HW4 - Blocked All-Pairs Shortest Path

106062322 江岷錡

Implementation

本次實作 `Floyd Warshall Algorithm` 的 cuda 版本，以下將分別描述 `資料讀取與儲存` 與 `cuda 實作細節`。

資料讀取與儲存

根據測試，本次開設 Block Size 為 `64` 有最好的效果。

而其他除了基本的 讀取 / 寫回方法與上次作業相同外，本次多做了以下更新：

1. `Memory Padding` ： 由於本次為實作 blocked(tiled) version APSP ，會有 matrix width 無法整除 block width 的可能性。同時，在 cuda 實作中，為了解決 bank conflict 的問題，也須透過 padding 的方式，讓資料避免 aligned 到統一的 bank 。於此，我擴充 matrix size ，讓其擴增到 `BLOCK_WIDTH` 的整數倍，並讓多餘的 element 存取 `INF` 值。

   ex: 若 `BLOCK_WIDTH` 為 `32x32` ，但 Matrix 大小為 `60x60` ，我會將 Matrix 擴增到 `64x64` 。

   ```
   n = original_n + (BLOCK_SIZE - (original_n%BLOCK_SIZE));
   ```

2. `Cuda Memory Pinned`： 由於本次實作為 cuda 版本，需將 matrix 讀取到 GPU global memory 內。我使用 `cudaHostRegister` ，將 allocated 的 CPU Matrix Memory 先 pin 住，再 load 到 global memory ，加快搬移效率。

   ```
   cudaHostRegister(Dist, matrixSize, cudaHostRegisterDefault);
   ```

3. `Load Memory to CUDA`: 延續前文，由於本次實作為 cuda ，我進行 memory 的搬移。值得注意的是，因為這次 matrix 相當大，需開 `unsigned long` 的大小才可紀錄 `Matrix Width`。

```
const unsigned long matrixSize = n * n * sizeof(int);

cudaMalloc(&dst, matrixSize);
cudaMemcpy(dst, Dist, matrixSize, cudaMemcpyHostToDevice);
```

Cuda 實作細節

Initialization

在 Blocked Version 中，每次操作都須進行 3 Phase 的個別計算。因此，計算完總共需跑的 `Round` 數後，我便在每個 Round 依序跑 3 Phases 。

```
// dst -> dev memory    r -> round    n -> matrix width
for (int r = 0; r < round; ++r) {
    /* Phase 1*/
    Phase1<<<1, block_dim>>>(dst, r, n);

    /* Phase 2*/
    Phase2<<<blocks, block_dim>>>(dst, r, n);

    /* Phase 3*/
    Phase3<<<grid_dim, block_dim>>>(dst, r, n);
}
```

對於任一 cuda block，因為設計上讓其只計算一個 `Matrix Block` ，我給予 Warp Thread 的倍數 ，也就是 32×32，以確保 memory access 上能更加快，也滿足 `Occupancy Optmization` 的要求。

```
dim3 block_dim(32, 32, 1);
```

而對於 Phase 2 而言， 因為需要計算橫軸一列 ＋縱軸一列的 matrix ，為求最快平行，我開設 `matrix blocks` 長的 `cuda block` 數量。

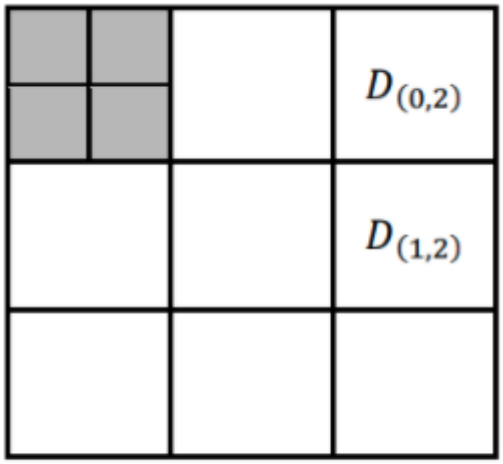
ex: 假設 `matrix` 是 `128x128` ， `matrix block` 為 `32x32` ，則`matrix blocks` 就是 `4` 。

```
const int blocks = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;
```

