Parallel Programming Hw3 Report

11208530余雪淩

(需要在淺色模式下閱讀，圖表的文字才看得到)

1. Implemention

a. HW3-1(CPU version)

使用Floyd-Warshall演算法，演算法的核心是逐步加入中繼點k，然後更新每個頂點間的最短距離，這樣的操作會有三層迴圈，其中 `k` 為中繼點，而 (i) 和 (j) 分別為起點和終點。

```
for (int k = 0; k < V; k++){
    for (int i = start; i < end; i ++) {
        for (int j = 0; j < V; j++) {
            D[i][j] = std::min(D[i][j], D[i][k] + D[k][j]);
        }
    }

    pthread_barrier_wait(&barrier);
}
```

本程式使用pthread來平行化演算法，具體方法如下：

(1) Data Division

- 程式根據process數量 `ncpus` ，將 `V` 個節點分為 `ncpus` 個區段，每個process處理一部分的 `i` 範圍。
- `start` 和 `end` 分別是每個process負責的 `i` 範圍起始和結束的index。

```
int tid = *(int *)args;
int start = tid * (V / ncpus) + std::min(tid, V % ncpus);
int end = start + V / ncpus + (tid < V % ncpus);
```

(2) 同步機制

- 在加入中繼點 `k` 時，所有process需要同步，確保當前的 `k` 更新完成後，才能繼續下一個 `k` 。
- 因此需要使用 `pthread_barrier_wait` 來確保當前第k輪算完後才會進入下一輪。

---

b. HW3-2(Single GPU version)

(1) Data Division

- 將 n*n 的矩陣分割成 B*B 的block，`B`為 Blocking Factor。
- 每個 B*B 的區塊可以看成一個小矩陣，會分別放到 GPU 的 **shared memory** 進行計算，以減少讀取記憶體的延遲。
- **Padding**：為了確保 n 是 B 的倍數，如果 (n % B != 0)，會額外補0進行 padding。

(2) CUDA Configuration

```
Device : "NVIDIA GeForce GTX 1080"
Total amount of constant memory:                65536 bytes
Total amount of shared memory per block:        49152 bytes
Total number of registers available per block: 65536
Warp size:                                      32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:            1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

Thread number

從Device Query中得知每個GPU block最多有 (32 * 32 = 1024) 個threads，因此threads number就設為32。

Blocking Factor

從Device Query中得知shared memory最大只有49152 bytes，一個int大小為4 bytes，因為計算需要(後續會介紹)，最多總共會用到3個share memory，3 * 64 * 64 * 4 = 49512 因此將Blocking Factor`B`設為64剛好可以有效利用所有memory，此時每個thread一次會處理64 * 64 / 32 * 32 = 4筆資料。
由於threads還是只有32 * 32個，因此share memory每次在讀取資料時，一次讀取Half Block`HB`的資料以加快運算。

```
整體的 B = 64 區塊分割成 4 個 HB = 32 半區塊
-------------------------------------
|        HB1        |        HB2        |
|    (32 x 32)      |    (32 x 32)      |
-------------------------------------
|        HB3        |        HB4        |
|    (32 x 32)      |    (32 x 32)      |
-------------------------------------
每個 thread 負責 4 個元素，分佈在不同的 HB 子區塊內
```

```
// load data from golbal memory to shared memory
// execute 32*32 data per block(32*32 threads in it)
shared_D[y][x] = d_Dist[global_y*pitch+ global_x];
shared_D[y + HB][x] = d_Dist[(global_y + HB) * pitch+ global_x];
```

```
shared_D[y][x + HB] = d_Dist[global_y * pitch+ global_x + HB];
shared_D[y + HB][x + HB] = d_Dist[(global_y + HB) * pitch+ global_x + HB];
```

(3) Implementation

Phase 1: Pivot Block

- Grid : 1 * 1
- 處理 B*B 的Pivot Block，並執行Floyd Warshall演算法更新距離。

1. 先將資料載入share memory，並利用 `__syncthreads()` 確保所有threads完成資料加載：

