hw3 All-Pairs Shortest Path

ID: 112062520 / Name: 戴維恩

Implementation

algorithm used in hw3-1

在 hw3-1 中，我選擇用 Blocked Floyd-Warshall Algorithm 作為實作 OpenMP thread 平行化的基礎。使用助教提供的 sequential templatel，並在內層迴圈加上 `#pragma omp parallel for collapse(2)` 如下所示：

```
// outer loop ...
    #pragma omp parallel for collapse(2)
    for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
        for (int j = block_internal_start_y; j < block_internal_end_y; ++j) {
            if (Dist[i][k] + Dist[k][j] < Dist[i][j]) {
                Dist[i][j] = Dist[i][k] + Dist[k][j];
            }
        }
    }
```

divide data / configuration in hw3-2

在 hw3-2 中，我總共用了三個 kernel function: `phase1_kernel` / `phase2_kernel` / `phase3_kernel`，分別去進行每個 round 的三個 phase。由於 `apollo GPU` 最多能夠 launch $32 * 32$ 個 thread，並且 shared memory 最多 $48kB$ ，因此 `phase1_kernel` 會由 $32 * 32$ 為 block dimension，並且每個 thread 做四個位置的 data loading 以及計算 (因為 shared memory 最多可以放 $64 * 64$ 的資料)。

三個 kernel launch 的方式如下：

```
// TS = 32
// BS = 64
int round = ceil_div(n, BS);
dim3 block_dim(TS, TS);
dim3 grid_dim_p2(round - 1, 2);
dim3 grid_dim_p3(round - 1, round - 1);
for (int r = 0; r < round; ++r) {
    phase1_kernel<<<1, block_dim>>>(d_Dist, n, r);
    phase2_kernel<<<grid_dim_p2, block_dim>>>(d_Dist, n, r);
    phase3_kernel<<<grid_dim_p3, block_dim>>>(d_Dist, n, r);
}
```

Phase 1

這個階段只需要做這個 round 的 pivot block，因此只需要一個 $32 * 32$ 的 thread block 即可。每個 thread 先 load 各自對應的資料位置，可以根據目前的 round 知道目前要做的 pivot block global index，再加上 thread 各自的 index offset 就可以做座標的 mapping：

```
int il = threadIdx.x;
int jl = threadIdx.y;
int p = Round * BS;
int i = p + il;
int j = p + jl;
```

每個 thread load 4筆資料進 shared memory，這裡有加 memory padding 以防止 bank conflict：

```
__shared__ int shared[BS][BS + 1];
shared[il][jl] = d_Dist[i_n + j];
shared[il+half][jl] = d_Dist[i_half_n + j];
shared[il][jl+half] = d_Dist[i_n + (j+half)];
shared[il+half][jl+half] = d_Dist[i_half_n + (j+half)];
```

進行 4 筆資料的運算，並在 for 迴圈結束後將 shared memory 中的結果存回 global。這裡有做 loop unroll 以提高效能：

```
#pragma unroll
for (int k = 0; k < BS; k++) {
    __syncthreads();
    shared[il][jl] = min(shared[il][jl], shared[il][k] + shared[k][jl]);
    shared[il+half][jl] = min(shared[il+half][jl], shared[il+half][k] + shared[k
    shared[il][jl+half] = min(shared[il][jl+half], shared[il][k] + shared[k][jl+
    shared[il+half][jl+half] = min(shared[il+half][jl+half], shared[il+half][k]

}
d_Dist[i_n + j] = shared[il][jl];
d_Dist[i_half_n + j] = shared[il+half][jl];
d_Dist[i_n + (j+half)] = shared[il][jl+half];
d_Dist[i_half_n + (j+half)] = shared[il+half][jl+half];
```

Phase 2

這個階段要做 pivot block 的那個 row 以及 column，我將 row 與 column 的計算寫在同一個 kernel中，用 index mapping 的方式決定哪些 thread 要做 row 哪些做 column，如下所示，若 block index y 為 0 則做 column，為 1 則做 row：

```
// at kernel launch
dim3 grid_dim_p2(round - 1, 2);

// in the kernel function
int is_row = blockIdx.y;
```

每個 thread 根據各自要做的 pivot row / column 去 load 資料，由於 phase 2 做一個 block 會需要用到已經算好的 pivot block 的資料，因此一個 thread load $4 * 2$ 筆資料。這裡的 shared memory 也有做 padding 以防 bank conflict：

```
int i = (is_row * p) + (!is_row * (bid * BS)) + il;
int j = (!is_row * p) + (is_row * (bid * BS)) + jl;
int i_n = i * n;
int i_half_n = (i + half) * n;


__shared__ int shared_p[BS][BS + 1];
__shared__ int shared[BS][BS + 1];

shared[il][jl] = d_Dist[i_n + j];
shared[il+half][jl] = d_Dist[i_half_n + j];
shared[il][jl+half] = d_Dist[i_n + (j+half)];
shared[il+half][jl+half] = d_Dist[i_half_n + (j+half)];

shared_p[il][jl] = d_Dist[(p + il) * n + (p + jl)];
shared_p[il+half][jl] = d_Dist[(p + (il+half)) * n + (p + jl)];
shared_p[il][jl+half] = d_Dist[(p + il) * n + (p + (jl+half))];
shared_p[il+half][jl+half] = d_Dist[(p + (il+half)) * n + (p + (jl+half))];
__syncthreads();
```

