Parallel Programming - Hw3 Report

107062228 陳劭愷

Implementation
- CPU Version (Hw3-1)
- Single-GPU Version (Hw3-2)
  - Padding
  - CUDA 2D Alignment
  - Shared Memory
  - Prevent branching
- Mutli-GPU Version (Hw3-3)

Profiling Results
Experiment & Analysis
- System Spec
- Blocking Factor
- Optimization
- Weak Scalability
- Time Distribution

Experience & Conslution
References

Implementation

CPU Version (Hw3-1)

CPU 版本使用 pthread 來做平行化，使用的是一般的 floyd warshall algorithm。因為 floyd warshall 的每個 i 的運算量是相等的，因此直接對 i 分割，每一個 threads 負責計算一部分的 i。

```cpp
void* floyd_warshall(void* arg) {
  int tid = *(int*)arg;

  int from = tid * (n / num_cpus) + std::min(tid, n % num_cpus);
  int to = from + n / num_cpus + (tid < n % num_cpus);

  for (int k = 0; k < n; ++k) {
    for (int i = from; i < to; ++i) {
      for (int j = 0; j < n; ++j) {
        d[i][j] = std::min(d[i][j], d[i][k] + d[k][j]);
      }
    }

    pthread_barrier_wait(&barrier);
  }

  return NULL;
}
```

因為第 $k+1$ 輪的運算是 depends on 第 k 輪運算的，因此使用 `pthread_barrier_wait` 來等待每一個 threads 完成其第 k 輪的用算才繼續下一輪。

Single-GPU Version (Hw3-2)

使用 blocked floyd warshall。

首先在一開始的實作中，每一個 thread 都負責一個 block 中某個 (i, j) 的計算，每一個 GPU block 都負責一個 floyd warshall block 的運算，因為從 device query 中得知每個 GPU block 最多有 1024 個 threads，因此 blocking factor (B) 就設為 32。

```
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
```

最初版本是直接有時做 shared memory 的，但在經過 unroll to recude branching、CUDA 2D alignment、padding 等優化後，發現速度都沒有顯著的提升，後來決定每一個 thread 多計算一些位置，使 shared memory 能夠被 fully utilized，來提升速度。

因此計算在 phase1 中，每個 block 用了 $1 \times B \times B$ 個 integer 的 shared memory，在 phase 2 與 phase 3 中，每個 block 用了 $3 \times B \times B$ 個 integer 的 shared memory，而由 device query 可以得知每個 GPU block 做多只能用 49152 bytes 的 shared memory。一個 integer 為 4 bytes，而恰好 $3 \times 64 \times 64 \times 4 = 49152$，因此取 blocking factor $B = 64$ 恰好能夠完美的運用完整個 shared memory。

但是我們的 thread 數量還是只有 32×32 個，因此每個 thread 應該要負責 4 個 $(i,\ j)$ 的計算。為了方便，原先 thread $(i,\ j)$ 負責計算 shared memory 中 $(i,\ j)$，並假設其對應的 global memory 位置為 $(x,\ y)$。現在則需多負責計算 shared memory 中 $(i + B/2,\ j)$、$(i,\ j + B/2)$、$(i + B/2,\ j + B/2)$，也就是 global memory 中 $(x + B/2,\ y)$、$(x,\ y + B/2)$、$(x + B/2,\ y + B/2)$ 這些位置。果然改成這樣計算後，速度大增，直接通過後面的一直過不了的後幾筆測資！

```
dim3 gridPhase1(1, 1);
dim3 gridPhase2(numberOfBlocks, 1);
dim3 gridPhase3(numberOfBlocks, numberOfBlocks);
dim3 threadsPerBlock(32, 32);

for (int blockId = 0; blockId < numberOfBlocks; ++blockId) {
  blockedFloydWarshallPhase1<<<gridPhase1, threadsPerBlock>>>(n, blockId, deviceD, pitch / sizeof(int));
  blockedFloydWarshallPhase2<<<gridPhase2, threadsPerBlock>>>(n, blockId, deviceD, pitch / sizeof(int));
  blockedFloydWarshallPhase3<<<gridPhase3, threadsPerBlock>>>(n, blockId, deviceD, pitch / sizeof(int));
}
```

