HW3

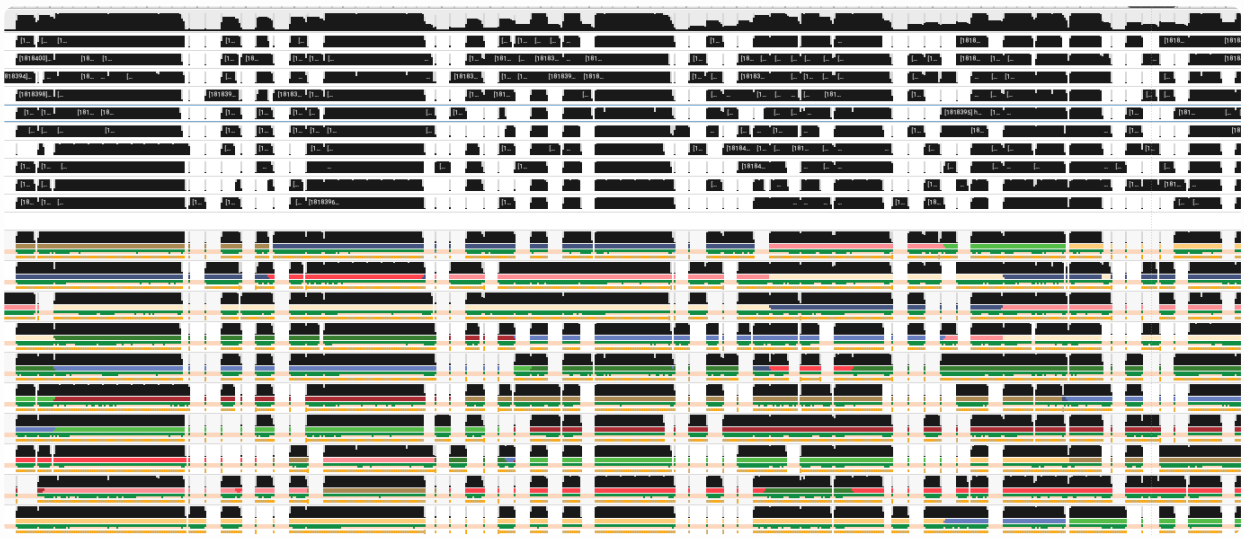
> 112062706 林泳成

Implementation

HW3-1

```
void cal(
    int B, int Round, int block_start_x, int block_start_y, int block_width, int block_height) {
#pragma omp parallel num_threads(NUM_THREADS)
{
    int block_end_x = block_start_x + block_height;
    int block_end_y = block_start_y + block_width;
    #pragma omp for schedule(dynamic) collapse(2)
    for (int b_i = block_start_x; b_i < block_end_x; ++b_i) {
        for (int b_j = block_start_y; b_j < block_end_y; ++b_j) {
            // To calculate B*B elements in the block (b_i, b_j)
            // For each block, it need to compute B times
            int block_internal_start_x = b_i * B;
            int block_internal_end_x = (b_i + 1) * B > n ? n : (b_i + 1) * B;
            int block_internal_start_y = b_j * B;
            int block_internal_end_y = (b_j + 1) * B > n ? n : (b_j + 1) * B;

            // if (block_internal_end_x > n) block_internal_end_x = n;
            // if (block_internal_end_y > n) block_internal_end_y = n;
            for (int k = Round * B; k < (Round + 1) * B && k < n; ++k) {
                // To calculate original index of elements in the block (b_i, b_j)
                // For instance, original index of (0,0) in block (1,2) is (2,5) for V=6,B=2
                for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
                    int dik = Dist[i][k];
                    if (dik == INF) continue;
                    #pragma omp reduction(+:Dist)
                    for (int j = block_internal_start_y; j < block_internal_end_y; ++j) {
                        if (Dist[k][j] != INF && dik + Dist[k][j] < Dist[i][j]) {
                            Dist[i][j] = dik + Dist[k][j];
                        }
                    }
                }
            }
        }
```

原本是用openMP直接對blocked floyd warshall做修改。但是好像是每個子工作太大，沒辦法做到太fine grained



用nsys看執行狀況發現等待的情況很嚴重，應該是子工作太大導致的

這個方法因為SIMD包住的範圍包含了三個額外的for迴圈做陣列更新，即便是用schedule(dynamic)，也會因為一個thread跑一次子工作太久導致所有thread都在等待

每次結束SIMD的迴圈時，都會額外有一個process獨自執行，其他process都沒工作。好像是因為openMP在做完SIMD操作後會附上一個barrier，之後會由主程式做資源管理，負責在runtime做資源

