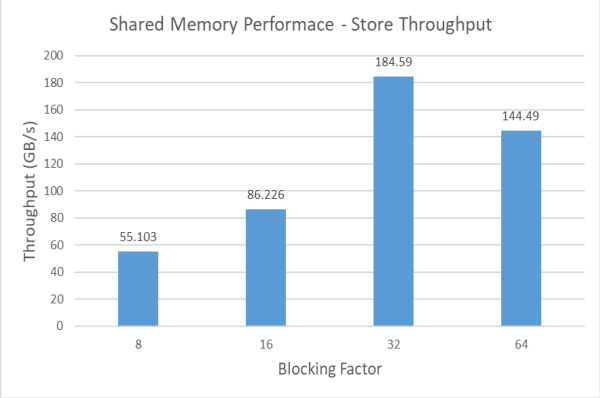
Testing Different Blocking Factors

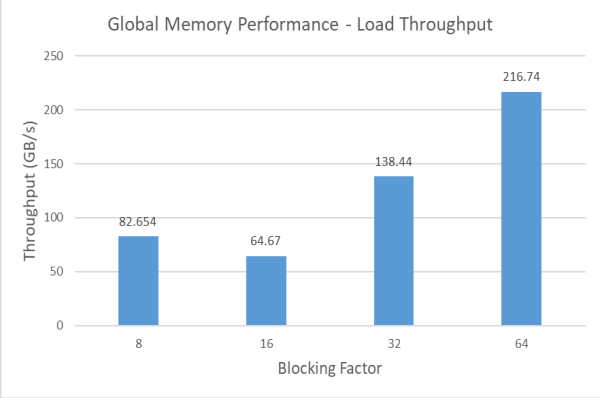
這邊我使用的測資為 p11k1，並且針對以下幾種 metrics 利用 Nvidia profiling tools 來做實驗。

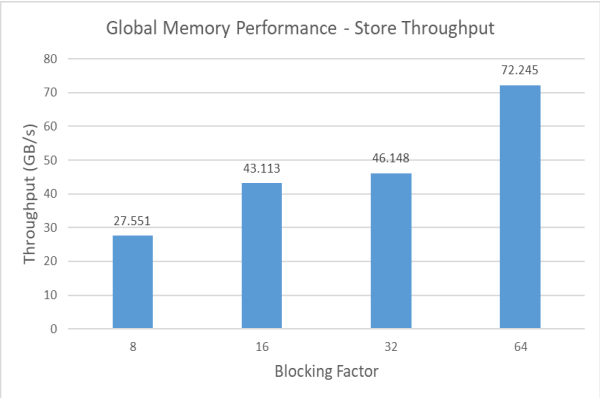
- IGOPS (Integer Giga Operations Per Second)
- Shared memory load throughput
- Shared memory store throughput
- Global memory load throughput
- Global memory store throughput









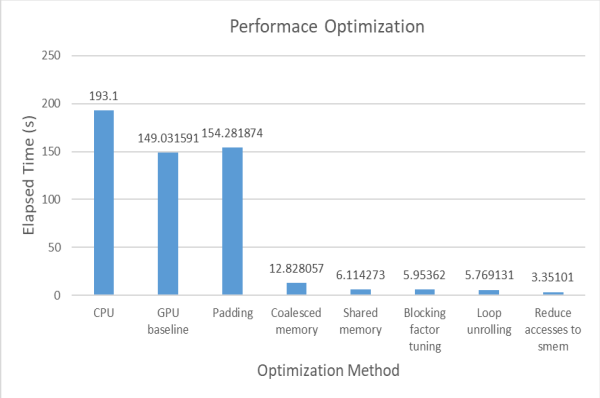


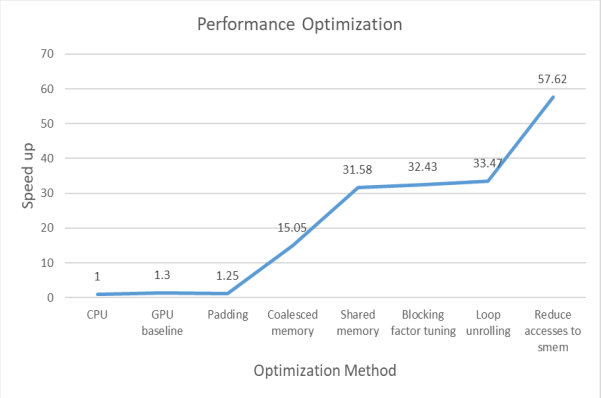
從以上結果可以看出，設定 blocking factor 為 64 相對其他大小的 blocking factor 來說，在 computation 以及 global & shared memory load/store 的 performance 上都表現得相對較好，這說明了此設定在 computation 以及 memory 的使用上是個相對較好的選擇。