最後進行計算並將資料存回 global memory：

```
#pragma unroll
for (int k = 0; k < BS; k++) {
    if (is_row) {
        shared[il][jl] = min(shared[il][jl], shared_p[il][k] + shared[k][jl]);
        shared[il+half][jl] = min(shared[il+half][jl], shared_p[il+half][k] + sha
        shared[il][jl+half] = min(shared[il][jl+half], shared_p[il][k] + shared[
        shared[il+half][jl+half] = min(shared[il+half][jl+half], shared_p[il+hal
    } else {
        shared[il][jl] = min(shared[il][jl], shared[il][k] + shared_p[k][jl]);
        shared[il+half][jl] = min(shared[il+half][jl], shared[il+half][k] + share
        shared[il][jl+half] = min(shared[il][jl+half], shared[il][k] + shared_p[
        shared[il+half][jl+half] = min(shared[il+half][jl+half], shared[il+half]
    }
}

d_Dist[i_n + j] = shared[il][jl];
d_Dist[i_half_n + j] = shared[il+half][jl];
d_Dist[i_n + (j+half)] = shared[il][jl+half];
d_Dist[i_half_n + (j+half)] = shared[il+half][jl+half];
```

Phase 3

這個階段是要計算除了 pivot row / column 以外的所有 block，會是程式計算量最大的一個階段。由於扣掉不需要計算的部分，總共需要計算的 block 數量會是 $(round - 1)^2$，因此 kernel launch 分配如下：

```
dim3 grid_dim_p3(round - 1, round - 1);
```

phase 3 每個 block 的計算會需要用到這個 block 對應到的 pivot row 跟 column 的那兩個 block，也就是在 phase 2 算完的部分，因此需要做 index mapping 把各 thread 負責的兩個對應 block 資料 load 進 shared memory，這裡的 shared memory 是一個大 1D array，前半放 row block 後半放 column block （為了優化執行時間）：

```
shared[row_offset + il * BS + jl] = d_Dist[i_n + (p + jl)];
shared[row_offset + (il+half) * BS + jl] = d_Dist[i_half_n + (p + jl)];
shared[row_offset + il * BS + (jl+half)] = d_Dist[i_n + (p + (jl+half))];
shared[row_offset + (il+half) * BS + (jl+half)] = d_Dist[i_half_n + (p + (jl+hal

shared[col_offset + il * BS + jl] = d_Dist[(p + il) * n + j];
shared[col_offset + (il+half) * BS + jl] = d_Dist[(p + (il+half)) * n + j];
shared[col_offset + il * BS + (jl+half)] = d_Dist[(p + il) * n + (j+half)];
shared[col_offset + (il+half) * BS + (jl+half)] = d_Dist[(p + (il+half)) * n + (
__syncthreads();
```

接著開始進行計算，一樣也是每個 thread 負責四個位置的資料計算。這裡用也有用 loop unroll 提升程式效率：

```
int current_0 = d_Dist[i_n + j];
int current_1 = d_Dist[i_half_n + j];
int current_2 = d_Dist[i_n + (j+half)];
int current_3 = d_Dist[i_half_n + (j+half)];

#pragma unroll
for (int k = 0; k < BS; k++) {
    current_0 = min(current_0,
        shared[row_offset + il * BS + k] +
        shared[col_offset + k * BS + jl]);
    current_1 = min(current_1,
        shared[row_offset + (il+half) * BS + k] +
        shared[col_offset + k * BS + jl]);
    current_2 = min(current_2,
        shared[row_offset + il * BS + k] +
        shared[col_offset + k * BS + (jl+half)]);
    current_3 = min(current_3,
        shared[row_offset + (il+half) * BS + k] +
        shared[col_offset + k * BS + (jl+half)]);
}

d_Dist[i_n + j] = current_0;
d_Dist[i_half_n + j] = current_1;
d_Dist[i_n + (j+half)] = current_2;
d_Dist[i_half_n + (j+half)] = current_3;
```

divide data / configuration / communication in hw3-3

由於 hw3-3 是 multi-gpu 版本，因此要想辦法將計算量分散到兩張 gpu 上，計算完再進行整合。對於 phase 1, 2 我的作法沒有太大的改變；因為計算量以 phase 3 最大，因此主要的變動是在 phase 3 的部份。

phase 3 要去計算除了 pivot row / column 以外的 $(round-1)^2$ 個 block，因此我以 row 為單位將 block 數量拆成兩半，總共有 $(round-1)$ 個 row 的 block 需要計算，gpu 0 會分到前半，gpu 1 分到後半，餘數交給 gpu 1：

```
#pragma omp parallel num_threads(2)
{
    int id = omp_get_thread_num();
    int round = ceil_div(n, BS);
    int begin_id = (round / 2) * id;
    int chunk = round / 2;
    if (id) chunk += round % 2;
    // kernel launch ...
}
```

由於每個 round 的計算都會需要用到正確、計算好的 pivot row 資料，因此每 round 開始時，負責計算 pivot row 的 gpu 要將正確資料傳給 另外一張 gpu，這裡使用 `cudaMemcpyPeer` 進行溝通：

```
for(int r = 0; r < round; r++){
    if (r >= begin_id && r < begin_id + chunk)
        cudaMemcpyPeer(d_Dist[!id] + (r * n * BS), !id, d_Dist[id] + (r * n * BS
    #pragma omp barrier
    phase1_kernel <<<1, block_dim>>> (d_Dist[id], n, r);
    phase2_kernel <<<grid_dim_p2, block_dim>>> (d_Dist[id], n, r);
    phase3_kernel <<<grid_dim_p3, block_dim>>> (d_Dist[id], n, r, begin_id);
}
```