對於 Phase 3 而言，同時有將近全部的 matrix block 都要計算，因此我開設總 `matrix blocks` 長的 `cuda blocks` (`[matrix blocks] x [matrix blocks]`)，讓所有計算能平行化。

```
dim3 grid_dim(blocks, blocks, 1);
```

Phase 1



(d) Phase 1

對於 Phase 1 而言，需利用 `dim3(32,32)` threads ，計算一個 `64x64` 的 diagonal Matrix 。因此，對於每個 thread 而言，每次需同時計算 4 個 element。

對於 `thread(0, 0)` 而言，他需要計算 `(0, 0), (32, 0), (0, 32), (32, 32)` 這些位置。

對於 `thread(1, 1)` 而言，他需要計算 `(1, 1), (33, 1), (1, 33), (33, 33)` 這些位置，其餘以此類推。

同時，為了提高計算效率，也設計了 `64x64` 的 `shared memory`，讓後續計算資料可以更快存取。

```
__shared__ int C[BLOCK_SIZE][BLOCK_SIZE]; // 2d
```

對於每個 thread 而言 ，主要工作便是讀取自己的 element 位置， load 進 shared memory。同時，因 load share memory 為 concurrent 執行，需透過 `__syncthreads` syncing 每個 thread 的讀取狀態。

```
const int innerI = threadIdx.y;
const int innerJ = threadIdx.x;
const int offset = BLOCK_SIZE * Round;

C[innerI][innerJ]
= dist[offset*(n+1) + innerI*n + innerJ];

C[innerI+HALF_BLOCK_SIZE][innerJ]
= dist[offset*(n+1) + (innerI+HALF_BLOCK_SIZE)*n + innerJ];

C[innerI][innerJ+HALF_BLOCK_SIZE]
= dist[offset*(n+1) + innerI*n + innerJ + HALF_BLOCK_SIZE];

C[innerI+HALF_BLOCK_SIZE][innerJ+HALF_BLOCK_SIZE]
= dist[offset*(n+1) + (innerI+HALF_BLOCK_SIZE)*n + innerJ + HALF_BLOCK_SIZE];
__syncthreads();
```

接著，便是進行 `Floyd Warshall Algorithm` 。此演算法的重點在於，每次計算中都找出第三點，計算距離連至第三點後，是否較短。也因此設計了一個大 `k` for-loop ，遍歷所有可能，並用 `__syncthreads` 確保全部 thread 計算狀態。以下是本步驟的額外設計：

1. **三元表示法**：在 CUDA 計算中，為求加快效率，我使用 `三元表示法` 進行設定。用 `三元表示法` 好處為，他會觸發 `ILP optimization` ，讓其 compile 後的 `assembly code` 能更快運行。
2. **切 32 為單位做計算**： 回應前述 load 的設計，這樣的配置得以讓 memory access 以 `Coalesced memory access` 讀取，並回應 `SIMD` 架構下 (Warp Size is 32 threads) concurrent 執行的優勢。

```
for (int k = 0; k < BLOCK_SIZE; k++) {
  C[innerI][innerJ]
  = (C[innerI][k] + C[k][innerJ]) < C[innerI][innerJ] ?
    (C[innerI][k] + C[k][innerJ]) : C[innerI][innerJ];

  C[innerI+HALF_BLOCK_SIZE][innerJ]
  = (C[innerI+HALF_BLOCK_SIZE][k] + C[k][innerJ]) <
     C[innerI+HALF_BLOCK_SIZE][innerJ] ?
    (C[innerI+HALF_BLOCK_SIZE][k] + C[k][innerJ]) :
     C[innerI+HALF_BLOCK_SIZE][innerJ];

  C[innerI][innerJ+HALF_BLOCK_SIZE]
  = (C[innerI][k] + C[k][innerJ+HALF_BLOCK_SIZE]) <
     C[innerI][innerJ+HALF_BLOCK_SIZE] ?
    (C[innerI][k] + C[k][innerJ+HALF_BLOCK_SIZE]) :
     C[innerI][innerJ+HALF_BLOCK_SIZE];

  C[innerI+HALF_BLOCK_SIZE][innerJ+HALF_BLOCK_SIZE]
  = (C[innerI+HALF_BLOCK_SIZE][k] + C[k][innerJ+HALF_BLOCK_SIZE]) <
     C[innerI+HALF_BLOCK_SIZE][innerJ+HALF_BLOCK_SIZE] ?
```

```
    (C[innerI+HALF_BLOCK_SIZE][k] + C[k][innerJ+HALF_BLOCK_SIZE]) :
     C[innerI+HALF_BLOCK_SIZE][innerJ+HALF_BLOCK_SIZE];
      __syncthreads();
}
```