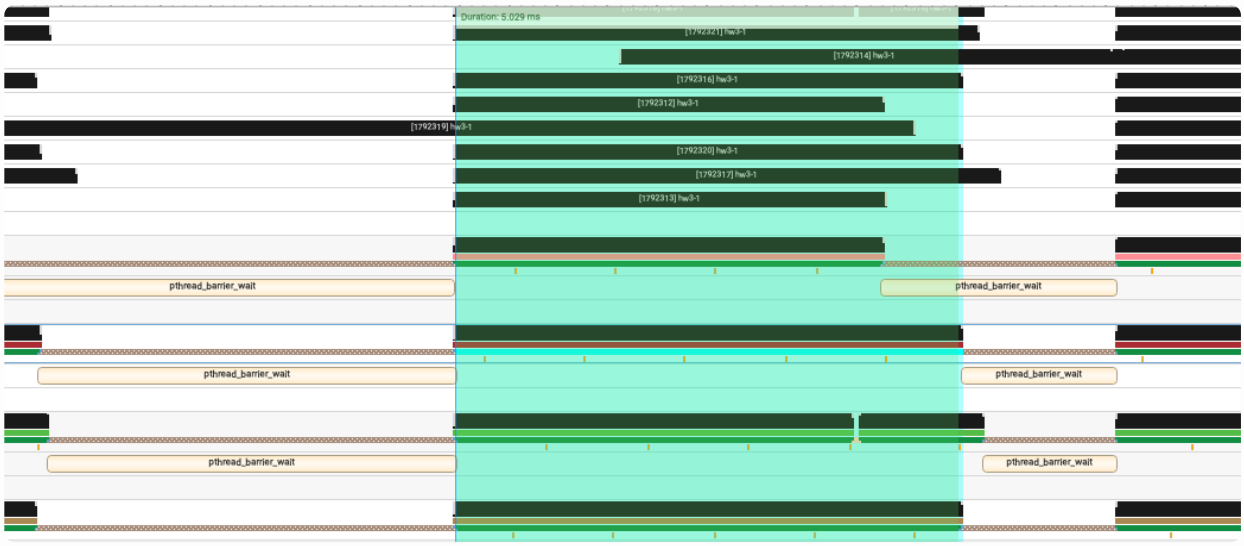
回收和調度
所以後面優化就是想優化thread資源分配的overhead
如果只在程式一開始初始化一次thread就可以避免中間執行花太多時間在資源管理

優化後

```
int jobPerThread = ceil(n * n, NUM_THREADS);
```

```
curX = *tid * jobPerThread;
for(int i = 0; i < jobPerThread; ++i){
    if(Dist[curX/n][curX%n] > Dist[curX/n][k] + Dist[k][curX%n])
        Dist[curX/n][curX%n] = Dist[curX/n][k] + Dist[k][curX%n];
    ++curX;
}
pthread_barrier_wait(&barrier);
```

由於之前的做法工作沒辦法切得很好，會有空隙，所以想說用pthread在每一輪就先把工作切得剛剛好，每個thread在每一輪就是處理ceil(n*n, NUM_THREAD)的工作量
結果表現好像沒有太好，其中一個部分是每個thread被fork出來的時間是由for loop一個一個分配出來的，所以先執行的thread就需要等待較慢初始化的thread



每一輪計算都會花5毫秒

```
for(k = 0;k < n;++k){
    for(i = *tid;i < n;i += NUM_THREADS){
        tmpDist = Dist[i][k];
        for(j = 0;j < n;++j){
            if(Dist[i][j] > tmpDist + Dist[k][j]){
                Dist[i][j] = tmpDist + Dist[k][j];
            }
        }
    }
    pthread_barrier_wait(&barrier);
```

因為之前的做法每一輪的計算會花很多時間，所以後來去看了cache要怎麼存取才會快
就發現主要是cache miss花太多時間，就連之後又測試了__builtin_prefetch都沒辦法在不同thread之間maintain cache，導致整體access時間增加

所以就改成每個thread同時只能抓一行的陣列來做運算，結果大幅改善cache miss的問題

每一輪計算只花4.7毫秒

HW3-2、HW3-3

```
for (int r = 0; r < round; ++r) {
    printf("%d %d\n", r, round);
    fflush(stdout);
    /* Phase 1*/
    cal1<<<gridsPerBlock1, threadsPerBlock>>>(r, deviceDist, pitch / sizeof(int));

    /* Phase 2*/
    cal2<<<gridsPerBlock2, threadsPerBlock>>>(r, deviceDist, pitch / sizeof(int));

    /* Phase 3*/
    cal3<<<gridsPerBlock3, threadsPerBlock>>>(r, deviceDist, pitch / sizeof(int));
}
```

```
for (int r = 0; r < round; ++r) {
    // 擁有這一輪需要的row的device釋出這個row給另一個device
    if (r >= yStart && r < yStart + gridsPerBlock3.y) {
        cudaMemcpy2DAsync(hostDist + r * B * pad_n, pad_n * sizeof(int), (char*)deviceDist[omp_get_thread_num()] + r * B * pitch, pitch, pad_n * sizeof(int), B, cudaMemcpyDeviceToHos
    }
    cudaStreamSynchronize(stream);
    #pragma omp barrier

    if (r < yStart || r >= yStart + gridsPerBlock3.y) {
        cudaMemcpy2DAsync((char*)deviceDist[omp_get_thread_num()] + r * B * pitch, pitch, hostDist + r * B * pad_n, pad_n * sizeof(int), pad_n * sizeof(int), B, cudaMemcpyHostToDevic
    }
    cal1<<<gridsPerBlock1, threadsPerBlock>>>(r, deviceDist[omp_get_thread_num()], pitch / sizeof(int));
    cal2<<<gridsPerBlock2, threadsPerBlock>>>(r, deviceDist[omp_get_thread_num()], pitch / sizeof(int));
    cal3<<<gridsPerBlock3, threadsPerBlock>>>(r, deviceDist[omp_get_thread_num()], pitch / sizeof(int), yStart);
}
```

資料的分配上都是照著每個phase要計算的工作去切

只有在HW3-3是把整個母工作分成上下兩大塊個別計算

HW3-2和HW3-3我都是選擇blocking factor 64、block裡的thread都是(32X32)。

Blocking factor選64是因為一個block是32X32個thread，而且share memory又可以長寬各開到64X64，所以在嘗試32和64兩種blocking factor後選擇效能比較好的64。