```
shared_D[y][x] = d_Dist[global_y*pitch+ global_x];
shared_D[y + HB][x] = d_Dist[(global_y + HB) * pitch+ global_x];
shared_D[y][x + HB] = d_Dist[global_y * pitch+ global_x + HB];
shared_D[y + HB][x + HB] = d_Dist[(global_y + HB) * pitch+ global_x + HB];
__syncthreads();
```

2. 執行Floyd Warshall演算法，這裡為了避免if-else產生，一律使用cuda min：

```
for(int i = 0; i < B; i++) {
    shared_D[y][x] = min(shared_D[y][x], shared_D[y][i] + shared_D[i][x]);
    shared_D[y + HB][x] = min(shared_D[y + HB][x], shared_D[y + HB][i] + shared_D
    shared_D[y][x + HB] = min(shared_D[y][x + HB], shared_D[y][i] + shared_D[i][
    shared_D[y + HB][x + HB] = min(shared_D[y + HB][x + HB], shared_D[y + HB][i]

    __syncthreads();
}
```

3. 執行完成後將資料載回Global memory

```
// shared memory --> global memory
d_Dist[global_y * pitch+ global_x] = shared_D[y][x];
d_Dist[(global_y + HB) * pitch+ global_x] = shared_D[y + HB][x];
d_Dist[global_y * pitch+ global_x + HB] = shared_D[y][x + HB];
d_Dist[(global_y + HB) * pitch+ global_x + HB] = shared_D[y + HB][x + HB];
```

Phase 2 (Row and Column Blocks)

- Grid : BlockNum * 1 (BlockNum = n/B)
- 使用已計算的Pivot Block (Phase 1)來更新該列和該行的其他blocks，每個thread負責與pivot block同行及同列的各一個block。

1. 加載 Pivot、Row、Column blocks

```
// pivot block from phase1
int global_x = x + id * B;
int global_y = y + id * B;
```

```
__shared__ int pivotD[B][B];
pivotD[y][x] = d_Dist[global_y*pitch+ global_x];
pivotD[y + HB][x] = d_Dist[(global_y + HB) * pitch+ global_x];
pivotD[y][x + HB] = d_Dist[global_y * pitch+ global_x + HB];
pivotD[y + HB][x + HB] = d_Dist[(global_y + HB) * pitch+ global_x + HB];

// load the target block of same column into shared memory
int i = y + id * B;
int j = x + blockIdx.x * B;

__shared__ int ColD[B][B];
ColD[y][x] = d_Dist[i * pitch+ j];
ColD[y + HB][x] = d_Dist[(i + HB) * pitch+ j];
ColD[y][x + HB] = d_Dist[i * pitch+ (j + HB)];
ColD[y + HB][x + HB] = d_Dist[(i + HB) * pitch+ (j + HB)];

// load the target block of same row
i = y + blockIdx.x * B;
j = x + id * B;

__shared__ int RowD[B][B];
RowD[y][x] = d_Dist[i * pitch+ j];
RowD[y + HB][x] = d_Dist[(i + HB) * pitch+ j];
RowD[y][x + HB] = d_Dist[i * pitch+ (j + HB)];
RowD[y + HB][x + HB] = d_Dist[(i + HB) * pitch+ (j + HB)];
```

2. 執行Floyd Warshall:

```
for (int k = 0; k < B; ++k) {
    // using cuda min
    ColD[y][x] = min(ColD[y][x], pivotD[y][k] + ColD[k][x]);
    ColD[y + HB][x] = min(ColD[y + HB][x], pivotD[y + HB][k] + ColD[k][x]);
    ColD[y][x + HB] = min(ColD[y][x + HB], pivotD[y][k] + ColD[k][x + HB]);
    ColD[y + HB][x + HB] = min(ColD[y + HB][x + HB], pivotD[y + HB][k] + ColD[k]

    RowD[y][x] = min(RowD[y][x], RowD[y][k] + pivotD[k][x]);
    RowD[y + HB][x] = min(RowD[y + HB][x], RowD[y + HB][k] + pivotD[k][x]);
    RowD[y][x + HB] = min(RowD[y][x + HB], RowD[y][k] + pivotD[k][x + HB]);
    RowD[y + HB][x + HB] = min(RowD[y + HB][x + HB], RowD[y + HB][k] + pivotD[k]
}
```

Phase 3 (Remaining Blocks)：

- Grid：BlockNum * BlockNum
- 使用Phase 1、2算完的blocks來更新所有Remaining Blocks。
  從Row和Column中讀取資料，進行矩陣相加並更新最短路徑。