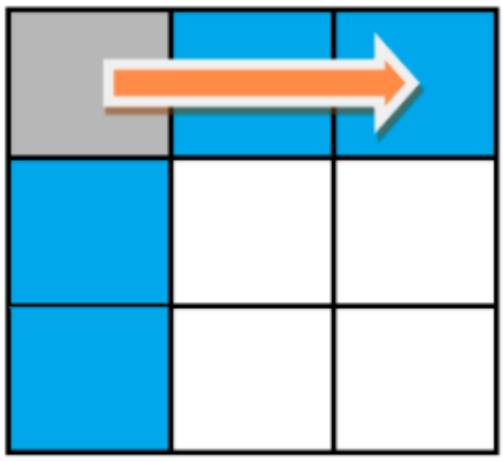
最終讀取完後，再將資料從 shared memory 寫回 device memory。

```
dist[offset*(n+1) + innerI*n + innerJ]
= C[innerI][innerJ];

dist[offset*(n+1) + (innerI+HALF_BLOCK_SIZE)*n + innerJ]
= C[innerI+HALF_BLOCK_SIZE][innerJ];

dist[offset*(n+1) + innerI*n + innerJ + HALF_BLOCK_SIZE]
= C[innerI][innerJ+HALF_BLOCK_SIZE];

dist[offset*(n+1) + (innerI+HALF_BLOCK_SIZE)*n + innerJ + HALF_BLOCK_SIZE]
= C[innerI+HALF_BLOCK_SIZE][innerJ+HALF_BLOCK_SIZE];
```

Phase 2

對於 Phases 2 而言，總共有 `(Total_V/Matrix_V)` 個 `cuda block` ，每個 `cuda block` 都有自己的 `32x32 threads` 。因此，對於任一 `cuda block` 而言，首要任務便是計算縱軸上的一個 `Matrix Block` 與橫軸上的一個 `Matrix Block` 。

根據 `Blocked Floyd Warshall Algorithm` ，對於任一 `cuda block` 而言，讀取上僅需讀 Phase1 的 Diagonal Block Matrix ，與各自的縱軸/橫軸 Matrix Block ，即可完成各自 Phase 2 的計算。



(e) Phase 2

```
const int i = blockIdx.x;
const int innerI = threadIdx.y;
const int innerJ = threadIdx.x;
const int diagonalOffset = BLOCK_SIZE * Round;

__shared__ int Diagonal[BLOCK_SIZE][BLOCK_SIZE];
__shared__ int A[BLOCK_SIZE][BLOCK_SIZE];
__shared__ int B[BLOCK_SIZE][BLOCK_SIZE];

A[innerI][innerJ] = dist[i*BLOCK_SIZE*n + Round*BLOCK_SIZE + innerI*n + innerJ];
// ... omit

B[innerI][innerJ] = dist[Round*BLOCK_SIZE*n + i*BLOCK_SIZE + innerI*n + innerJ];
// ... omit

Diagonal[innerI][innerJ] = dist[diagonalOffset*(n+1) + innerI*n + innerJ];
// ... omit

__syncthreads();
```