Padding

因為測資給定的 N 不一定是 B 的倍數，會導致需要在 kernel code 裡面的時候多做判斷來確定有沒有超過陣列的範圍，而 branch 會使得 kernel code 變慢，因此用額外的空間來換取速度的提升。

所以在讀入 N 之後，先把 N 放大到 B 的倍數，假設有更多的點，因為其他的點都是沒有邊的，因此不影響原圖的計算結果。使用新算出來的 N 計算完畢後，寫回檔案時只要把部分矩陣寫回即可。

```
origN = n;
if (n % BLOCK_SIZE != 0) {
  n = n + (BLOCK_SIZE - n % BLOCK_SIZE);
}
```

```
FILE* file = fopen(outputFile, "w");

for (int i = 0; i < origN; ++i) {
  fwrite(hostD + i * n, sizeof(int), origN, file);
}
```

CUDA 2D Alignment

對於 host memory → device global memory 的優化，除了使用 `cudaHostRegister` pin 住 host memory 來提升 I/O 的速度，另外使用 `cudaMallocPitch` 以及 `cudaMemcpy2D` 來將 2D 的陣列對齊，使得 memory access 的速度可以提升。

因此底下可以看到 code 中計算 global memory 的索引時，都將原本 $i \times n + j$ 替換成 $i \times pitch + j$。其中 $pitch$ 是 `cudaMallocPitch` 時回傳的 `pitch` 代表每個 rows 的長度（bytes）再除以 integer 的大小 `sizeof(int)`。

```
/* normal cuda malloc + memcpy */
// cudaMalloc((void**)&deviceD, n * n * sizeof(int));
// cudaMemcpy(deviceD, hostD, n * n * sizeof(int), cudaMemcpyHostToDevice);

/* cudaMallocPitch + cudaMemcpy2D */
size_t pitch;
cudaMallocPitch((void**)&deviceD, &pitch, n * sizeof(int), n);
cudaMemcpy2D(deviceD, pitch, hostD, n * sizeof(int), n * sizeof(int), n, cudaMemcpyHostToDevice);
```

Shared Memory

Phase 1 使用一個 $B \times B$ 大小的 shared memory，紀錄 block 中每個位置的值。首先先把 thread 負責的 4 個位置都複製到 shared memory 中：

```
// x: [0, BLOCK_SIZE), y: [0, BLOCK_SIZE)
const unsigned int &x = threadIdx.x;
const unsigned int &y = threadIdx.y;

// load the block into shared memory
int i = y + blockId * BLOCK_SIZE;
int j = x + blockId * BLOCK_SIZE;

__shared__ int cacheD[BLOCK_SIZE][BLOCK_SIZE];
cacheD[y][x] = d[i * pitch + j];
cacheD[y + HBS][x] = d[(i + HBS) * pitch + j];
cacheD[y][x + HBS] = d[i * pitch + (j + HBS)];
cacheD[y + HBS][x + HBS] = d[(i + HBS) * pitch + (j + HBS)];
__syncthreads();
```

再來就是計算 4 個位置的值，因為 phase 1 是 dependense phase（其實就是跟一般的 floyd warshall 一樣），每一輪 k 的結果都是依賴於上一輪 k 的結果，因此需要 `__syncthreads` 來同步其他 threads 的計算結果。

```
// compute phase 1 - dependent phase
#pragma unroll 32
for (int k = 0; k < BLOCK_SIZE; ++k) {
  // using cuda min
  cacheD[y][x] = min(cacheD[y][x], cacheD[y][k] + cacheD[k][x]);
  cacheD[y + HBS][x] = min(cacheD[y + HBS][x], cacheD[y + HBS][k] + cacheD[k][x]);
  cacheD[y][x + HBS] = min(cacheD[y][x + HBS], cacheD[y][k] + cacheD[k][x + HBS]);
  cacheD[y + HBS][x + HBS] = min(cacheD[y + HBS][x + HBS], cacheD[y + HBS][k] + cacheD[k][x + HBS]);
  __syncthreads();
}
```

最後再將值更新回 global memory 中：

```
// load shared memory back to the global memory
d[i * pitch + j] = cacheD[y][x];
d[(i + HBS) * pitch + j] = cacheD[y + HBS][x];
d[i * pitch + (j + HBS)] = cacheD[y][x + HBS];
d[(i + HBS) * pitch + (j + HBS)] = cacheD[y + HBS][x + HBS];
```