Profiling Results (hw3-2)

```
pp24s057@apollo-login:~/hw3-2$ srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --metrics achieved_occupancy,sm_efficiency,gld_throughput,gst_throug
hput,shared_load_throughput,shared_store_throughput ./hw3-2 testcases/c20.1 c20.1.out
==2582885== NVPROF is profiling process 2582885, command: ./hw3-2 testcases/c20.1 c20.1.out
==2582885== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==2582885== Profiling application: ./hw3-2 testcases/c20.1 c20.1.out
==2582885== Profiling result:
==2582885== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase2_kernel(int*, int, int)
         79                    achieved_occupancy                              Achieved Occupancy    0.954880    0.974587    0.963201
         79                          sm_efficiency                         Multiprocessor Activity      91.19%      95.26%      92.92%
         79                          gld_throughput                          Global Load Throughput  41.468GB/s  48.799GB/s  47.231GB/s
         79                          gst_throughput                         Global Store Throughput  20.734GB/s  24.400GB/s  23.615GB/s
         79                  shared_load_throughput                  Shared Memory Load Throughput  2011.2GB/s  2366.8GB/s  2290.7GB/s
         79                 shared_store_throughput                 Shared Memory Store Throughput  1368.4GB/s  1610.4GB/s  1558.6GB/s
    Kernel: phase3_kernel(int*, int, int)
         79                    achieved_occupancy                              Achieved Occupancy    0.930571    0.939808    0.933779
         79                          sm_efficiency                         Multiprocessor Activity      99.47%      99.66%      99.58%
         79                          gld_throughput                          Global Load Throughput  187.14GB/s  205.73GB/s  203.07GB/s
         79                          gst_throughput                         Global Store Throughput  62.380GB/s  68.577GB/s  67.691GB/s
         79                  shared_load_throughput                  Shared Memory Load Throughput  2994.2GB/s  3291.7GB/s  3249.2GB/s
         79                 shared_store_throughput                 Shared Memory Store Throughput  124.76GB/s  137.15GB/s  135.38GB/s
    Kernel: phase1_kernel(int*, int, int)
         79                    achieved_occupancy                              Achieved Occupancy    0.488140    0.490863    0.490559
         79                          sm_efficiency                         Multiprocessor Activity      4.46%       4.58%       4.52%
         79                          gld_throughput                          Global Load Throughput  3.6261GB/s  4.2135GB/s  4.0999GB/s
         79                          gst_throughput                         Global Store Throughput  3.6261GB/s  4.2135GB/s  4.0999GB/s
         79                  shared_load_throughput                  Shared Memory Load Throughput  87.027GB/s  101.12GB/s  98.398GB/s
         79                 shared_store_throughput                 Shared Memory Store Throughput  29.462GB/s  34.235GB/s  33.312GB/s
```