計算上，因為 Phase 2 沒有 k round Dependency 的問題，所以就將 `__syncthreads` 移除，並使用 `unroll` 加快計算效率。 儘管一次 K round 需計算 `BLOCK_SIZE = 64` 長的大小，但此部分 `unroll` 設定為 `32` ，因設為 `64` 會遇見 `bank conflict` 的問題(bank size 為 32 ，超過 32 後會重新由 0 開始排)。

```
#pragma unroll 32
for (int k = 0; k < BLOCK_SIZE; k++) {
    A[innerI][innerJ]
    = (A[innerI][k] + Diagonal[k][innerJ]) <
       A[innerI][innerJ] ?
      (A[innerI][k] + Diagonal[k][innerJ]) :
       A[innerI][innerJ];
    // ... omit

    B[innerI][innerJ]
    = (Diagonal[innerI][k] + B[k][innerJ]) <
       B[innerI][innerJ] ?
      (Diagonal[innerI][k] + B[k][innerJ]) :
       B[innerI][innerJ];
    // ... omit
}
```

最終讀取完後，再將資料從 shared memory 寫回 device memory。

```
dist[i*BLOCK_SIZE*n + Round*BLOCK_SIZE + innerI*n + innerJ] = A[innerI][innerJ];
// ... omit
```

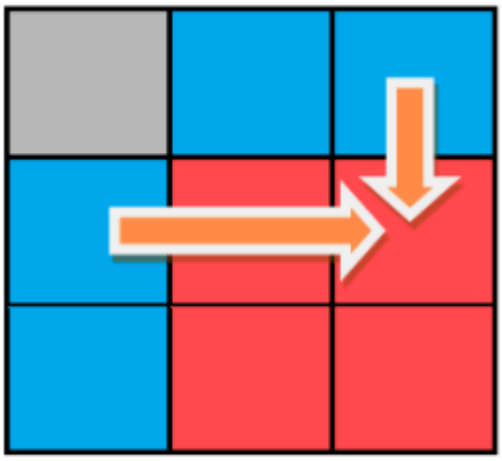
```
dist[Round*BLOCK_SIZE*n + i*BLOCK_SIZE + innerI*n + innerJ] = B[innerI][innerJ];
// ... omit
```

Phase 3

相似於 Phase 2 ，但此部分總共有 `(Total_V/Matrix_V)^2` 個 `cuda block` ，每個 `cuda block` 需負責一個 `Matrix Block` 的計算工作。由於在 Phase 1 與 Phase 2 中，已經處理過一個 Diagonal Matrix 與其 橫縱軸 的 Matrix ，Phase 3 須先略過這些已計算的 block。

```
const int j = blockIdx.x;
const int i = blockIdx.y;
if (i == Round && j == Round) return;
```

對於每個 `cuda block` 而言，便是 load 各一個 Phase2 橫軸上 + 縱軸上 的 Matrix ，完成自己 matrix block 的計算。計算後，再由 shared memory 寫回 global memory 。



(f) Phase 3

```
C[innerI][innerJ] = dist[i*BLOCK_SIZE*n + j*BLOCK_SIZE + innerI*n + innerJ];
// ... omit

A[innerI][innerJ] = dist[i*BLOCK_SIZE*n + Round*BLOCK_SIZE + innerI*n + innerJ];
// ... omit

B[innerI][innerJ] = dist[Round*BLOCK_SIZE*n + j*BLOCK_SIZE + innerI*n + innerJ];
// ... omit
```

```
__syncthreads();

#pragma unroll 32
for (int k = 0; k < BLOCK_SIZE; k++) {
    C[innerI][innerJ] =
   (A[innerI][k] + B[k][innerJ]) < C[innerI][innerJ] ?
   (A[innerI][k] + B[k][innerJ]) : C[innerI][innerJ];
    // ... omit
}

dist[i*BLOCK_SIZE*n + j*BLOCK_SIZE + innerI*n + innerJ] = C[innerI][innerJ];
// ... omit
```