至於block的thread分配則是選擇講義裡建議的x, y各是32倍數。
至於grid的分布則是把graph的長寬除以block大小。

```
Device 0: "NVIDIA GeForce GTX 1080"
  CUDA Driver Version / Runtime Version          12.6 / 12.6
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 8107 MBytes (8500871168 bytes)
  (20) Multiprocessors, (128) CUDA Cores/MP:     2560 CUDA Cores
  GPU Max Clock rate:                            1835 MHz (1.84 GHz)
  Memory Clock rate:                             5005 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  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)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
```

```
#pragma omp parallel num_threads(2)
{
    cudaSetDevice(omp_get_thread_num());
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    dim3 gridsPerBlock3(round, (omp_get_thread_num() == 1) && (blockNum & 1) ? ceil(blockNum, 2) : blockNum / 2);
    // y座標從第幾個block開始
    int yStart = blockNum / 2 * omp_get_thread_num();
    cudaMallocPitch(&deviceDist[omp_get_thread_num()], &pitch, pad_n * sizeof(int), pad_n);
    // cudaMemcpy2DAsync(deviceDist[omp_get_thread_num()], pitch, hostDist, pad_n * sizeof(int), pad_n * sizeof(int), pad_n, cudaMemcpyHostToDevice, stream);
    cudaMemcpy2DAsync((char*)deviceDist[omp_get_thread_num()] + yStart * B * pitch, pitch, hostDist + yStart * B * pad_n, pad_n * sizeof(int), pad_n * sizeof(int), gridsPerBlock3.y *
    B, cudaMemcpyHostToDevice, stream);
    cudaStreamSynchronize(stream);
    for (int r = 0; r < round; ++r) {
        // 擁有這一輪需要的row的device釋出這個row給另一個device
        if (r >= yStart && r < yStart + gridsPerBlock3.y) {
            cudaMemcpy2DAsync(hostDist + r * B * pad_n, pad_n * sizeof(int), (char*)deviceDist[omp_get_thread_num()] + r * B * pitch, pitch, pad_n * sizeof(int), B,
            cudaMemcpyDeviceToHost, stream);
        }
        cudaStreamSynchronize(stream);
        #pragma omp barrier

        if (r < yStart || r >= yStart + gridsPerBlock3.y) {
            cudaMemcpy2DAsync((char*)deviceDist[omp_get_thread_num()] + r * B * pitch, pitch, hostDist + r * B * pad_n, pad_n * sizeof(int), pad_n * sizeof(int), B,
            cudaMemcpyHostToDevice, stream);
        }
        cal1<<<gridsPerBlock1, threadsPerBlock>>>(r, deviceDist[omp_get_thread_num()], pitch / sizeof(int));
        cal2<<<gridsPerBlock2, threadsPerBlock>>>(r, deviceDist[omp_get_thread_num()], pitch / sizeof(int));
        cal3<<<gridsPerBlock3, threadsPerBlock>>>(r, deviceDist[omp_get_thread_num()], pitch / sizeof(int), yStart);
    }
    cudaMemcpy2DAsync(hostDist + yStart * B * pad_n, pad_n * sizeof(int), (char*)deviceDist[omp_get_thread_num()] + yStart * B * pitch, pitch, pad_n * sizeof(int), gridsPerBlock3.y *
    B, cudaMemcpyDeviceToHost, stream);
    cudaStreamSynchronize(stream);
    cudaStreamDestroy(stream);
}
```

HW3-3因為會把整個graph切成上下兩大塊，所以在溝通上會是根據每一輪，擁有那一round資料的人把資料丟出來給沒有的GPU拿。這樣就可以讓兩個GPU平行計算phase 3，但是phase 1因為只需要一個GPU算，phase 2因為工作量少所以效果都有限。最後再把兩個GPU上的graph組合起來就好了。

實作上有試著改變access pattern來解決bank conflict(因為phase 2會剛好用完share memory，就不能用padding)，但是效果很差。
其餘部分就是分工把整塊block搬到share memory上。

```
sharedCenter[ty * B + tx] = deviceDist[centerY * pitch + centerX];
sharedCenter[(ty+halfB) * B + tx] = deviceDist[(centerY+halfB) * pitch + centerX];
sharedCenter[ty * B + (tx+halfB)] = deviceDist[centerY * pitch + (centerX+halfB)];
sharedCenter[(ty+halfB) * B + (tx+halfB)] = deviceDist[(centerY+halfB) * pitch + (centerX+halfB)];
sharedCol[ty * B + tx] = deviceDist[colY * pitch + colX];
sharedCol[(ty+halfB) * B + tx] = deviceDist[(colY+halfB) * pitch + colX];
sharedCol[ty * B + (tx+halfB)] = deviceDist[colY * pitch + (colX+halfB)];
sharedCol[(ty+halfB) * B + (tx+halfB)] = deviceDist[(colY+halfB) * pitch + (colX+halfB)];
sharedRow[ty * B + tx] = deviceDist[rowY * pitch + rowX];
sharedRow[(ty+halfB) * B + tx] = deviceDist[(rowY+halfB) * pitch + rowX];
sharedRow[ty * B + (tx+halfB)] = deviceDist[rowY * pitch + (rowX+halfB)];
sharedRow[(ty+halfB) * B + (tx+halfB)] = deviceDist[(rowY+halfB) * pitch + (rowX+halfB)];
__syncthreads();

#pragma unroll 64
for (int k = 0; k < B; ++k) {
    // sharedCol[ty * B + tx] = min(sharedCol[ty * B + tx], sharedCol[ty * B + k] + sharedCenter[k * B + tx]);
    // sharedRow[ty * B + tx] = min(sharedRow[ty * B + tx], sharedCenter[ty * B + k] + sharedRow[k * B + tx]);
    // Update column
    if (sharedCol[ty * B + k] + sharedCenter[k * B + tx] < sharedCol[ty * B + tx]) {
        sharedCol[ty * B + tx] = sharedCol[ty * B + k] + sharedCenter[k * B + tx];
    }
    if (sharedCol[(ty+halfB) * B + k] + sharedCenter[k * B + tx] < sharedCol[(ty+halfB) * B + tx]) {
        sharedCol[(ty+halfB) * B + tx] = sharedCol[(ty+halfB) * B + k] + sharedCenter[k * B + tx];
    }
    if (sharedCol[ty * B + k] + sharedCenter[k * B + (tx+halfB)] < sharedCol[ty * B + (tx+halfB)]) {
        sharedCol[ty * B + (tx+halfB)] = sharedCol[ty * B + k] + sharedCenter[k * B + (tx+halfB)];
    }
    if (sharedCol[(ty+halfB) * B + k] + sharedCenter[k * B + (tx+halfB)] < sharedCol[(ty+halfB) * B + (tx+halfB)]) {
        sharedCol[(ty+halfB) * B + (tx+halfB)] = sharedCol[(ty+halfB) * B + k] + sharedCenter[k * B + (tx+halfB)];
    }
    // Update row
    if (sharedCenter[ty * B + k] + sharedRow[k * B + tx] < sharedRow[ty * B + tx]) {
        sharedRow[ty * B + tx] = sharedCenter[ty * B + k] + sharedRow[k * B + tx];
    }
    if (sharedCenter[(ty+halfB) * B + k] + sharedRow[k * B + tx] < sharedRow[(ty+halfB) * B + tx]) {
        sharedRow[(ty+halfB) * B + tx] = sharedCenter[(ty+halfB) * B + k] + sharedRow[k * B + tx];
    }
    if (sharedCenter[ty * B + k] + sharedRow[k * B + (tx+halfB)] < sharedRow[ty * B + (tx+halfB)]) {
        sharedRow[ty * B + (tx+halfB)] = sharedCenter[ty * B + k] + sharedRow[k * B + (tx+halfB)];
    }
    if (sharedCenter[(ty+halfB) * B + k] + sharedRow[k * B + (tx+halfB)] < sharedRow[(ty+halfB) * B + (tx+halfB)]) {
```

