HW3 All-Pairs Shortest Path

student ID: 112062619 **name:** 陳航希

1. Implementation

a. Which algorithm do you choose in hw3-1?

我在hw3-1中參考了助教提供的 seq.cc，實作了openmp版本的Blocked Floyd-Warshall Algorithm, 為了要最大化openmp的好處, 我改了助教將整個區域拆成四等分的寫法, phase2, phase3針對四個區域寫的做法, 改成是針對每個block做更新,以下update_block的寫法, 以做到更簡單的平行化。

```
static inline void update_block(int i_start, int i_end,                int j_start, int j_end, int k_start, int k_end) {
    for (int k=k_start; k<k_end; ++k) {
        int *Dk = Dist[k];
        for (int i=i_start; i<i_end; ++i) {
            int dik = Dist[i][k];
            if (dik == INF) continue;
            int *Di = Dist[i];
            for (int j=j_start; j<j_end; ++j) {
                int nd = dik + Dk[j]; if (nd<Di[j]) Di[j] = nd;}}}}
```

接著使用openmp對phase2, phase3的blocks做update,然後針對每一個blocks使用openmp做平行化, 因為每個blocks之間互相沒有相依性所以可以很容易的做到平行化。下面的是對phase2的同一個column的blocks做的平行化。phase3的平行化也可以使用(: `#pragma omp parallel for collapse(2) schedule(static)` 拆開雙重迴圈做到**`blocks level`**的平行化。

```
#pragma omp parallel for schedule(static)
for (int i=0; i<R; ++i) {
    int i_start = i*B;
    int i_end   = (i+1)*B;
    if (i_end>n) i_end = n;
    if (i==r) continue;
    update_block(i_start, i_end, r*B, (r+1)*B>n ? n : (r+1)*B, k_start, k_end);}
```

b. divide data/ configuration in hw3-2

首先因為想將資料平均分給gpu中的各個threads, 我先將整個nodes的個數original_n拆成是以BSZ=64的大小為一組的tiles, 如果有缺少的話就使用ceil_div補齊缺少的部分。會分成 64x64 的tiles是因為gtx1080的shared memory大小大約為48kb, 因為在之後的phase2-kernel, phase3_kernel 每個block都會使用到 2個64x64的tiles 傳到shared memory當中, 所以需要

預留 至少 2x64x64x4 bytes = 32768 bytes, 也就是32kb 的空間。也剛好 1080 每個blocks 可以有的threads 個數為 32x32=1024 , 可以剛好跟 64x64的 tile對齊，所以選了這個大小作為tile。

```
#define BSZ 64
n = ceil_div(original_n, BSZ) * BSZ;
```

接著根據 blocked floyd-warshell 演算法實作gpu版的phase1, phase2, phase3。
每個block分配 32*32個threads。因為phase1只會對一個pivot block做更新所以只分配一個kernel。phase2會做pivot block上的rows, cols 所以會需要跑R-1個blocks因此分配R-1個kernel, 讓每個blocks可以獨立跑。最後的phase3有相同。三個phase共同更新在device上的 d_Dist 。

```
int R = n/BSZ;
dim3 block(32, 32);
dim3 grid_row(R-1);
dim3 grid_col(R-1);
dim3 grid3(R-1, R-1);
for (int r=0; r<R;r++) {
    phase1_kernel<<<1, block>>>(d_Dist, n, r);
    phase2_row_kernel<<<grid_row, block>>>(d_Dist, n, r);
    phase2_col_kernel<<<grid_col, block>>>(d_Dist, n, r);
    if (R>1)
        phase3_kernel<<<grid3, block>>>(d_Dist, n, r);
}
```

phase1:

每個phase做的事情大致上相，都是要做 “在一個blocks的範圍內固定一個k以後更新dits可能達到的最小值” 。要在gpu上做這件事就會希望blocks中的資訊都在shared memory上所以這裡就讓每個threads處理4個pixels, 這裡的Round代表做到第幾個 pivot block, ty為在這個block中 row的編號，tx為在這個block中column 的編號。每個thread會以 half=BSZ/2 的大小去取pixel, 以此達到loading平衡。

```
__global__ void phase1_kernel(int* __restrict__ d, const int N, const int Round) {
    __shared__ int s[BSZ][BSZ];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int half = BSZ >> 1;
    int base = Round * BSZ;
```

```
  int i0 = base + ty;
  int j0 = base + tx;

  s[ty][tx]               = d[i0*N + j0];
  s[ty][tx+half]          = d[i0*N + (j0 + half)];
  s[ty + half][tx]        = d[(i0+half)*N + j0];
  s[ty + half][tx + half] = d[(i0+half)*N + (j0 + half)];
  __syncthreads();
```