Profiling Results

在 Profiling 中，我使用 `nvprof` 執行 `p24k1` 的測資測試 `Phase3` Kernel，並搜集 `achieved_occupancy` \ `sm_efficiency` \ `shared_load_throughput` \ `shared_store_throughput` \ `gld_throughput` \ `gst_throughput` 六大 Metrics，實驗結果如下：

Metrics

Aa Metric Name	Min	Max	Avg.
Achieved Occupancy	0.946334	0.948199	0.947345
Multiprocessor Activity	99.98%	99.99%	99.99%
Shared Memory Load Throughput	3318.9GB/s	3429.6GB/s	3376.4GB/s
Shared Memory Store Throughput	270.93GB/s	279.97GB/s	275.62GB/s
Global Load Throughput	203.20GB/s	209.98GB/s	206.72GB/s
Global Store Throughput	67.733GB/s	69.992GB/s	68.905GB/s

Experiment & Analysis

System Spec

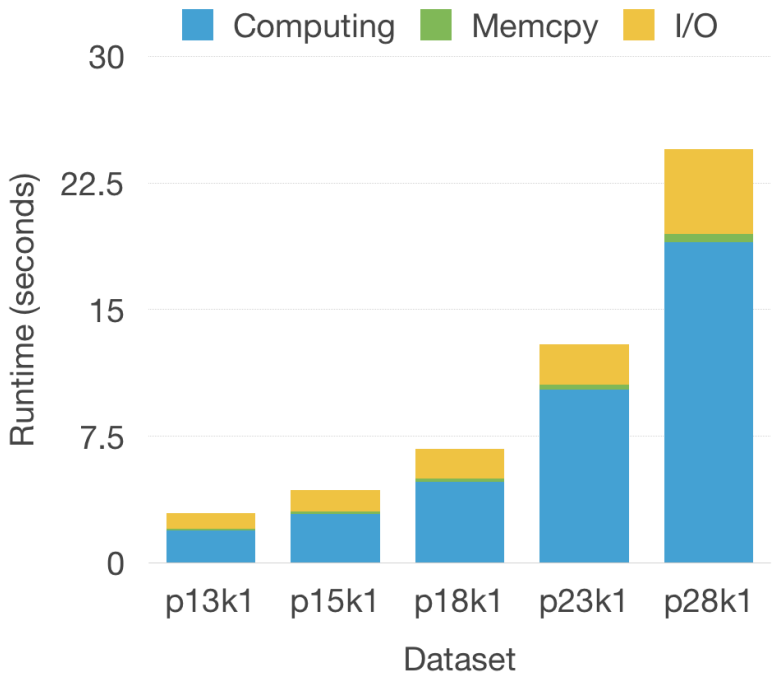
使用學校提供的 hades Cluster。

Time Distribution

為了分別計算不同 input size 下，cuda 執行的 performance，我分別執行 `p13k1` 、 `p15k1` 、 `p18k1` 、 `p23k1` 、 `p28k1` 的測資。 `computing` 與 `memory copy (H2D, D2H)` 的時間計算，我使用 `nvprof` 負責搜集資訊，而對於 CPU 的 IO ，我使用 `clock_gettime(CLOCK_MONOTONIC, &start)` 進行時間採樣。

Time Distribution

	p13k1	p15k1	p18k1	p23k1	p28k1
Computing	1.939	2.91	4.814	10.257	19.011
Memcpy	0.104	0.139	0.195	0.324	0.484
I/O	0.91	1.26	1.73	2.36	5.02