Experiment & Analysis

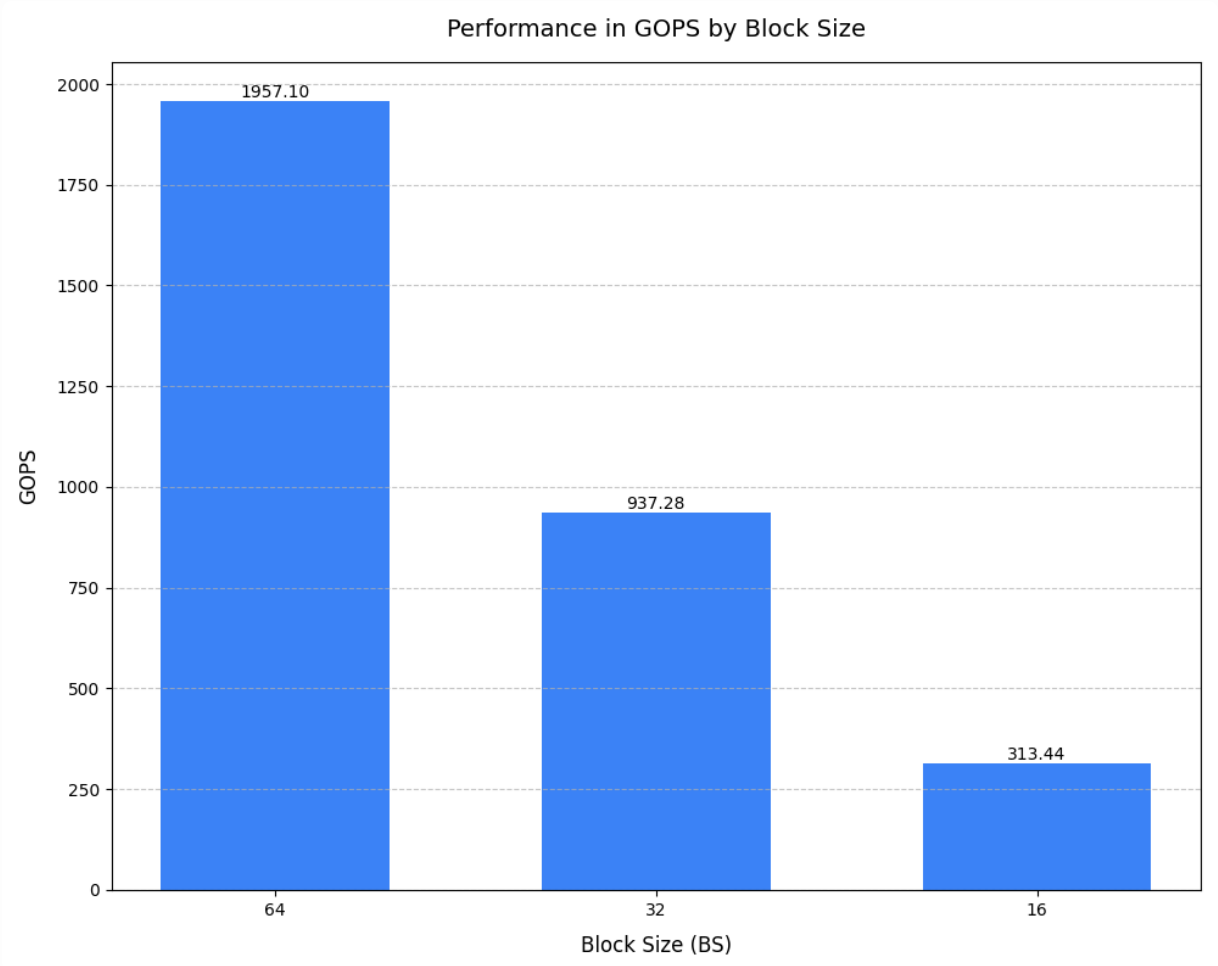
System Spec

實驗皆使用課程提供的 `apollo GPU` 伺服器，並使用 `c20.1` 測資（如有使用其他測資會另外註明）。由於 phase 3 計算量最大，因此部分實驗以 phase 3 的數值為主要依據。

Blocking Factor (hw3-2)

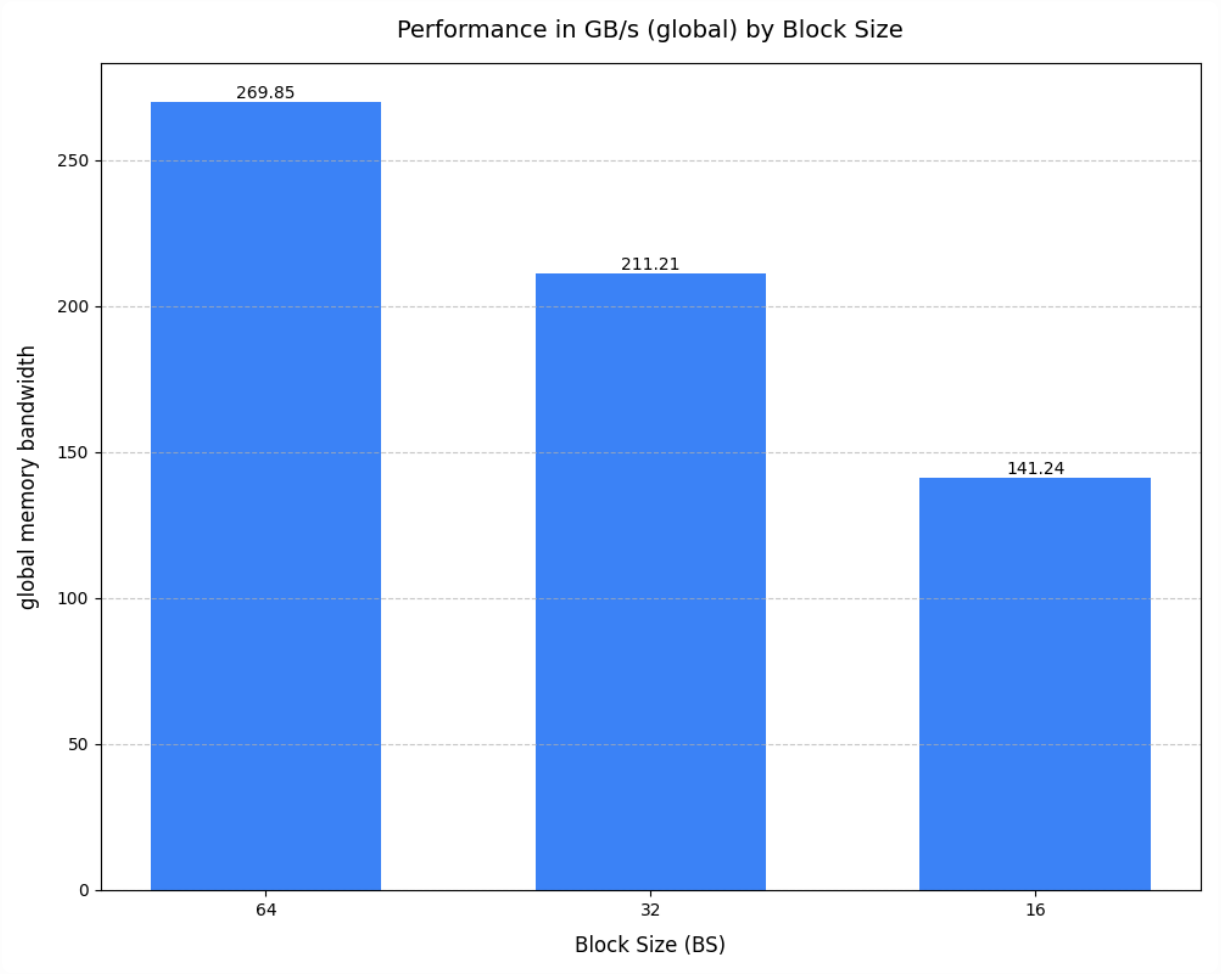
這個實驗研究在不同的演算法 block size 底下，對 gpu 運算造成的效能影響。從實驗結果可以看見，block size 調越大 integer instruction count 越多，而呼叫 kernel 的次數則減少（因為 block size 越大需要做的次數越少），總執行時間隨著 block size 變大而減少。結論：GOPS 隨著 block size 變大而增加：