最後在block中做總共BSZ次的更新，獲得pivot block的結果。

```
for (int k=0; k<BSZ; k++) {
  s[ty][tx]               = min(s[ty][tx],               s[ty][k] + s[k][tx]);
  s[ty][tx + half]        = min(s[ty][tx + half],        s[ty][k] + s[k][tx + half]);
  s[ty + half][tx]        = min(s[ty + half][tx],        s[ty + half][k] + s[k][tx]);
  s[ty + half][tx + half] = min(s[ty + half][tx + half], s[ty + half][k] + s[k][tx + half]);
  __syncthreads();
}
```

phase2:

在phase2中我們會對rows, cols做更新。這裡以row為基礎做說明, cols的結果可以以相同方式獲得。因為我們需要有pivot block 跟當前 self block的資訊, 我們準備兩個shared mem中的tensors來做更新。另外因為bid==Round的block已經在phase1的時候做過了 如果遇到就會需要跳過。

```
__global__ void phase2_row_kernel(int* __restrict__ d, const int N, const int Round) {
  __shared__ int sPivot[BSZ][BSZ];
  __shared__ int sSelf[BSZ][BSZ];

  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int half = BSZ >> 1;

  int p = Round * BSZ;

  int bid = blockIdx.x;
  if (bid >= Round) bid++;
```

剩下的部分與phase1的情況相同，只是需要對pivot, self的index進行計算獲得正確結果。

phase3:

phase3的部分也跟phase1類似，是利用phase2更新完的結果去更新其他部分的blocks, 共有 R-1 x R-1個kernels會同時運作來計算剩餘blocks的值。為了要計算第i, j個blocks的值, 我們需要知道對應的 sRow[i, p0], sCol[p0, j] 的值。遇到pivot block的index (i,j) 也需要做加 1 來迴避。其餘的部分跟phase1相同。

```
__global__ void phase3_kernel(int* __restrict__ d, const int N, const int Round) {
  __shared__ int sRow[BSZ][BSZ];
  __shared__ int sCol[BSZ][BSZ];
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int half = BSZ>>1;
  int p0 = Round * BSZ;

  int bx = blockIdx.x + (blockIdx.x >= Round);
  int by = blockIdx.y + (blockIdx.y >= Round);

  int row_i0 = bx * BSZ;
  int col_j0 = by * BSZ;

  int i = row_i0 + ty;
  int j = col_j0 + tx;

  sRow[ty][tx]            = d[i*N + (p0 + tx)];
  sRow[ty][tx + half]     = d[i*N + (p0 + tx + half)];
  sRow[ty + half][tx]     = d[(i+half)*N + (p0 + tx)];
  sRow[ty + half][tx+half] = d[(i+half)*N + (p0 + tx + half)];
  __syncthreads();
```

How do you implement the communication in hw3-3?

為了要有效利用兩張gpu所帶來的好處，我將整個 RxR 個blocks 分成上下兩塊，上半塊給其中一個gpu, 另一塊給另一個gpu, 當pivot block在其中一邊的時候就由那個gpu完成計算，每次做完phase1與phase2 row的計算後, 就會需要把 有pivot block那邊的gpu的資料傳給另一塊gpu, 來讓另一塊gpu做phase2,3的計算, 這樣可以讓兩個gpu之間的通信時間最小化, 缺點為兩邊都需要有完整的dist tensor需要更新，gpu memory的用量較大。剩下kernel部分的計算跟hw3-3相同。

```
int splitBlock = R / 2;       // block-row
int splitRow   = splitBlock * BSZ;
int rowStart[2] = {0,        splitRow};
int rowEnd  [2] = {splitRow, n       };
```