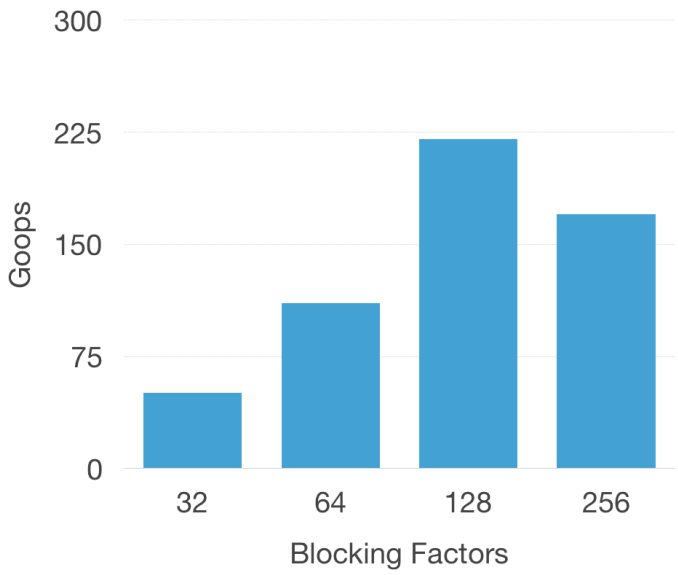
從數據中可見，當資料量逐步升級時，I/O Time 與 Computation Time 的成長速度皆較高，兩者佔據的時間也較多。然則，IO Time 是全部實作中唯一無法避免的，因此如何提升 CUDA Computation Time ，也是本次實作的其一目標。而 CUDA 的 Computation Time ，也通常與其 access data 的效率有絕對正比關係，因此如何提升 memory access ，也是可關注的焦點。

Blocking Factor

計算 Blocking Factor 的 `Bandwidth` 與 `Gops` ，首先須計算兩者的計算複雜度與空間複雜度。我參考這篇文章 (https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/) 進行計算。

實驗中，我使用 `p11k1` 進行實驗計算，該測資有 `11000` 個 `n` ，在我們的實驗中會擴展到 `11008` 個 n 。

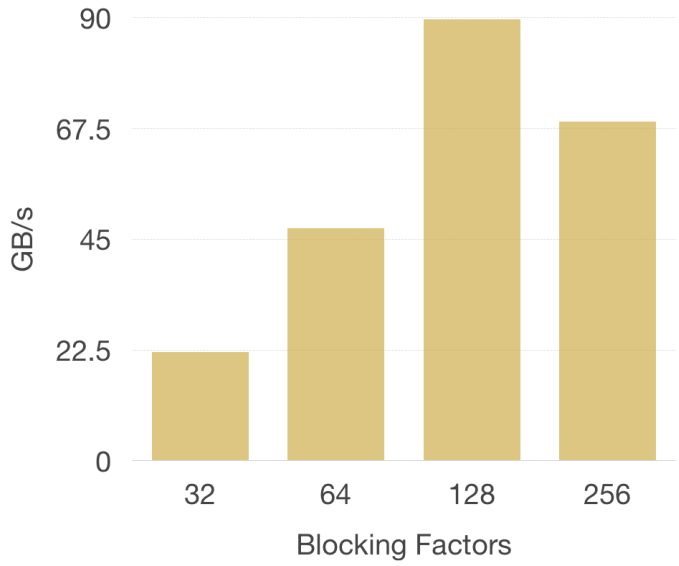
對於 `GOps` 而言，需先觀察目前的計算量：當我們綜觀最大量的 Phase 3 ，所執行的 k-loop 迴圈時，即可發現每次matrix element 更新中，基本上都運行 10 個 add operation (array index 計算與大小計算) ，總共有 `10*N^3 = 10*11008^3 = 13,339,061,125,120` 種計算量。