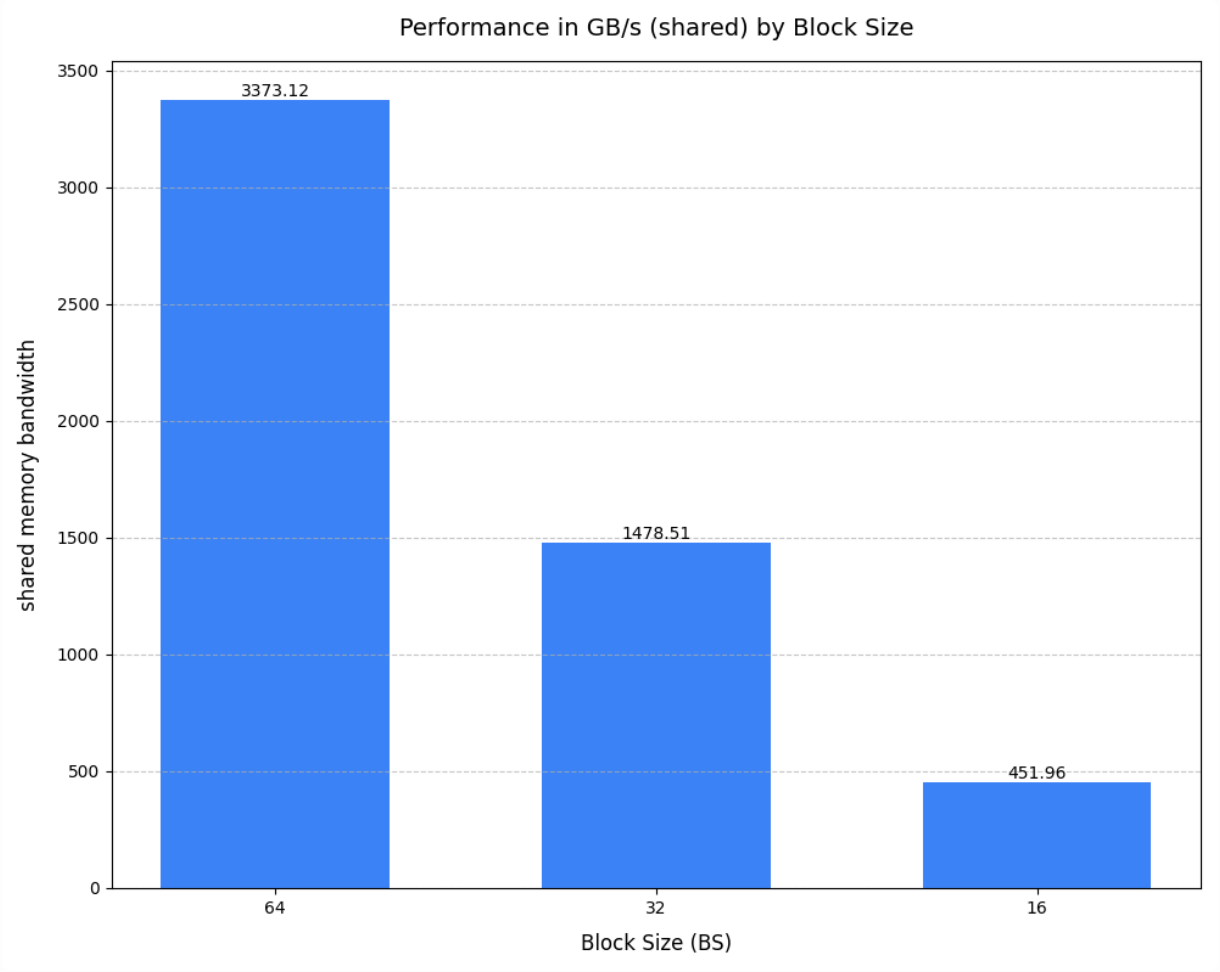
BS	integer instruction count	call count	exe time	GOPS
64	2803507200	79	0.11317	1957.1
32	1607344128	157	0.26924	937.28
16	1009262592	313	1.00784	313.44



對於 memory bandwidth 的部分，也可以看見類似 GOPS 的結果，由於 block size 變大後每次要傳輸的資料量也變大，因此不管是 shared memory 還是 global memory 的 bandwidth 都呈現隨著 block size 增加而遞增的現象：

BS	global load	global store	shared load	shared store	global GB/s	shared GB/s
64	202.39	67.462	3238.2	134.92	269.852	3373.12
32	158.41	52.804	1267.3	211.21	211.214	1478.51
16	84.743	56.495	338.97	112.99	141.238	451.96