每次做完前半段的run以後需要傳送資料給另一塊。

```
for (int r=0; r<R;r++) {
    int p = r * BSZ;

    int pivotOwner = (p < splitRow) ? GPU0 : GPU1;
    int other      = pivotOwner ^ 1;

    cudaSetDevice(pivotOwner);
    phase1_kernel<<<1, block>>>(d_Dist[pivotOwner], n, r);
    phase2_row_kernel<<<grid_row, block>>>(d_Dist[pivotOwner], n, r);
    cudaDeviceSynchronize();

    cudaMemcpyPeer(
        d_Dist[other] + (size_t)p * n, other,
        d_Dist[pivotOwner] + (size_t)p * n, pivotOwner,
        bandBytes
    );
```

2. Profiling Results (hw3-2)

```
pp25s051@apollo-login:~/hw3/hw3-2/kk5$ srun -p nvidia -N1 -n1 --gres=gpu:1 \
    nvprof --metrics \
    achieved_occupancy,sm_efficiency,shared_load_throughput,shared_store_throughput,gld_throughput,gst_throughput \
    ./hw3-2 /home/pp25/pp25s051/share/hw3/testcases/c20.1 c20.1.out
==1503891== NVPROF is profiling process 1503891, command: ./hw3-2 /home/pp25/pp25s051/share/hw3/testcases/c20.1 c20.1.out
==1503891== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==1503891== Profiling application: ./hw3-2 /home/pp25/pp25s051/share/hw3/testcases/c20.1 c20.1.out
==1503891== Profiling result:
==1503891== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: phase2_row_kernel(int*, int, int)
          79                        achieved_occupancy                     Achieved Occupancy    0.969769    0.991565    0.975455
          79                            sm_efficiency                Multiprocessor Activity      92.27%      94.34%      93.66%
          79                    shared_load_throughput           Shared Memory Load Throughput  2145.2GB/s  2589.2GB/s  2410.6GB/s
          79                   shared_store_throughput          Shared Memory Store Throughput  879.41GB/s  1061.4GB/s  988.19GB/s
          79                            gld_throughput                 Global Load Throughput  26.649GB/s  32.165GB/s  29.945GB/s
          79                            gst_throughput                Global Store Throughput  13.324GB/s  16.082GB/s  14.973GB/s
    Kernel: phase2_col_kernel(int*, int, int)
          79                        achieved_occupancy                     Achieved Occupancy    0.969433    0.976227    0.972839
          79                            sm_efficiency                Multiprocessor Activity      92.06%      93.73%      93.03%
          79                    shared_load_throughput           Shared Memory Load Throughput  2149.3GB/s  2495.3GB/s  2412.5GB/s
          79                   shared_store_throughput          Shared Memory Store Throughput  881.09GB/s  1022.9GB/s  988.20GB/s
          79                            gld_throughput                 Global Load Throughput  26.700GB/s  30.998GB/s  29.970GB/s
          79                            gst_throughput                Global Store Throughput  13.350GB/s  15.499GB/s  14.985GB/s
    Kernel: phase3_kernel(int*, int, int)
          79                        achieved_occupancy                     Achieved Occupancy    0.925652    0.936176    0.929338
          79                            sm_efficiency                Multiprocessor Activity      98.91%      99.65%      99.58%
          79                    shared_load_throughput           Shared Memory Load Throughput  2979.8GB/s  3282.9GB/s  3241.6GB/s
          79                   shared_store_throughput          Shared Memory Store Throughput  124.16GB/s  136.79GB/s  135.07GB/s
          79                            gld_throughput                 Global Load Throughput  186.24GB/s  205.18GB/s  202.60GB/s
          79                            gst_throughput                Global Store Throughput  62.080GB/s  68.393GB/s  67.533GB/s
    Kernel: phase1_kernel(int*, int, int)
          79                        achieved_occupancy                     Achieved Occupancy    0.498026    0.498120    0.498081
          79                            sm_efficiency                Multiprocessor Activity       3.15%       4.54%       4.13%
          79                    shared_load_throughput           Shared Memory Load Throughput  100.78GB/s  125.21GB/s  122.89GB/s
          79                   shared_store_throughput          Shared Memory Store Throughput  36.636GB/s  42.169GB/s  41.389GB/s
          79                            gld_throughput                 Global Load Throughput  577.16MB/s  664.33MB/s  652.03MB/s
          79                            gst_throughput                Global Store Throughput  577.16MB/s  664.33MB/s  652.03MB/s
pp25s051@apollo-login:~/hw3/hw3-2/kk5$
```