Testing Several Optimization Techniques

這邊我使用的測資為 p11k1，並且針對以下 methods 來作 optimization。

- GPU baseline
- Padding
- Coalesced memory access
- Shared memory
- Blocking factor tuning
- Loop unrolling
- Remove redundant accesses to shared memory





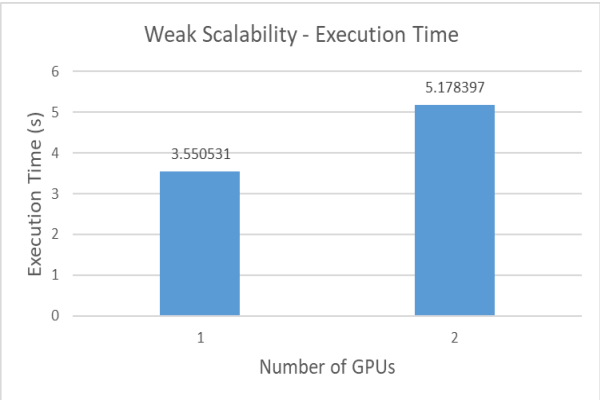
透過以上幾種優化技巧的使用，從圖中可以發現整體 program 的執行效率有了接近 60x 的提速，而其中也可以觀察到針對 global & shared memory 做的優化所帶來的提速相對來說十分地顯著，這同時也說明了對於提升 cuda program 的效能來說，memory 的管控是十分關鍵的。

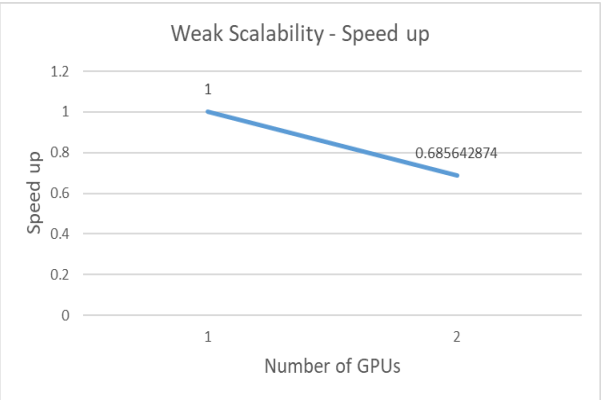
Weak Scalability

在 Weak scalability 的實驗上，針對 One GPU version 的測試我主要是使用測資 `p11k1`，而針對 Two GPUs version 的測試我則是使用自己產生的測資，兩筆測資的 configurations 如下：

Testcases	Input Size	Edges
p11k1 (for One GPU version)	11000	505586
Self-generated testcase (for Two GPUs version)	13859	505586

其中 Self-generated testcase 的 input size 是由兩倍的 11000^3 開三次方根得到，edges 的數量則是設定和 `p11k1` 相同，而以下是實際針對兩個版本所測量到的結果。