也有使用stream把傳資料和計算分成兩個stream，但是效果也有限。因為總共只需要傳來回一次，就要為了這來回一次創建一個stream，導致變更慢。

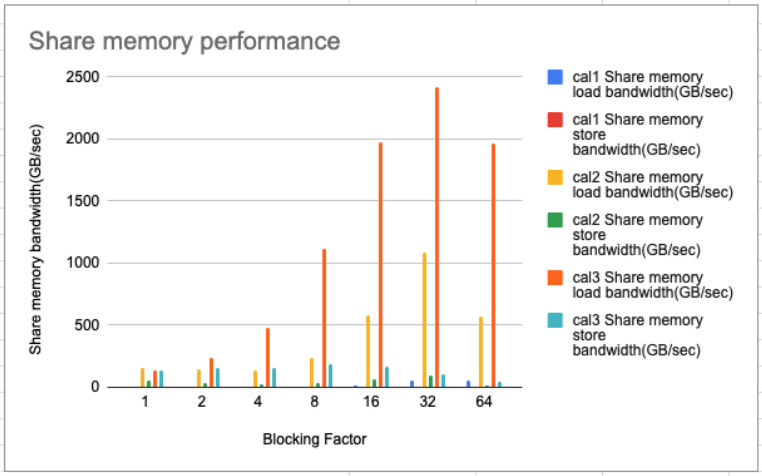
```
cudaStream_t stream;
cudaStreamCreate(&stream);

cudaHostRegister(hostDist, pad_n * pad_n * sizeof(int), cudaHostAllocDefault);
cudaMallocPitch(&deviceDist, &pitch, pad_n * sizeof(int), pad_n);
cudaMemcpy2DAsync(deviceDist, pitch, hostDist, pad_n * sizeof(int), pad_n * sizeof(int), pad_n, cudaMemcpyHostToDevice, stream);
// cudaMalloc(&deviceDist, pad_n * pad_n * sizeof(int));
// cudaMemcpy(deviceDist, hostDist, pad_n * pad_n * sizeof(int), cudaMemcpyHostToDevice);

cudaGetDeviceProperties(&prop, DEV_NO);
printf("maxThreadsPerBlock = %d, sharedMemPerBlock = %d\n", prop.maxThreadsPerBlock, prop.sharedMemPerBlock);

block_FW(deviceDist);
cudaMemcpy2DAsync(hostDist, pad_n * sizeof(int), deviceDist, pitch, pad_n * sizeof(int), pad_n, cudaMemcpyDeviceToHost, stream);
cudaStreamSynchronize(stream);
cudaStreamDestroy(stream);
// cudaMemcpy(hostDist, deviceDist, pad_n * pad_n * sizeof(int), cudaMemcpyDeviceToHost);
```