```
shared_D[y][x] = d_Dist[global_y*pitch+ global_x];
shared_D[y + HB][x] = d_Dist[(global_y + HB) * pitch+ global_x];
shared_D[y][x + HB] = d_Dist[global_y * pitch+ global_x + HB];
shared_D[y + HB][x + HB] = d_Dist[(global_y + HB) * pitch+ global_x + HB];
```

```
int i = y + id * B;
int j = x + blockIdx.x * B;

__shared__ int ColD[B][B];
ColD[y][x] = d_Dist[i * pitch+ j];
ColD[y + HB][x] = d_Dist[(i + HB) * pitch+ j];
ColD[y][x + HB] = d_Dist[i * pitch+ (j + HB)];
ColD[y + HB][x + HB] = d_Dist[(i + HB) * pitch+ (j + HB)];

i = y + blockIdx.y * B;
j = x + id * B;

__shared__ int RowD[B][B];
RowD[y][x] = d_Dist[i * pitch+ j];
RowD[y + HB][x] = d_Dist[(i + HB) * pitch+ j];
RowD[y][x + HB] = d_Dist[i * pitch+ (j + HB)];
RowD[y + HB][x + HB] = d_Dist[(i + HB) * pitch+ (j + HB)];
__syncthreads();
```

Floyd Waarshall:

```
for (int k = 0; k < B; ++k) {
    shared_D[y][x] = min(shared_D[y][x], RowD[y][k] + ColD[k][x]);
    shared_D[y + HB][x] = min(shared_D[y + HB][x], RowD[y + HB][k] + ColD[k][x]);
    shared_D[y][x + HB] = min(shared_D[y][x + HB], RowD[y][k] + ColD[k][x + HB]);
    shared_D[y + HB][x + HB] = min(shared_D[y + HB][x + HB], RowD[y + HB][k] + C
}
```

---

c. HW3-3(Multiple GPU version)

使用openmp開兩個threads，分別操作兩個GPU，基本上與單GPU版本差異不大，以下將針對不一樣的地方做介紹：

(1)Data Division

- 第一個 GPU 處理前半資料，第二個 GPU 處理後半資料。
- OpenMP 通過 `omp_get_thread_num()` 獲取threads編號，根據第一個或第二個 GPU 設定offset處理區域，具體的 offset 也根據block分割進行分配，如遇到n/B=奇數，則第二個GPU多算一筆資料。

```
#pragma omp parallel num_threads(2)
{
    int threadID = omp_get_thread_num();
    cudaSetDevice(threadID);
    dim3 grid3(BlockNum, BlockNum/2);

    int offset;
    if(threadID == 0) offset = 0; else offset = BlockNum/2;
    if(threadID == 1 && (BlockNum % 2 == 1)) grid3.y++;
//...
```

(2) Implementation

由於phase3計算量最大，因此這裡將phase3分成上下兩半，分別給兩個GPU運算，每個GPU各需計算 $\frac{(V/B-1)^2}{2}$ 的資料量，其餘與單GPU版本沒有差異

```
dim3 grid3(BlockNum, BlockNum/2);
Phase3 <<<grid3, NumofThreads>>> (d_Dist[threadID], id, n, offset);
```

(3) Communication

- **Device 與 Host 之間:**
  使用 cudaMemcpy 在 Host 與 Device 之間傳遞資料，為減少傳輸量，僅傳輸該GPU需要的資料區塊。

```
cudaMemcpy(d_Dist[threadID] + id * B * n, Dist + id * B * n,
           B * n * sizeof(int), cudaMemcpyHostToDevice);
```

- **雙 GPU :**
  以**Host**作為中繼站: 在一個 GPU 更新完後，將結果傳回Host，另一個 GPU 再從Host讀取更新的區塊。