Phase 2 總共使用 N/B 個 GPU blocks 來計算（但是會扣除掉 phase 1 計算過的 block），每個 thread 負責一個與 pivot block 同 row 的 block **以及** 一個與 pivot block 同 column 的 block。

這裡也可以使用 $2 \times N/B$ 個 GPU blocks 來計算，每個 thread 負責一個與 pivot block 同 row 的 block **或是** 一個與 pivot block 同 column 的 block。

但是這樣的一個 GPU block 只使用到 $2 \times B \times B$ 個 integers 的 shared memory，並沒有完整的運用 shared memory 資源；另外他們共用的 pivot block 還需要被 load 兩次，會更沒有效率。因此採用上面的方案。

首先先將 pivot block 載入到 shared memory 中，再將 row 與 column 對應的 block 也載入到 shared memory 中，最後 `syncthreads` 確認資料載入完畢：

```
// load the base block into shared memory
int i = y + blockId * BLOCK_SIZE;
int j = x + blockId * BLOCK_SIZE;

__shared__ int cacheBaseD[BLOCK_SIZE][BLOCK_SIZE];
cacheBaseD[y][x] = d[i * pitch + j];
cacheBaseD[y + HBS][x] = d[(i + HBS) * pitch + j];
cacheBaseD[y][x + HBS] = d[i * pitch + (j + HBS)];
cacheBaseD[y + HBS][x + HBS] = d[(i + HBS) * pitch + (j + HBS)];
```

```
// load the target block of same row into shared memory
i = y + blockId * BLOCK_SIZE;
j = x + blockIdx.x * BLOCK_SIZE;

__shared__ int cacheRowD[BLOCK_SIZE][BLOCK_SIZE];
cacheRowD[y][x] = d[i * pitch + j];
cacheRowD[y + HBS][x] = d[(i + HBS) * pitch + j];
cacheRowD[y][x + HBS] = d[i * pitch + (j + HBS)];
cacheRowD[y + HBS][x + HBS] = d[(i + HBS) * pitch + (j + HBS)];
```

```
// load the target block of same column into shared memory
i = y + blockIdx.x * BLOCK_SIZE;
j = x + blockId * BLOCK_SIZE;

__shared__ int cacheColD[BLOCK_SIZE][BLOCK_SIZE];
cacheColD[y][x] = d[i * pitch + j];
cacheColD[y + HBS][x] = d[(i + HBS) * pitch + j];
cacheColD[y][x + HBS] = d[i * pitch + (j + HBS)];
cacheColD[y + HBS][x + HBS] = d[(i + HBS) * pitch + (j + HBS)];
```

```
__syncthreads();
```

接著開始 phase 2 的計算，phase 2 的運算只 dependent on 同一個點的運算結果以及 phase 1 的運算結果，因此可以不需要在每一輪 k 都做一次同步的動作：

```
// compute phase 2 - partial dependent phase
#pragma unroll 32
for (int k = 0; k < BLOCK_SIZE; ++k) {
  // using cuda min
  cacheRowD[y][x] = min(cacheRowD[y][x], cacheBaseD[y][k] + cacheRowD[k][x]);
  cacheRowD[y + HBS][x] = min(cacheRowD[y + HBS][x], cacheBaseD[y + HBS][k] + cacheRowD[k][x]);
  cacheRowD[y][x + HBS] = min(cacheRowD[y][x + HBS], cacheBaseD[y][k] + cacheRowD[k][x + HBS]);
  cacheRowD[y + HBS][x + HBS] = min(cacheRowD[y + HBS][x + HBS], cacheBaseD[y + HBS][k] + cacheRowD[k][x + HBS]);

  cacheColD[y][x] = min(cacheColD[y][x], cacheColD[y][k] + cacheBaseD[k][x]);
  cacheColD[y + HBS][x] = min(cacheColD[y + HBS][x], cacheColD[y + HBS][k] + cacheBaseD[k][x]);
  cacheColD[y][x + HBS] = min(cacheColD[y][x + HBS], cacheColD[y][k] + cacheBaseD[k][x + HBS]);
  cacheColD[y + HBS][x + HBS] = min(cacheColD[y + HBS][x + HBS], cacheColD[y + HBS][k] + cacheBaseD[k][x + HBS]);
}
```