但是對於HW3-3多GPU來說因為每一輪都會溝通一次來回，所以多創一個stream就有價值，因此有顯著加速。

Profiling Results

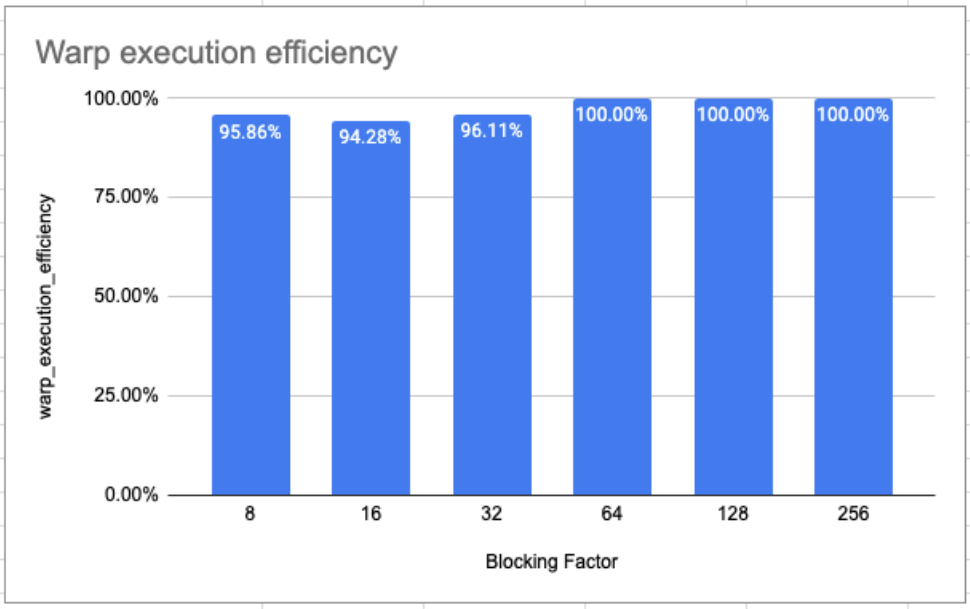
使用c10.1測資當作input，實驗不同blocking factor對GPU效能的各項指標有什麼影響

Blocking Factor	1	2	4	8	16	32	64
cal1 Share memory load bandwidth(GB/sec)	0.22088	0.37278	0.65714	2.2523	11.658	51.964	58.861
cal1 Share memory store bandwidth(GB/sec)	0.055219	0.053253	0.069555	0.24601	1.1143	4.339	1.1139
cal2 Share memory load bandwidth(GB/sec)	154.39	144.73	133.32	233.25	576.75	1086.6	567.96
cal2 Share memory store bandwidth(GB/sec)	57.897	32.563	21.383	32.109	61.984	91.783	10.355
cal3 Share memory load bandwidth(GB/sec)	140.89	241.2	475.84	1119.5	1971	2416.6	1966.7
cal3 Share memory store bandwidth(GB/sec)	140.89	160.8	158.61	186.59	164.25	100.69	40.973



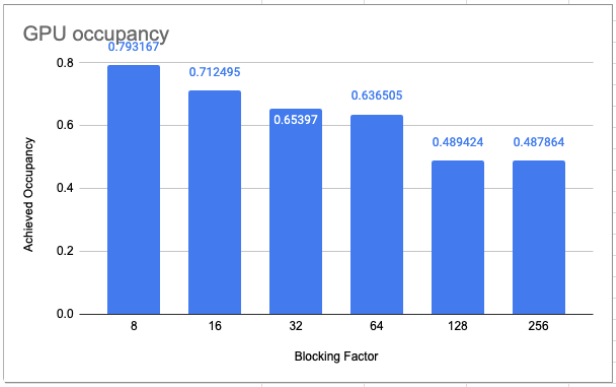
▲不同blocking factor影響GPU share memory在不同kernel call load和store操作的bandwidth

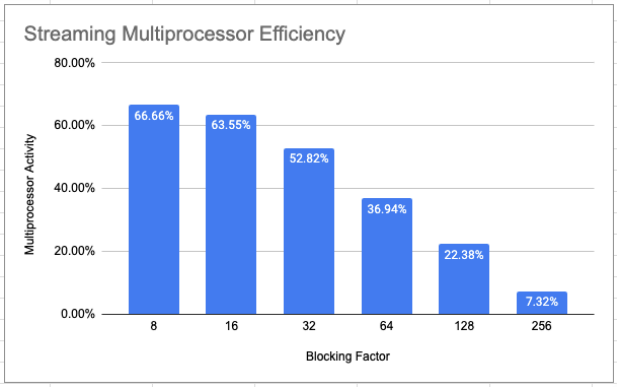
Blocking Factor	8	16	32	64	128	256
warp_execution_efficiency	95.86%	94.28%	96.11%	100.00%	100.00%	100.00%



▲不同blocking factor影響GPU warp的使用效率

Blocking Factor	8	16	32	64	128	256
Achieved Occupancy	0.793167	0.712495	0.65397	0.636505	0.489424	0.487864
Multiprocessor Activity	66.66%	63.55%	52.82%	36.94%	22.38%	7.32%