對於 `Bandwidth` 而言，需先觀察目前的儲存量：

$$BW_{Effective} = (R_B + W_B)/(t * 10^9)$$

其中 Rb 為 kernel read 的量， Wb 為 kernel write back 的量。依 kernel phase 3 為例，基本上需讀取 3 個 matrix + 寫回 1 個 matrix ，total size 為 `4 R+W` 量，也就是 `4*N^3 = 5,335,624,450,048`。

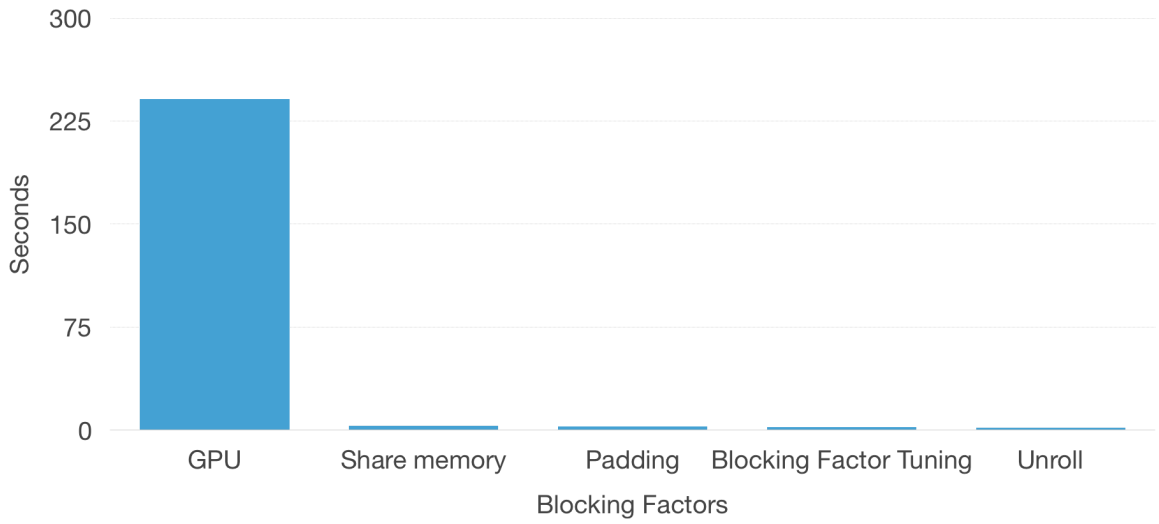


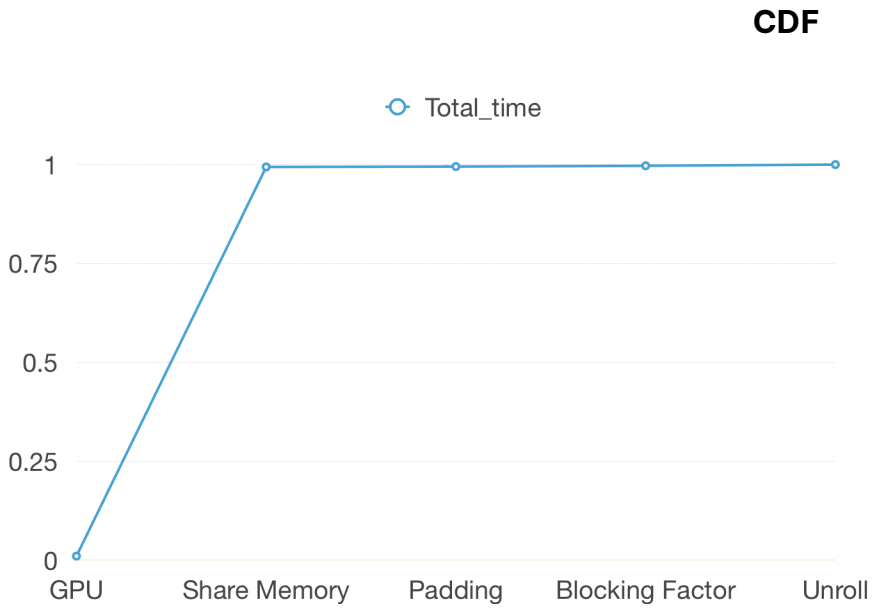
Blocking Factor 經測試後，若在 128 有較佳的 Performance 。由於本次實作的其一 bottleneck 為 memory access ，當數字小於 128 時，會花費過多時間進行 memory copy ，導致效率較差。當數至設定超過 128 時，會疊加過多的計算資源在單一 ALU 上，無法有效平行計算資源，導致效率也較差。