最後將 shared memory 中的資料載入回 global memory 中：

```
// load shared memory back to the global memory

// i = y + blockIdx.x * BLOCK_SIZE;
// j = x + blockId * BLOCK_SIZE;
d[i * pitch + j] = cacheColD[y][x];
d[(i + HBS) * pitch + j] = cacheColD[y + HBS][x];
d[i * pitch + (j + HBS)] = cacheColD[y][x + HBS];
d[(i + HBS) * pitch + (j + HBS)] = cacheColD[y + HBS][x + HBS];

i = y + blockId * BLOCK_SIZE;
j = x + blockIdx.x * BLOCK_SIZE;
d[i * pitch + j] = cacheRowD[y][x];
d[(i + HBS) * pitch + j] = cacheRowD[y + HBS][x];
d[i * pitch + (j + HBS)] = cacheRowD[y][x + HBS];
d[(i + HBS) * pitch + (j + HBS)] = cacheRowD[y + HBS][x + HBS];
```

Phase 3 總共使用 $N/B \times N/B$ 個 GPU blocks 來計算（但是會扣除掉 phase 1 與 phase 2 計算過的 block），每個 thread 負責計算一個 block 的資料，另外需要與之同 row 以及同 column 的兩個 block 的資料。

因此首先將這三個 block 的資料都 load 到 shared memory 中：

```
// load the base column block (same row) into shared memory
__shared__ int cacheBaseColD[BLOCK_SIZE][BLOCK_SIZE];
i = y + blockIdx.y * BLOCK_SIZE;
j = x + blockId * BLOCK_SIZE;
cacheBaseColD[y][x] = d[i * pitch + j];
cacheBaseColD[y + HBS][x] = d[(i + HBS) * pitch + j];
cacheBaseColD[y][x + HBS] = d[i * pitch + (j + HBS)];
cacheBaseColD[y + HBS][x + HBS] = d[(i + HBS) * pitch + (j + HBS)];

// load the base row block (same column) into shared memory
__shared__ int cacheBaseRowD[BLOCK_SIZE][BLOCK_SIZE];
i = y + blockId * BLOCK_SIZE;
j = x + blockIdx.x * BLOCK_SIZE;
cacheBaseRowD[y][x] = d[i * pitch + j];
cacheBaseRowD[y + HBS][x] = d[(i + HBS) * pitch + j];
cacheBaseRowD[y][x + HBS] = d[i * pitch + (j + HBS)];
cacheBaseRowD[y + HBS][x + HBS] = d[(i + HBS) * pitch + (j + HBS)];
__syncthreads();

// load the target block into shared memory
__shared__ int cacheD[BLOCK_SIZE][BLOCK_SIZE];
i = y + blockIdx.y * BLOCK_SIZE;
j = x + blockIdx.x * BLOCK_SIZE;
cacheD[y][x] = d[i * pitch + j];
cacheD[y + HBS][x] = d[(i + HBS) * pitch + j];
cacheD[y][x + HBS] = d[i * pitch + (j + HBS)];
cacheD[y + HBS][x + HBS] = d[(i + HBS) * pitch + (j + HBS)];
```

再來一樣做計算，phase 3 只 dependent on 同一個點的運算結果以及 phase 2 的運算結果，因此也不需要做額外的同步：

```
// compute phase 3 - independence phase
#pragma unroll 32
for (int k = 0; k < BLOCK_SIZE; ++k) {
  // using cuda min
  cacheD[y][x] = min(cacheD[y][x], cacheBaseColD[y][k] + cacheBaseRowD[k][x]);
  cacheD[y + HBS][x] = min(cacheD[y + HBS][x], cacheBaseColD[y + HBS][k] + cacheBaseRowD[k][x]);
  cacheD[y][x + HBS] = min(cacheD[y][x + HBS], cacheBaseColD[y][k] + cacheBaseRowD[k][x + HBS]);
  cacheD[y + HBS][x + HBS] = min(cacheD[y + HBS][x + HBS], cacheBaseColD[y + HBS][k] + cacheBaseRowD[k][x + HBS]);
}
```

最後將 shared memory 中的資料載入回 global memory 中：

```
// load shared memory back to the global memory
d[i * pitch + j] = cacheD[y][x];
d[(i + HBS) * pitch + j] = cacheD[y + HBS][x];
d[i * pitch + (j + HBS)] = cacheD[y][x + HBS];
d[(i + HBS) * pitch + (j + HBS)] = cacheD[y + HBS][x + HBS];
```