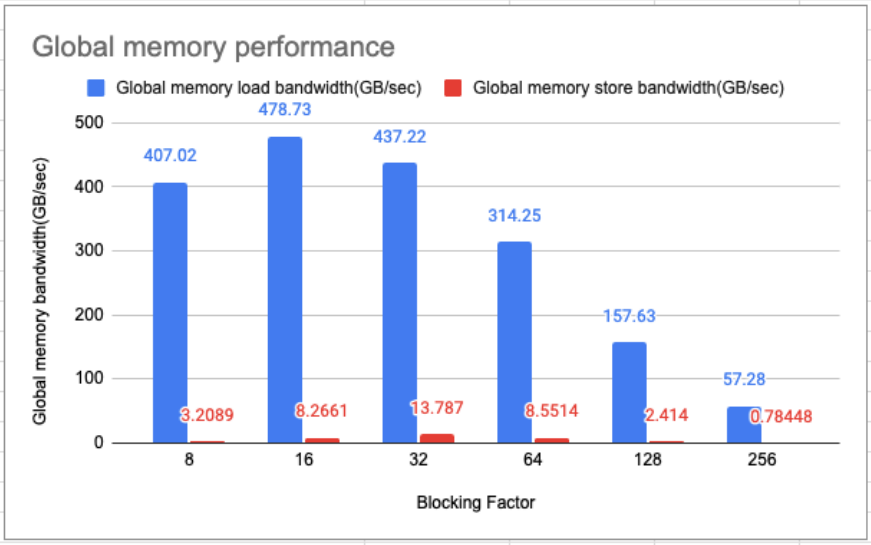




▲不同blocking factor影響GPU core的佔用率和Streaming Multiprocessor的使用效率

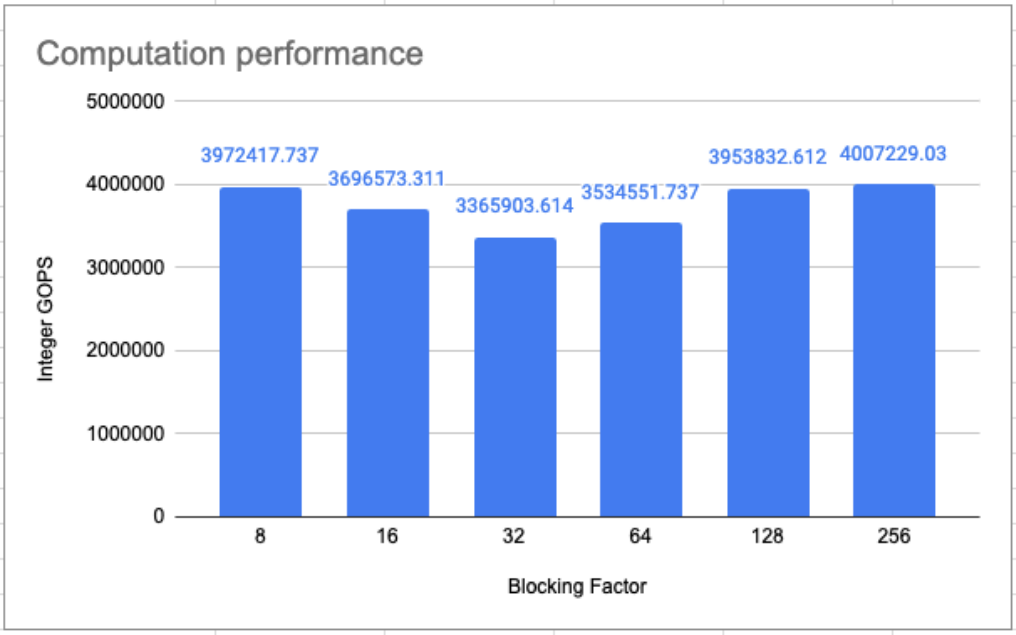
Experiment & Analysis

Blocking Factor	8	16	32	64	128	256
Global memory load bandwidth(GB/sec)	407.02	478.73	437.22	314.25	157.63	57.28
Global memory store bandwidth(GB/sec)	3.2089	8.2661	13.787	8.5514	2.414	0.78448



▲不同blocking factor影響GPU global memory load和store操作的bandwidth

Blocking Factor	8	16	32	64	128	256
inst_integer	94368973	37891389	17294777	8736341	4504746	2570126
Execution time(sec)	23.75605469	10.25040918	5.138227051	2.471697021	1.139336548	0.641372375
Integer GOPS	3972417.737	3696573.311	3365903.614	3534551.737	3953832.612	4007229.03

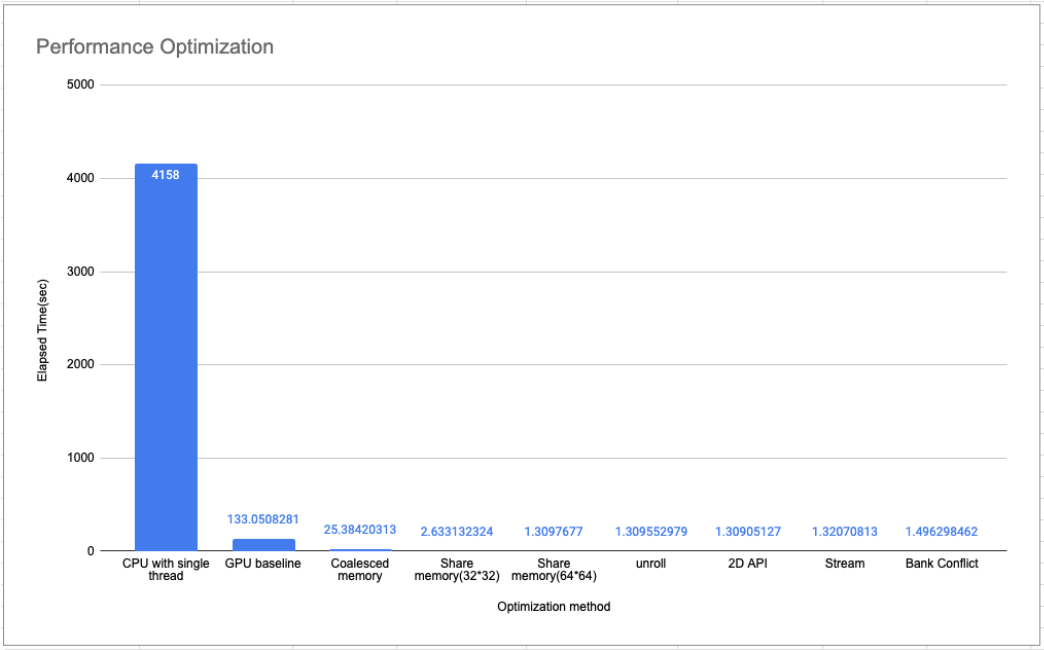


▲不同blocking factor影響GPU 每秒整數計算量

Optimization

用p11k1測資當作input資料，測試每多增加一種優化方法，能加速多少

Optimization method	CPU with single thread	GPU baseline	Coalesced memory	Share memory(32*32)	Share memory(64*64)	unroll	2D API	Stream	Bank Conflict
Elapsed Time(sec)	4158	133.0508281	25.38420313	2.633132324	1.3097677	1.309552979	1.30905127	1.32070813	1.496298462