但調高至 128 後， 128×128 的 shared matrix memory 即無法 allocate ，因為 share memory 的容量不足。所以最終仍選定 64 blocking factor 作為最終輸出。

Optimization

本次實作了所有的 Optimization 方法，我拆解其中少數的優化進行分別測試，分別為：GPU 、Share Memory 、Padding 、Blocking Factor Tuning 、Unroll 。

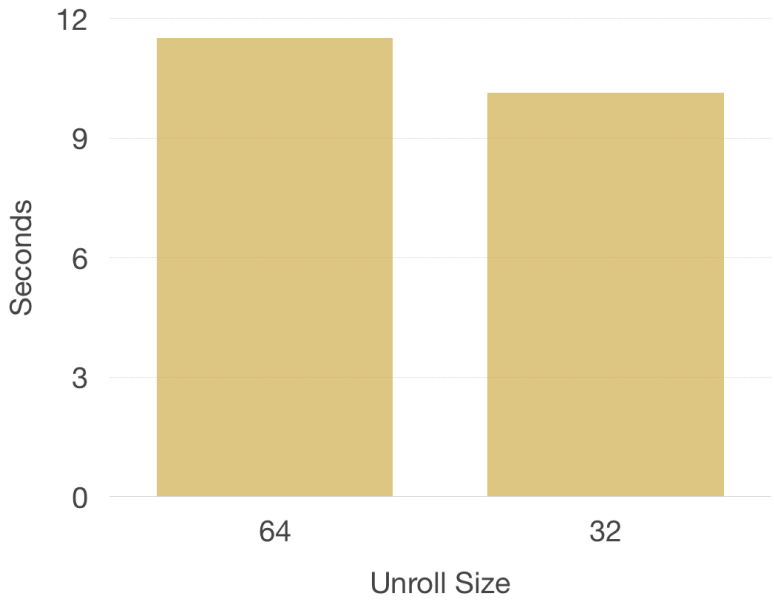




經過測試， Share Memory 的提升效率最高，加快約 100x 的效率。其餘優化，也有逐步將計算效率提高。也因此，實作中為了擁有 share memory 的優化，我們需將 BlockSize 下降到 64×64 的維度。

Other Experiment

Unroll Size 的最佳化設定



依照常理，我們將 for-loop 進行更多次的 unroll 時，理論上 Performance 會更佳，然則在這次實作中得出相反結論。當我們設定 Unroll 為 32 時，相對於 64 有較好的 Performance 。個人推測原因在於，設定一個 Warp 大小的 thread size ，同時進行 concurrent 運算時，會遇見較少的 bank conflict，導致效果更佳。

三元表示式

當我們用 `三元表示式 (Tenary Expression)` ，替代 `if-else` 的方法時，即可避免 CUDA ADU 的運算 Pipeline ，讓 compiler 以將近 2 倍的加快時間執行。

Conclusion

這次實作中，熟悉到不同 cuda 的 optimization 的方法。個人認為困難點在於，Optimization 方法很多，當套用任一方法時，也須顧慮到是否違反其餘 Optimization，反而讓效率變差。同時，因為這次 APSP Algorithm 套用在 CUDA 上，基本上需專注處理 Memory Bound 的問題，因此也重新理解許多 CUDA Memory Architecture 的問題，對 GPU Arch 也更加熟悉。