```
for(int id = 0; id < BlockNum; id++){
    if (id >= offset && id < offset + grid3.y) {
        cudaMemcpy(Dist + id * B * n,d_Dist[threadID] + id * B * n,
                   B * n * sizeof(int), cudaMemcpyDeviceToHost);
    }
    #pragma omp barrier
    if (id < offset || id >= offset + grid3.y) {
        cudaMemcpy(d_Dist[threadID] + id * B * n, Dist + id * B * n,
                   B * n * sizeof(int), cudaMemcpyHostToDevice);
    }

    Phase1 <<<grid1, NumofThreads>>> (d_Dist[threadID], id, n);
    Phase2 <<<grid2, NumofThreads>>> (d_Dist[threadID], id, n);
    Phase3 <<<grid3, NumofThreads>>> (d_Dist[threadID], id, n, offset);
}
```

以`id`代表當前輪次，如果不屬於當前 GPU 的責任範圍（>=`offset`）時，從Host將其他GPU已計算的區塊複製到當前GPU，作為其下一步計算的依據，若屬於當前 GPU 的負責區域，則將 GPU 中計算的結果複製回Host Memory。

---

2. Profiling Results (hw3-2)

使用p11k1測資，因為phase3 kernel loading最重，所以挑這個來觀察：

Shared Memory Throughput

Kernel	Metric Name	Min	Max	Avg
Phase3	Shared Load Throughput	3255.9GB/s	3326.7GB/s	3297.4GB/s
	Shared Store Throughput	265.79GB/s	271.56GB/s	269.18GB/s

Other Metrics

Metric Name	Min	Max	Avg
Achieved Occupancy	0.921288	0.923417	0.922308