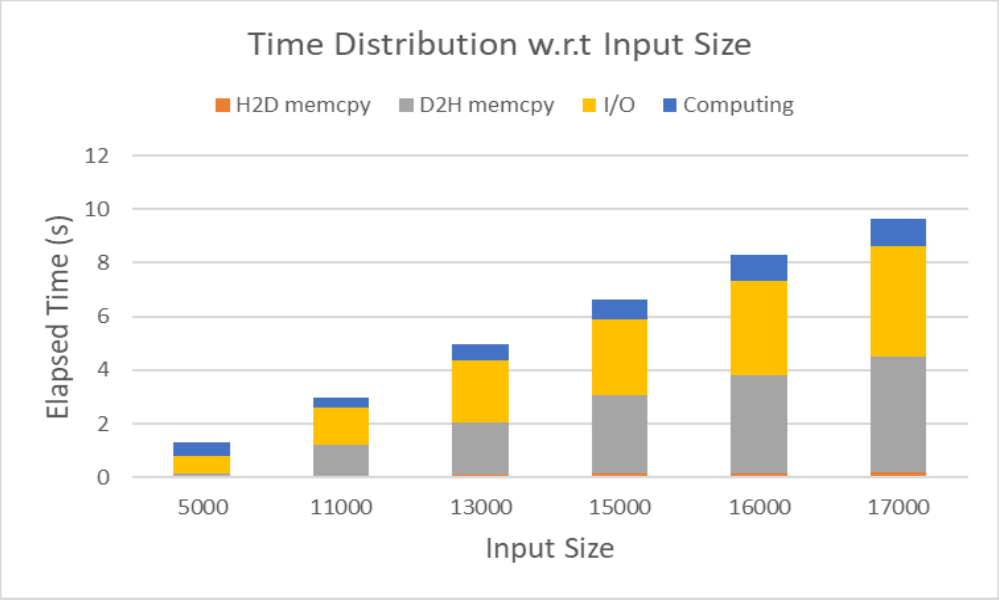
從以上結果可以看出，整體 program 的 weak scalability 離 perfect linear scaling 還有段距離，其中有個主要原因是在於 Two GPUs version 的每個 round 開始之前，其中一顆 GPU 都必須透過 device to device 的溝通方式

將 pivot row 傳給另一顆 GPU，另一顆 GPU 才可以獨立且正確地完成計算，無形地增加了溝通的時間成本，因而導致整體 program 沒辦法達到 perfect linear scaling。

Time Distribution

以下是我針對不同 input size 所做的實驗，使用的測資為 `c21.1`、`p11k1`、`p13k1`、`p15k1`、`p16k1` 以及 `p17k1`。而在計算 Computing time 上面，我是將總執行時間減掉 host 和 device 之間 memcpy 的時間以及 I/O 的時間來獲取大概的值。

Testcases	Input Size	Edges	H2D	D2H	I/O	Computing
c21.1	5000	10723117	0.016784s	0.131218s	0.640524s	0.512629s
p11k1	11000	505586	0.075898s	1.153144s	1.386033s	0.378542s
p13k1	13000	1829967	0.107907s	1.927981s	2.306138s	0.636250s
p15k1	15000	5591272	0.139579s	2.917929s	2.817620s	0.747433s
p16k1	16000	5444739	0.161455s	3.662603s	3.490268s	0.968693s
p17k1	17000	4326829	0.182423s	4.323537s	4.100939s	1.046810s



從以上結果可以看出隨著 input size 的增加，處理的資料量以及計算量變多，導致以上實驗 metrics 所測量到的時間也都隨之增加。然而，其中可以觀察到雖然 H2D 以及 D2H data transfer 的量在相同的 input size 下相等，但是 D2H memcpy 和 H2D memcpy 所需的時間可能存在差異。根據測資 `p17k1` 所測出來的結果可以發現，這兩個 metrics 在 input size N = 17000 的時候速度差異甚至來到了將近 24x，這反映出了 H2D 和 D2H 這兩個 data transfer 方向的 bandwidth 並不完全相同，H2D data transfer 的 bandwidth 可能相對來說大於 D2H data transfer 的 bandwidth。

Experience & Conclusion

這次作業其實還滿有趣的，其一是因為之前沒有實作過 Blocked Floyd-Warshall algorithm，再者因為這也算是我第一次寫如此複雜的 cuda program。在嘗試通過所有測資的時候，我試了滿多上課有提到的優化技巧，像是 coalesced memory access、shared memory、bank conflict avoidance 以及改變 coding style 來避免 warp

divergence 的發生，而其中從 profiling 的結果來看，我發現針對 global & shared memory 做的優化所帶來的提速十分地顯著，也因此通過了許多先前會超時的測資。

而透過此次作業的實作，我想我有更加了解如何利用 profiling tools 來找到整體 cuda program 的 bottlenecks 以及針對那些 bottlenecks 利用上課所教的技巧來進行優化，進而達到顯著的提速，希望之後還有機會可以嘗試撰寫更多不同種類的 cuda program。