Prevent branching

使用 unroll 將 64 次的迴圈展開，可以減少 branching。從 global memory 中可以看到 warp size 是 32，因此 unroll 時也只 unroll 32 個以免造成 bank conflict。

另外不使用 if-else 來做運算，而改用 cuda 提供的 `min` 函數，一樣可以降低 branching 提高運算速度。

Mutli-GPU Version (Hw3-3)

使用 OpenMP 來開啟兩個 threads，分別操作兩個 GPU 的工作。

使用兩個 GPU 的版本基本上跟 Single-GPU 差不多，我優化的是 phase 3 計算部分。因為 phase 3的運算量是最主要的（phase 1 只要計算一個 block、phase 2 要計算 $2 \times N/B - 1$ 個、phase 3 要計算 $((N/B) \times (N/B)) - 2 \times N/B$ 個）。

將 phase 3 要計算的 block 直接切成上下兩半，phase 1、phase 2 還是可以兩個 GPU 個別計算。計算後，在第 i 輪時，需要將負責該行的 block 的資料同步到另一個 GPU 上，因此使用 `cudaMemcpy` 將資料複製到另一張 GPU 上。

```
#pragma omp parallel num_threads(2)
{
  int threadId = omp_get_thread_num();
  cudaSetDevice(threadId);
  cudaMalloc((void**)&deviceD[threadId], n * n * sizeof(int));

  dim3 gridPhase3(numberOfBlocks, numberOfBlocks / 2);
  if (threadId == 1 && (numberOfBlocks & 1)) ++gridPhase3.y;

  int yOffset = (threadId == 0) ? 0 : numberOfBlocks / 2;

  cudaMemcpy(deviceD[threadId] + yOffset * BLOCK_SIZE * n, hostD + yOffset * BLOCK_SIZE * n, gridPhase3.y * BLOCK_SIZE * n * sizeof(int), cudaMemcpyHostToDevice);

  for (int blockId = 0; blockId < numberOfBlocks; ++blockId) {
    if (blockId >= yOffset && blockId < yOffset + gridPhase3.y) {
      cudaMemcpy(hostD + blockId * BLOCK_SIZE * n, deviceD[threadId] + blockId * BLOCK_SIZE * n, BLOCK_SIZE * n * sizeof(int), cudaMemcpyDeviceToHost);
      // cudaMemcpy(deviceD[threadId ^ 1] + blockId * BLOCK_SIZE * n, deviceD[threadId] + blockId * BLOCK_SIZE * n, BLOCK_SIZE * n * sizeof(int), cudaMemcpyDeviceToDevice);
    }
    #pragma omp barrier

    if (blockId < yOffset || blockId >= yOffset + gridPhase3.y) {
      cudaMemcpy(deviceD[threadId] + blockId * BLOCK_SIZE * n, hostD + blockId * BLOCK_SIZE * n, BLOCK_SIZE * n * sizeof(int), cudaMemcpyHostToDevice);
    }

    blockedFloydWarshallPhase1<<<gridPhase1, threadsPerBlock>>>(n, blockId, deviceD[threadId], n);
    blockedFloydWarshallPhase2<<<gridPhase2, threadsPerBlock>>>(n, blockId, deviceD[threadId], n);
    blockedFloydWarshallPhase3<<<gridPhase3, threadsPerBlock>>>(n, blockId, deviceD[threadId], n, yOffset);
  }

  cudaMemcpy(hostD + yOffset * BLOCK_SIZE * n, deviceD[threadId] + yOffset * BLOCK_SIZE * n, gridPhase3.y * BLOCK_SIZE * n * sizeof(int), cudaMemcpyDeviceToHost);
}
```

Profiling Results

使用的是 p11k1 這筆測資來做 profiling，觀察的是 load 最重的 phase 3 kernel。

Metric Name	Min	Max	Average
Achieved Occupancy	0.922403	0.924590	0.923450
Multiprocessor Activity	99.86%	99.93%	99.91%
Shared Memory Load Throughput	3114.0GB/s	3180.9GB/s	3148.8GB/s
Shared Memory Store Throughput	259.50GB/s	265.07GB/s	262.40GB/s
Global Memory Load Throughput	194.62GB/s	198.80GB/s	196.80GB/s
Global Memory Store Throughput	64.875GB/s	66.268GB/s	65.600GB/s