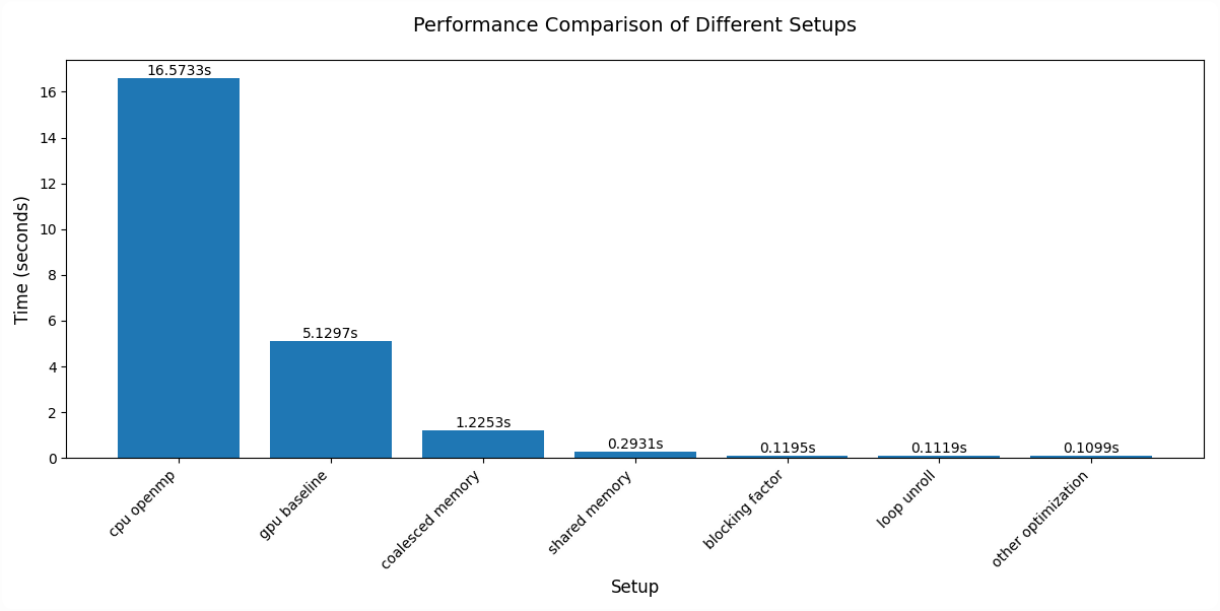

Performance in GB/s (global) by Block Size
global memory bandwidth
Block Size (BS)
269.85
211.21
141.24
0
50
100
150
200
250
64
32
16

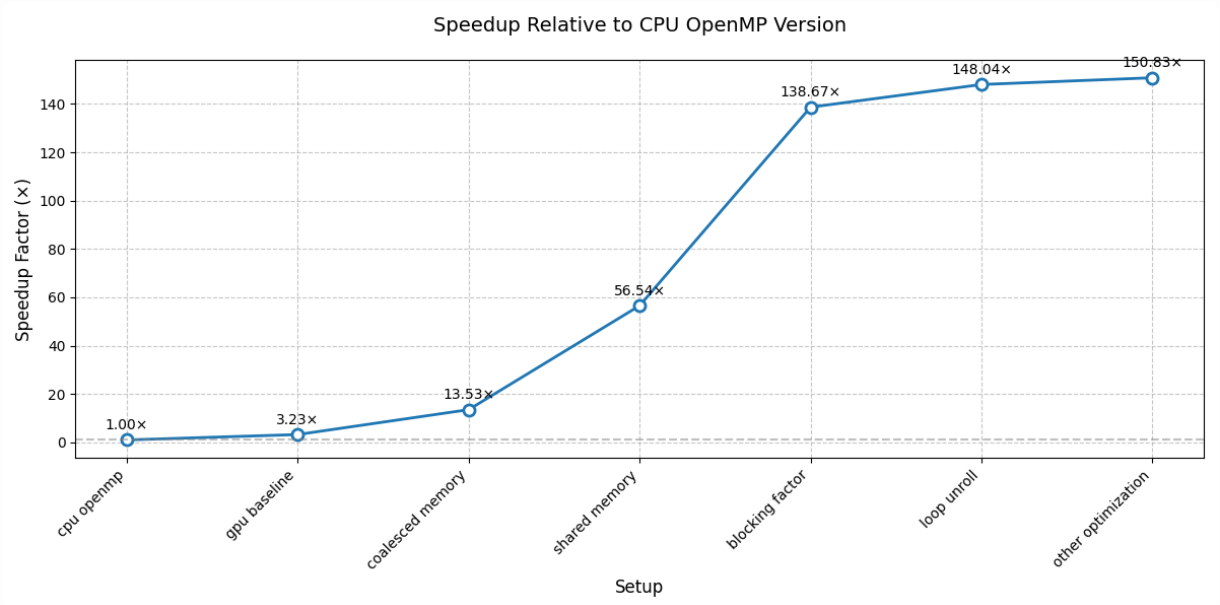


Optimization (hw3-2)