由於CPU會超時，所以實驗時是把(一輪的時間 X round)當作結果

CPU baseline是完全沒有優化的方法

GPU baseline是把每個pixel都丟給一個GPU thread計算，所以可以快很多倍

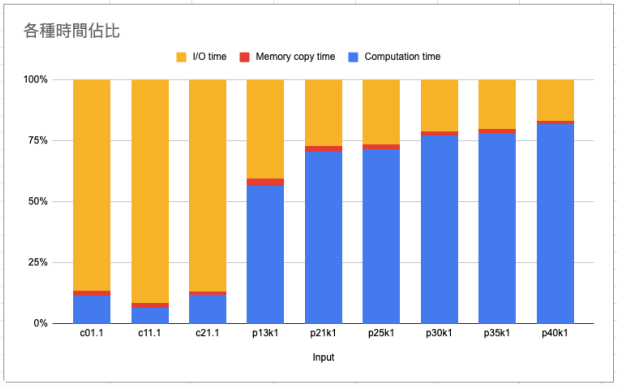
後面實作unroll、stream、bank conflict方法好像對整體影響不大，甚至有可能讓整體效能變差

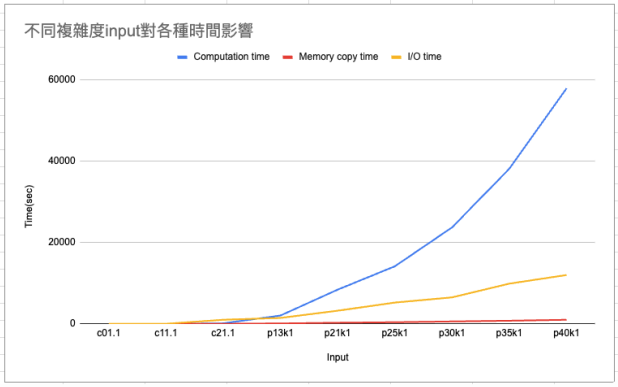
Time Distribution

根據不同的scale選擇了不同的測資當作input

單GPU實驗

Input	c01.1	c11.1	c21.1	p13k1	p21k1	p25k1	p30k1	p35k1	p40k1
Node	5	555	5000	13000	20959	25000	30000	34921	39857
Edge	10	146912	10723117	1829967	987205	5780158	3907489	28054826	4232291
Computation time	0.30304	1.01888	136.931335	2046.70459	8403.708008	14141.80762	23698.09766	38090.11328	57815.54688
Memory copy time	0.05344	0.32896	16.984064	109.131939	279.278855	406.098175	591.594727	773.557373	1004.743744
I/O time	2.277656	14.414354	1011.483093	1459.574615	3232.197632	5234.843505	6508.891358	9878.794434	11993.39258
Total time(ms)	2.634136	15.762194	1165.398492	3615.411144	11915.1845	19782.7493	30798.58374	48742.46509	70813.6832

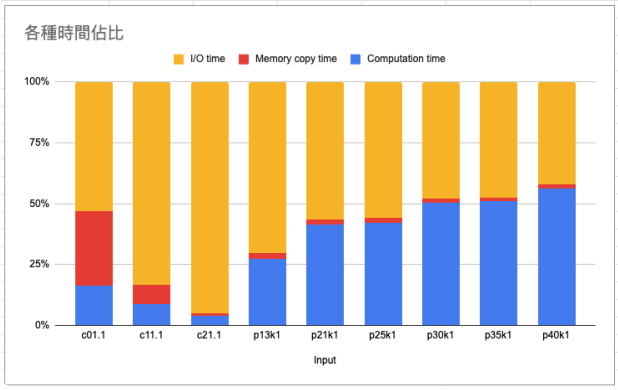


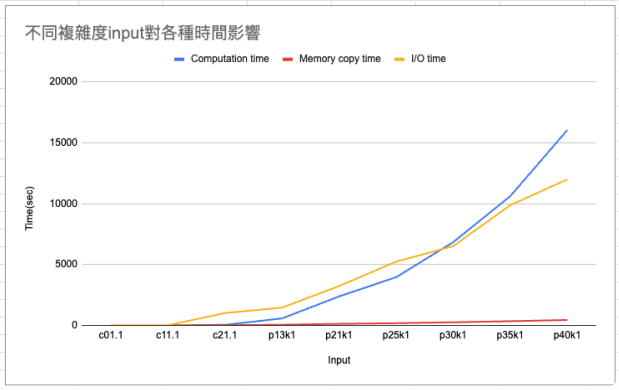


在計算工作複雜度較小時，I/O幾乎等於整體運算時間。但隨著vertex和edge的數量上升，計算量變大，計算才逐漸變成程式的bottleneck。

單GPU跑AMD實驗

Input	c01.1	c11.1	c21.1	p13k1	p21k1	p25k1	p30k1	p35k1	p40k1
Computation time	0.700958	1.523515	43.241486	570.48407	2380.165039	3957.399902	6836.986328	10612.17383	16043.71582
Memory copy time	1.307194	1.321596	8.419497	48.819574	123.781322	175.262612	251.580422	340.663758	442.935669
I/O time	2.277656	14.414354	1011.483093	1459.574615	3232.197632	5234.843505	6508.891358	9878.794434	11993.39258
Total time(ms)	4.285808	17.259465	1063.144076	2078.878259	5736.143993	9367.506019	13597.45811	20831.63202	28480.04407