Experiment & Analysis

System Spec

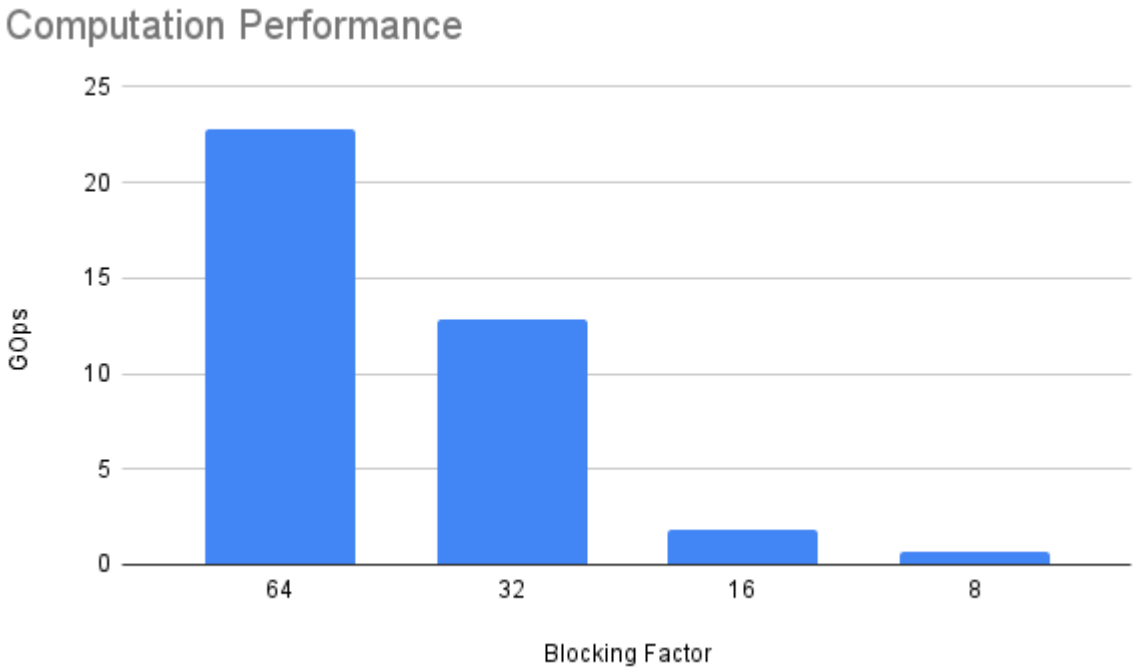
使用課程提供的 hades server。

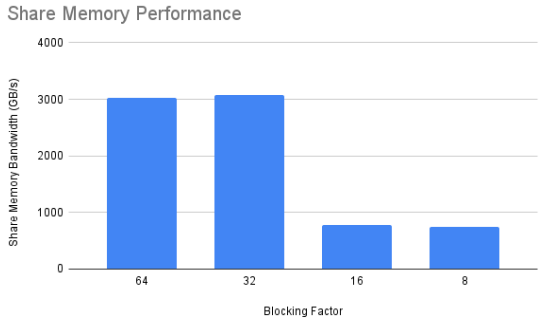
Blocking Factor

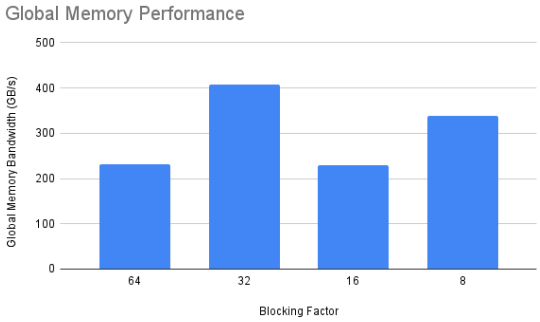
由於使用過大的測資會使得 profiling timeout，因此這裡使用較小的 c21.1 測資來做測試。

GOps 計算為總 integer instruction 數量除以 total time。Memory bandwidth 的部分則是將 load/store 的加總。