程式優化的部分，主要實驗的優化項目有：

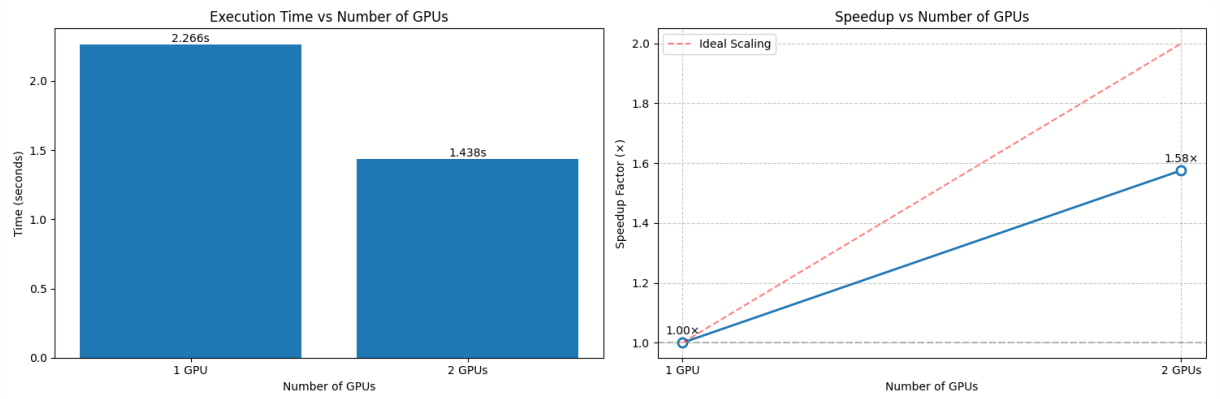
- **coalesced memory access:** 實驗 row major 以及 column major 兩種不同的 indexing pattern。
- **shared memory usage:** 比較單純使用 global memory 以及有用 shared memory 的程式速度差異。
- **blocking factor:** 調整演算法 block size 至 64。
- **loop unroll:** 在部分 for-loop 使用 unroll 技巧。
- **others:** 包含在 Implementation 章節有提到的 shared memory 改為 1D array 等細節優化。





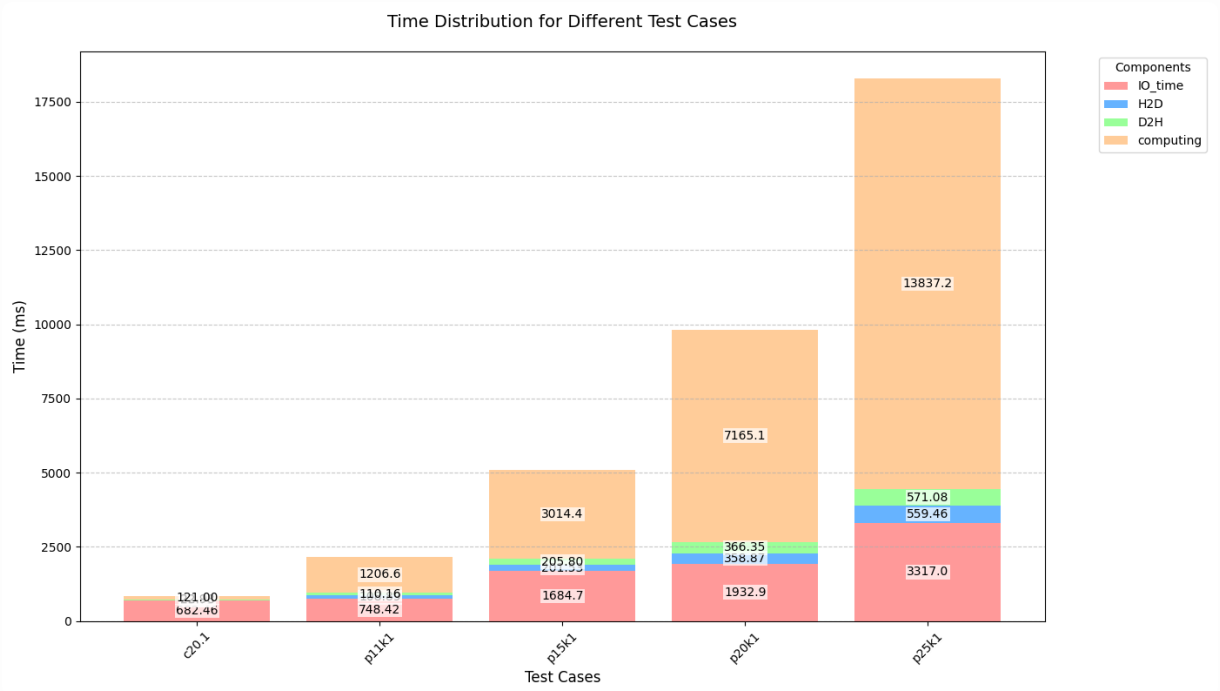
Weak Scalability (hw3-3)

這裡比較同測資（`C20.1`）在單張 gpu 以及兩張 gpu 上的執行速度比較，由實驗結果可得知，將計算量大的 phase 3 分在兩張 gpu 上確實能夠增進程式速度效能，即使可能會有 gpu 溝通上的 overhead。



Time Distribution (hw3-2)

這裡可以看出程式主要的瓶頸其實除了 computing 以外，IO 時間也是程式優化的困難點之一。隨著測資變大，computing 時間逐漸成為程式的 bottle-neck；H2D / D2H 隨著測資變大有微幅增加，不過差異並不似 computing 及 IO 明顯。