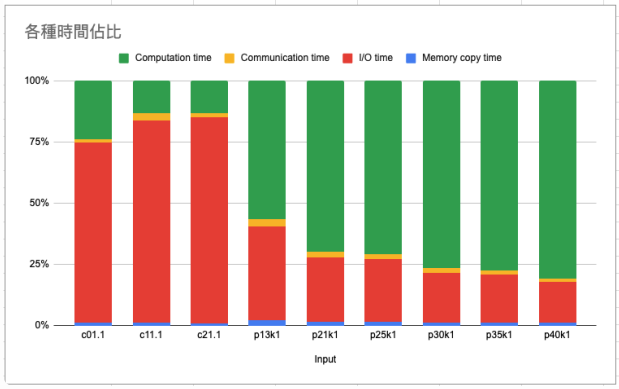


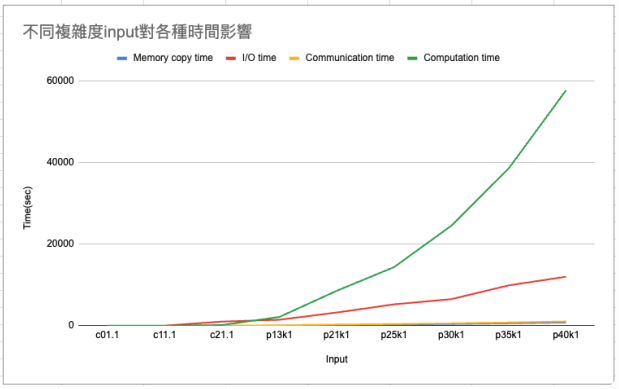


一樣的程式轉成使用AMD的API，並跑在AMD的GPU上，運算速度竟然快了好多。當測到p40k1這筆測資時，I/O時間居然和計算時間沒差多少。而且在做GPU memory copy的時候，還發現nvidia的copy from host to device和device to host的時間差距其實蠻大的。AMD的API對這兩個操作所花的時間幾乎是一樣的。

雙GPU實驗

Input	c01.1	c11.1	c21.1	p13k1	p21k1	p25k1	p30k1	p35k1	p40k1
Computation time	0.740384	2.302144	160.536331	2147.756592	8549.0625	14346.36816	24499.72266	38506.06641	57649.85938
Communication time	0.0472	0.510944	18.413467	113.399483	288.417542	407.381744	584.707092	789.605896	1043.651611
Memory copy time	0.044032	0.221312	12.769152	79.794367	206.621857	294.007965	423.592086	585.690231	790.233215
I/O time	2.277656	14.414354	1011.483093	1459.574615	3232.197632	5234.843505	6508.891358	9878.794434	11993.39258
Total time(ms)	3.109272	17.448754	1203.202043	3800.525057	12276.29953	20282.60138	32016.91319	49760.15697	71477.13678





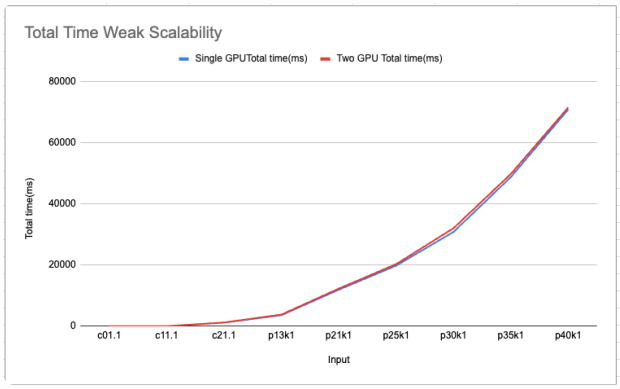
多GPU和單GPU的狀況一樣，都是計算工作的scale決定計算時間和I/O的佔比。

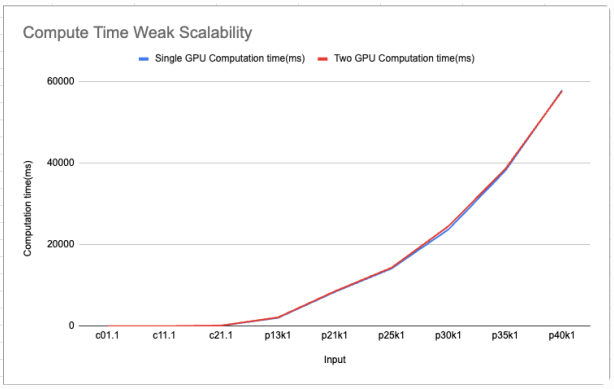
雙GPU跑AMD實驗

```
pp24s086@apollo-login:~/hw3-3$ srun -N1 -n1 -c2 -p amd --gres=gpu:2 hw3-3-amd testcases/c01.1 c01.1.o
ut
srun: error: apollo-amd-vm: task 0: Exited with exit code 127
hw3-3-amd: error while loading shared libraries: libomp.so: cannot open shared object file: No such f
ile or directory
```

雙GPU跑編譯好的AMD code不知道為什麼會遇到不能用OpenMP的問題，所以就沒有測了。

Weak Scalability





對於單GPU和雙GPU而言差距不算太大，但因為實作的關係，多GPU在運作上還比單GPU慢一點點。如果只看計算不看I/O，結果也是一樣。

Conclusion

這次作業有讓我嘗試了很多上課教到的GPU優化技巧。經過不同的實作，我也發現不同的情境下並不是每個優化技巧都是用，有些技巧如stream會需要花時間創建一個stream，若stream只用沒幾次就不用了，會很浪費。又或是2D memory copy API會需要先pin host memory再對齊，才能讓API加速。如果要copy的資料很小且很雜，那麼pin memory加上對齊的時間就會搞垮整個效率，因此要慎重定使用的優化技巧。多GPU在實作後才發現，以目前所學不太有優化的空間，基本上沒有優化還可能比單GPU還糟。