	Global Memory Load	Global Memory Store	Share Memory Load	Share Memory Store	Integer Instruction	Total Time
64	173.65GB/s	57.882GB/s	2789.4GB/s	232.45GB/s	3071719424	134.73ms
32	305.63GB/s	101.88GB/s	2455.7GB/s	613.92GB/s	1825554944	142.03ms
16	137.42GB/s	91.614GB/s	520.51GB/s	260.26GB/s	1227393152	656.27ms
8	253.25GB/s	84.416GB/s	500.75GB/s	250.37GB/s	928312352	1.30977s



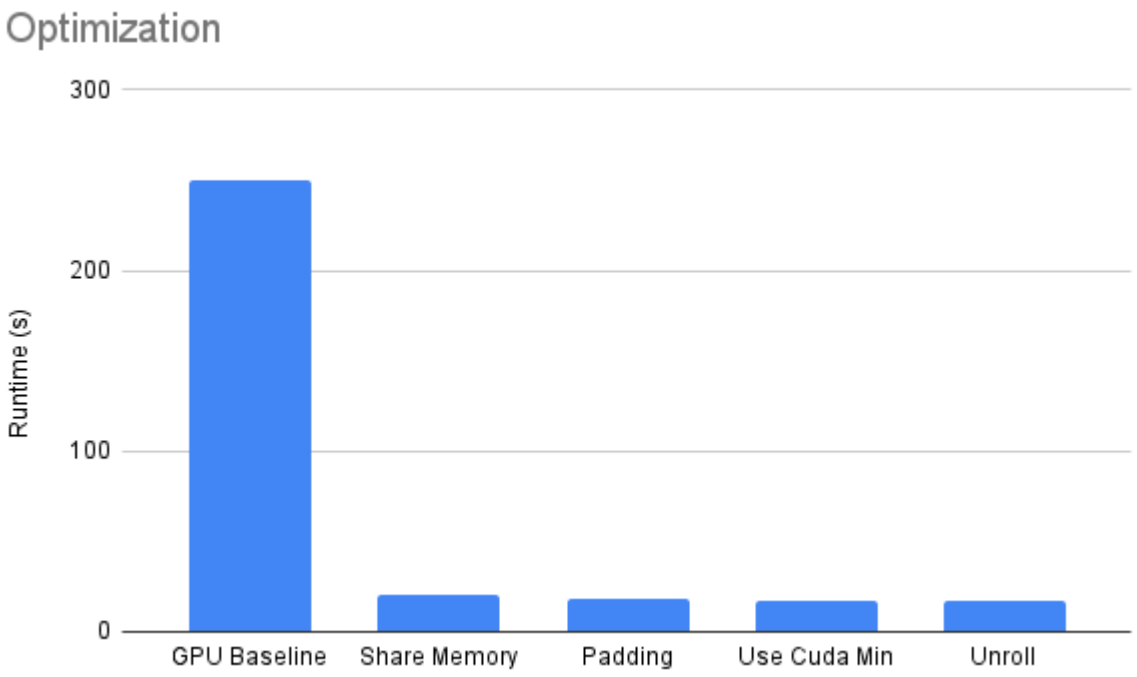




觀察圖表可以發現當 blocking factor 越大時 computation performance 就越好。而在下面 time distribution 有提到我們最需要優化的部分是 computation 的部分，因此選擇 blocking factor 64 是最好的。Blocking factor 128 時，會需要過多的 share memory space，因此無法選擇也比較困難測試。

另外可以看到當 blocking factor 到 64 時，share memory 與 global memory bandwidth 都有下降，推測是一次處理的資料量多到超過 memory 的傳輸上限了，因此反而造成下降。

Optimization



根據實驗發現除了 share memory 以外的優化基本上都只有加速一點點而已。Share memory 直接優化了原本沒有 share memory 的版本超過 100 倍，因此也可以看出 fully utilized share memory 的重要性。Optimization 的方法都有在上面實作的部分提到。