SM Efficiency	99.86%	99.92%	99.90%

---

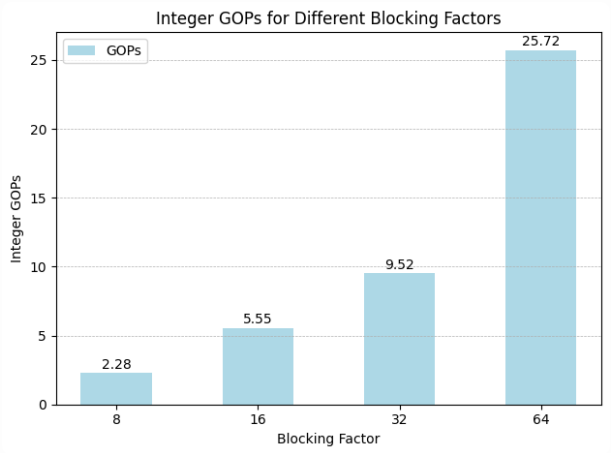
3. Experiment & Analysis

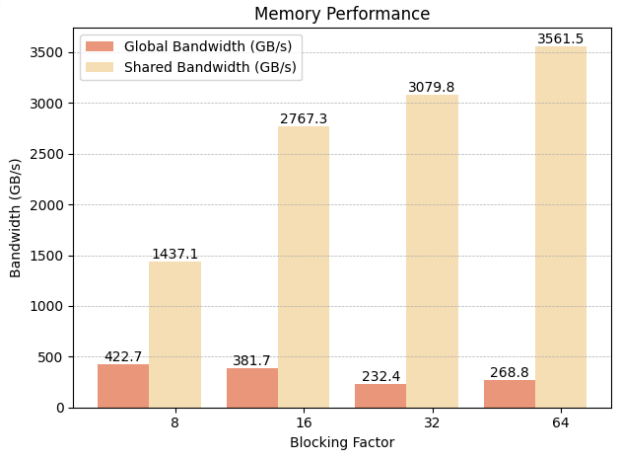
a. System Spec

使用課程所提供的Apollo GPU server

b. Blocking Factor (hw3-2)

考量到profiling時間，這邊主要使用c21.1測資+nvprof分析。

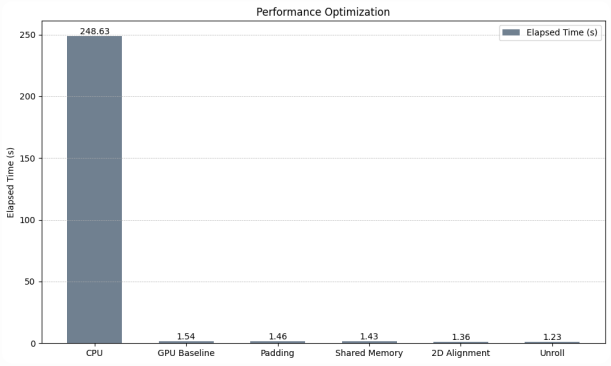


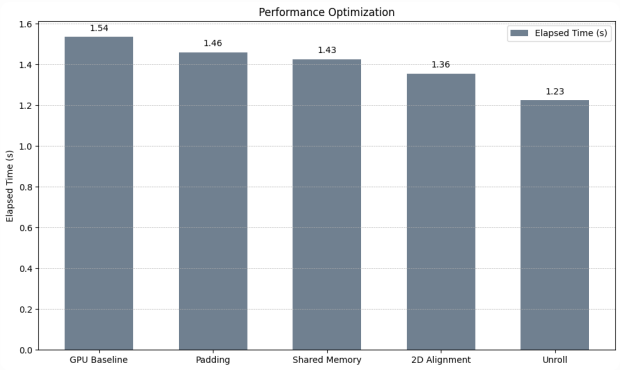


- **Shared Memory** 的Bandwidth遠高於 **Global Memory**，代表程式處理大部分資料還是位於share memory，有效減少讀取Global Memory的次數，提升效率。
- Shared Memory隨著 Blocking Factor 增加而穩定提升，這是因為Blocking Factor 較大時，更多資料可以平行處理，增加了Shared Memory的使用率。
- 而當Blocking Factor增加時，Global Bandwidth反而較不穩定，甚至在32、64的時候有所下降，可能因為較大的 Blocking Factor導致Thread Blocks 同時大量存取Global Memory造成記憶體壅塞（Memory Contention），導致bandwidth下降。

c. Optimization (hw3-2)

考量到profiling時間，這邊主要使用c21.1測資+nvprof分析。





可以看到相較於CPU，GPU有將近200倍的加速。針對GPU的優化中，每個版本均有逐步加速。可以看到經過Padding對齊數據後，首先提升了Memory存取效率，接下來改為share memory後，減少了Global Memory的存取次數，因此也提升了運算速度。再經過Cuda 2D Alignment的記憶體對齊後，減少了地址計算開銷，最後加上unroll展開迴圈後，相較於baseline有大約20.1%的速度提升，大幅提升運算效率。

c. Weak scalability (hw3-3)

由於 **Weak Scalability** 的要求是計算量與計算資源量成比例增加，也就是說當使用兩張 GPU 運算時，其計算量應該是一張 GPU 的兩倍。
本作業中，計算量的大小取決於V，並與 V^2 成正比。因此，為了滿足這個條件，選用了以下兩筆測資進行比較：