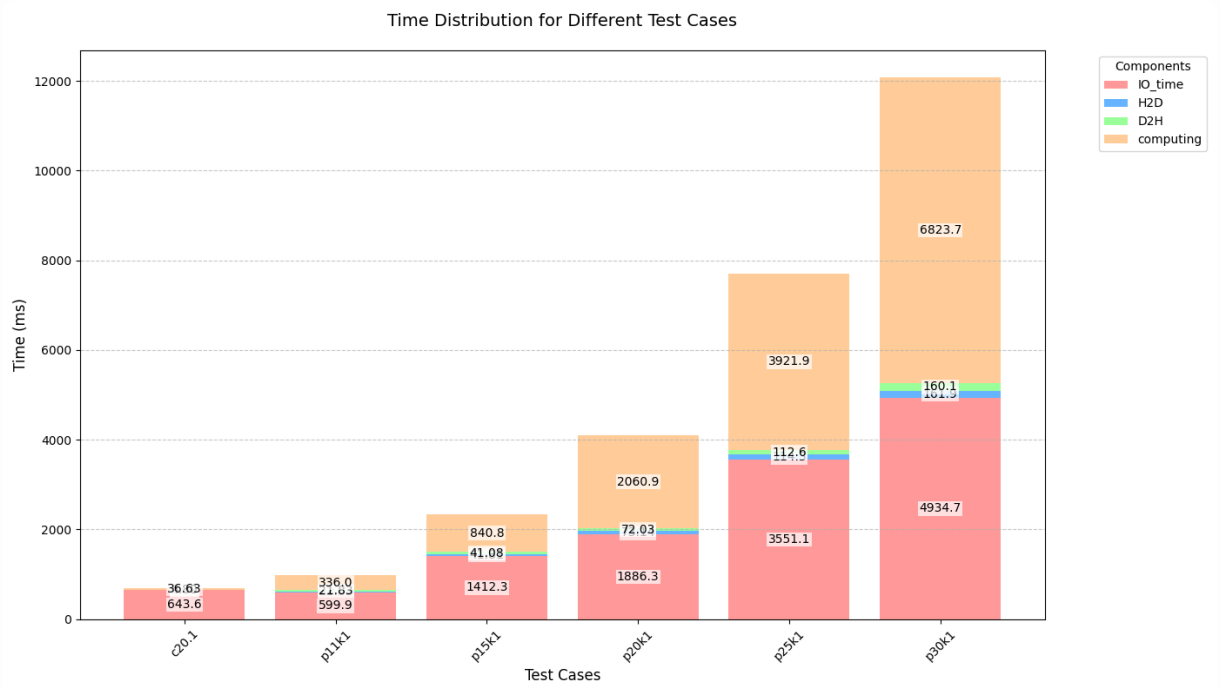


Others

我另外有實驗了在程式有進行優化的情況下，跟 GPU baseline 比較，可見優化前後的 H2D / D2H 時間差異並不大，主要是針對 computing 的時間做優化，優化後的計算時間有明顯的下降。

Experiment on AMD GPU

將 hw3-2 跑在 amd GPU 上的 time distribution 如下，跟上一小節的結果相比，可以發現以硬體效能來說，amd GPU 明顯是比較進階的。IO 的時間差異並不明顯，但 H2D / D2H / computing 這三種時間組成都是 amd GPU 的執行時間較佳：



有使用 amd-judge 來進行 hw3-2 的測試，順利通過所有測資：

```
p35k1   15.40   accepted
p36k1   16.66   accepted
p37k1   17.86   accepted
p38k1   17.71   accepted
p39k1   20.87   accepted
p40k1   20.32   accepted
Removing temporary directory /share/judge_dir/.judge_exe.1062019291
Scoreboard: updated {51 257.85} --> {51 255.43}
```

hw3-3 multi-GPU 版本的程式碼轉為 `.hip` 檔編譯之後，跑 judge 會出現部分測資錯誤的情況：

```
pp24s057@apollo-login:~/hw3-3$ hw3-3-amd-judge
Looking for hw3-3.hip: OK
Looking for Makefile: OK
Running: /usr/bin/make -C /share/judge_dir/.judge_exe.1728103232 hw3-3-amd
make: Entering directory '/share/judge_dir/.judge_exe.1728103232'
hipcc -std=c++11 -O3 --offload-arch=gfx90a -fopenmp -lm -o hw3-3-amd hw3-3.hip
make: Leaving directory '/share/judge_dir/.judge_exe.1728103232'
c01.1    0.62   accepted
c02.1    0.57   wrong answer: be270f9b16bc64b13d9866ab02896c162e8dbb6ae9e909e8d57365efbaca932a,19000
c03.1    0.62   wrong answer: 08a8eb46b74722181f01c0ae582fd3a5c75173bf061ee5b7a0a326fa0781d184,3ce9c4
c04.1    1.22   wrong answer: fc2e9205c72ef14a335399bf8b31199076479a38db1d46e875372e0ba0528026,5f5e100
c05.1    1.17   wrong answer: 44b7ef4499bfa85c0405b420bde9eb1fa15c2a986b10519a35d06534dfce1b52,1cd94100
c06.1   13.20   wrong answer: 9687ce117160c469e20864ef3f5e073c3efcd19767bc5784dca943b7234d79d7,17abf4184
c07.1   17.47   accepted
Removing temporary directory /share/judge_dir/.judge_exe.1728103232
Scoreboard: created {2 18.08}
```

Experience & conclusion

透過這次 CUDA 程式的作業，我學習到許多 GPU 平行運算的重要概念；在效能優化的過程中發現 block size 的選擇對程式執行時間有顯著的影響，較大的 block size 不僅能提升運算效能（GOPS），也能增加記憶體頻寬的使用效率。此外，shared memory 的使用、memory coalescing 以及 loop unroll 等優化技巧都能有效提升程式效能。

在實作 multi-GPU 版本時，雖然將運算量分散到兩張 GPU 確實能加快程式執行速度，但也需要謹慎處理 GPU 之間的資料同步問題。從實驗結果也可以觀察到，程式的效能瓶頸除了計算時間外，I/O 操作也佔據了相當大的比重。

這次的作業讓我對 GPU 程式設計有更深入的理解，也體會到在追求效能優化時，必須同時考慮演算法設計和硬體特性，才能達到最佳的執行效果。