Weak Scalability

因為 weak scalability 是希望兩邊的運算量約等於計算資源量。而 floyd warshall 的時間複雜度為 $O(N^3)$，因此我找到兩筆測資其中 ${N_1}^3 \approx {N_2}^3$。使用 p23k1 以及 p29k3 兩筆測資來實驗。

p23k1 → $N_1 = 22973, {N_1}^3 = 12124201281317.$

p29k3 → $N_2 = 28979, {N_2}^3 = 24336055357739.$

$2 \times {N_1}^3 = 24248402562634 \approx {N_2}^3 = 24336055357739$。

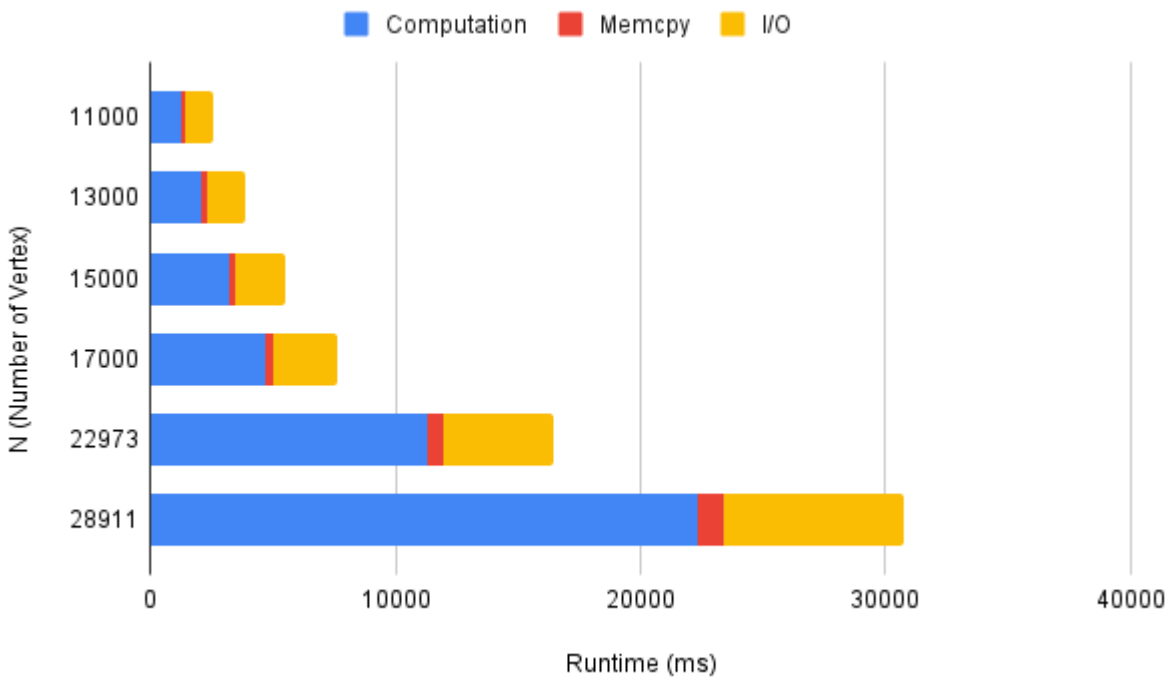
Test Case	# of GPUs	Total Time (ms)	Computation Time (ms)
p23k1	1	16421	11216
p29k3	2	17543	12877

由圖表可以看出在 phase1 與 phase2 沒有優化的情況下，weak scalability 還是不錯的。基本上執行時間並沒有太大的差距。

Time Distribution

Computation time 與 memory copy time 利用 nvprof 的 summary mode 來計算。而 I/O 部分使用與之前作業一樣的方式，也就是 c++11 的 chrono library。

Time Distribution (ms)	N	Computation (ms)	Memcpy (ms)	I/O (ms)
p11k1	11000	1244.515	149.54	1136
p13k1	13000	2097.422	209.97	1546
p15k1	15000	3181.580	280.90	2054
p17k1	17000	4652.929	356.92	2580
p23k1	22973	11309.478	655.02	4483
p29k1	28911	22361.554	1038.98	7356



由圖表可以看出 I/O time 與 Computation Time 都隨著 N 增加而變長，但是 I/O Time 是這次比較難優化的部分，因此我們主要要優化的是 Computation 的部分。

Experience & Conslution

這次作業前前後後花了好幾十個小時，包括一開始沒有做 fully utilized share memory 時不管怎麼優化都跑不過後面的測資，到寫 reports 時做的種種測試。這次作業學到了很多 GPU 優化的技巧，也透過實驗了解到底哪些優化對程式整體的運行速度是最大的。

References

https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

https://zhuanlan.zhihu.com/p/248156323

https://docs.nvidia.com/cuda/profiler-users-guide/index.html#profiling-overview