1. c19.1 : V = 2100
2. c18.1 : V = 3000

$$\frac{\mathrm{V}_{18}^2}{\mathrm{V}_{19}^2} = \frac{3000^2}{2100^2} \approx 2.04$$

Results:

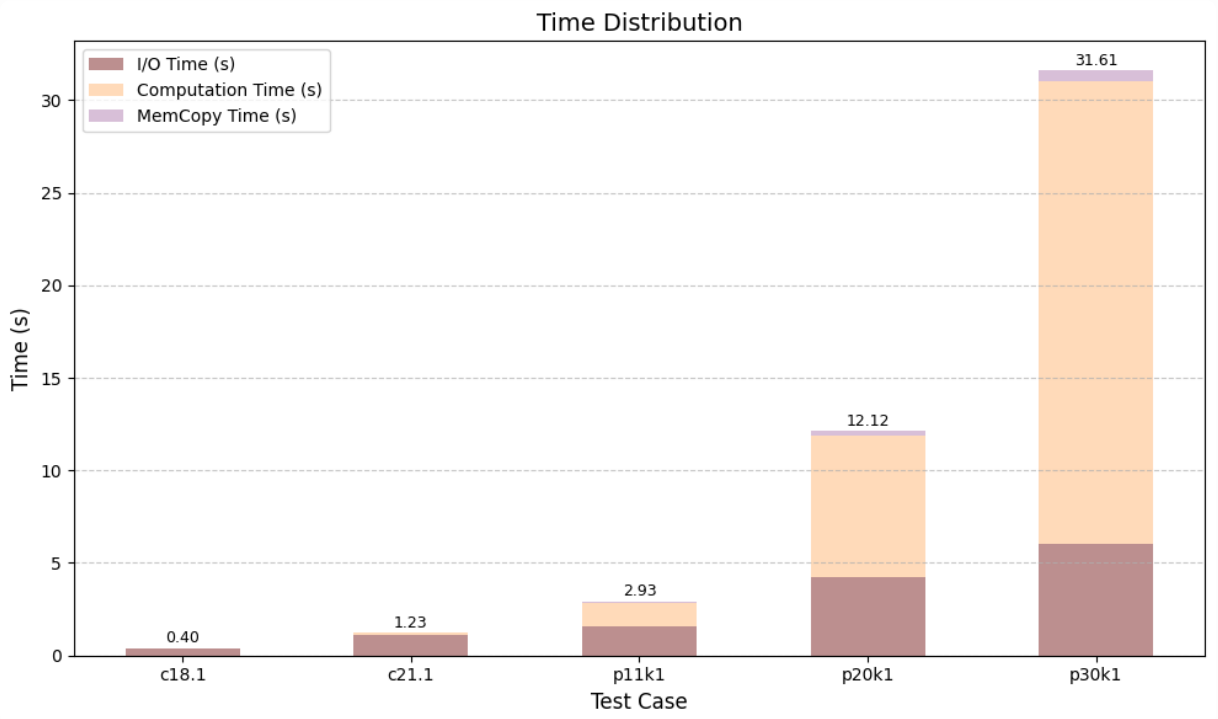
GPU	Computation Time	Overall time
1 GPU	1.2148 ms	439.43 ms
2 GPUs	43.900 ms	491.03 ms

- Computation Time在雙GPU的情況下顯著增加，由於每個phase間都有data dependency，在雙GPU的運行下勢必要進行多次GPU間的溝通，大幅增加了計算時間。
- 在Overall time方面，雖然雙 GPU 的執行時間略有增加，但增長並不多，可能在I/O或CUDA memcpy的開銷上，雙GPU有助於加速這部分的效率。

e. Time Distribution (hw3-2)

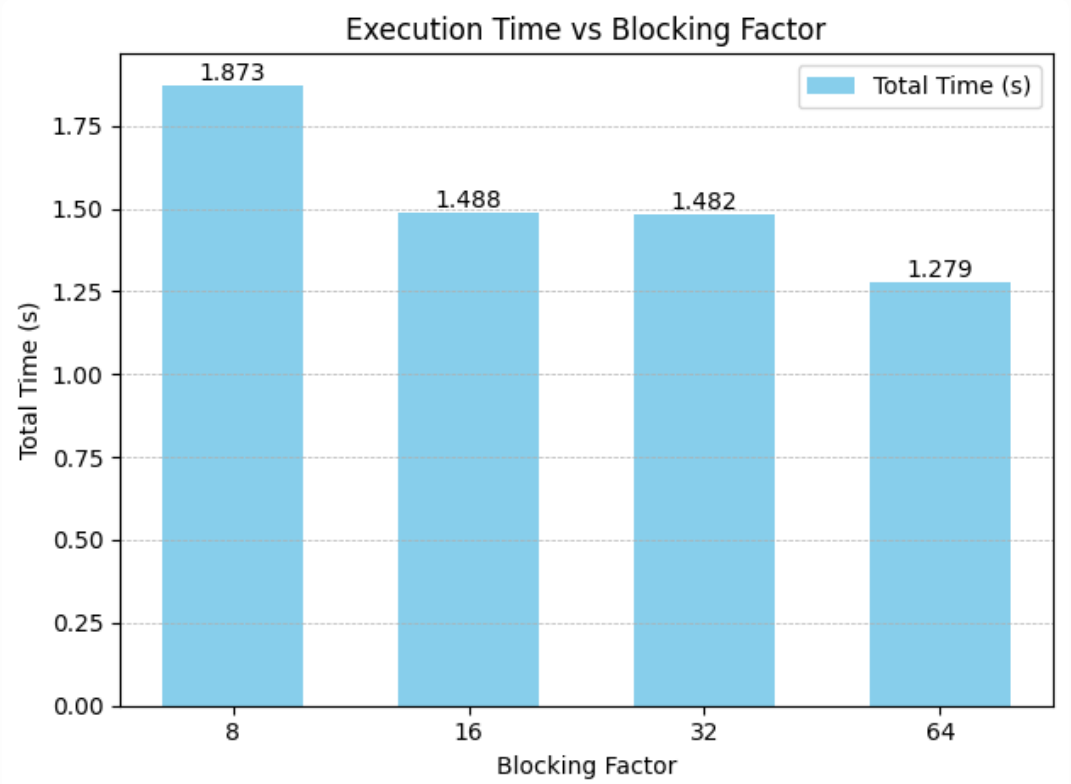
使用以下測資進行測試，並使用NVTX、nvprof獲取不同部分的運行時間

Test Case	V	I/O Time (ms)	Computation Time (ms)	Memcopy Time (ms)
c18.1	3000	355.91	33.578	5.7511
c21.1	5000	1085.39	129.090	16.2464
p11k1	11000	1561.45	1289.220	76.7240
p20k1	20000	4244.85	7622.290	254.0600
p30k1	30000	6044.98	24996.100	567.2100



可以看到隨著測資的size增加，計算時間的增加幅度最明顯，顯示計算的複雜度與測資規模成非線性關係。相比之下，I/O Time 和 MemCopy Time的增加幅度較為平緩，這表示程式的主要瓶頸可能來自於運算過程，尤其在phase3的運算量最大，較大的測資確實會大幅增加運算時間。

f. Others



上圖為不同blocking factor下總運行時間的差異，可以看到較大的blocking factor確實可以有效提升程式的運行效率。

不同Unroll的比較

使用的Blocking factor為64，所以在Floyd-Warshall迴圈計算中照理說會跑64，直觀情況下unroll次數也會設為64。

```
// B = 64
#pragma unroll 64
for(int i = 0; i < B; i++) {
    shared_D[y][x] = min(shared_D[y][x], shared_D[y][i] + shared_D[i][x]);
    shared_D[y + HB][x] = min(shared_D[y + HB][x], shared_D[y + HB][i] + shared_
    shared_D[y][x + HB] = min(shared_D[y][x + HB], shared_D[y][i] + shared_D[i][
    shared_D[y + HB][x + HB] = min(shared_D[y + HB][x + HB], shared_D[y + HB][i]

    __syncthreads();
}
```

Unroll	Overall time
32	1.28927s
48	1.24134s
64	1.37020s

實際以c21.1測試後，unroll 32次反而比64快一些，在試了其他數字後，48反而是最快的，推測原因可能如下：

1. 記憶體存取效率
當迴圈完全展開（#pragma unroll 64）時，會在編譯期間生成更多指令來處理每個迭代步驟，導致增加register pressure。當register不足時，程式會使用慢速的Global Memory作為暫存區，導致性能下降。

2. 同步開銷（Synchronization Overhead）
__syncthreads() 需要在迴圈中同步所有threads的資料，每次迴圈展開增加一個同步點，當展開過多時，同步開銷會增大。

---

4. Experiment on AMD GPU

用hipify將cuda轉成hip程式後，使用C內建的 `gettimeofday` 來計算整體運行時間。以下為c21.1測資在兩種GPU上的比較，可以看到AMD較CUDA快一些。

GPU	Overall time
NVIDIA	1.246 s
AMD	0.878 s

5. Experience & conclusion

這次作業花了超多時間，大概是所有作業中最久的一次，不只有三種程式要寫，在分析數據及寫report也花了很多時間，但寫完這份作業對CUDA的運用又了解得更透徹了，也體會到GPU優化需要考慮很多細節，算是一次非常扎實的訓練。