3. Experiment & Analysis (Nvidia GPU)

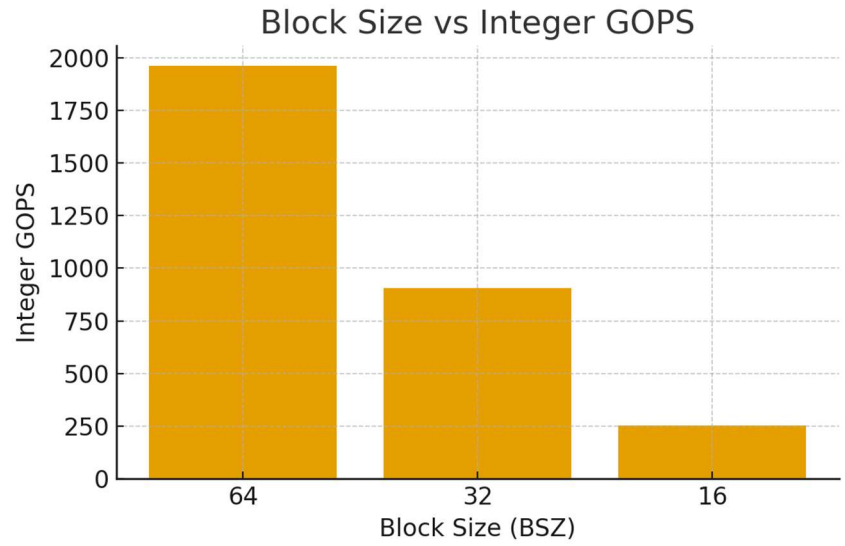
a.System Spec: 以下實驗我使用課堂提供的apallo-gpu執行，實驗資料使用作業測資的c21.1完成，如果有使用其他測資會另外說明。

$$\text{GOPS} = \frac{2n^3}{T_{\text{kernel}} \cdot 10^9}$$

b. Blocking Factor

以下對block size為16, 32, 64的情況，計算對應的gops,下表為記錄不同block size下kernel被count的總次數與所花的時間，可以看到隨著block size的增加gflops呈現穩定的增加，表示 blocking 對效能的改善是顯著的。

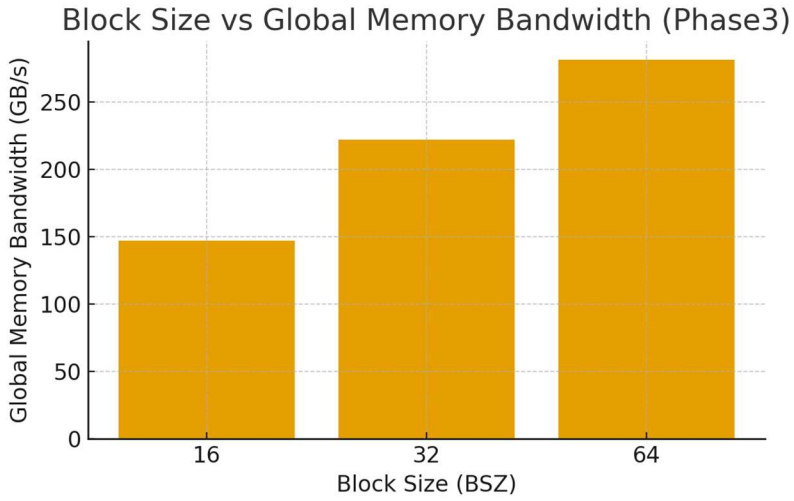
BSZ	N	Total Kernel Calls	Time (ms)	Integer GOPS
64	5000	493039	127.558	1959.892
32	5000	3869893	276.348	904.655
16	5000	30664297	987.394	253.192



下面是memory bandwidth 的部分，為了計算方便選了計算量最大的phase3作為標準。由於 block size 變大後每次要傳輸的資料量也變大，我們可以看到跟GOPS 相似的結果，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。

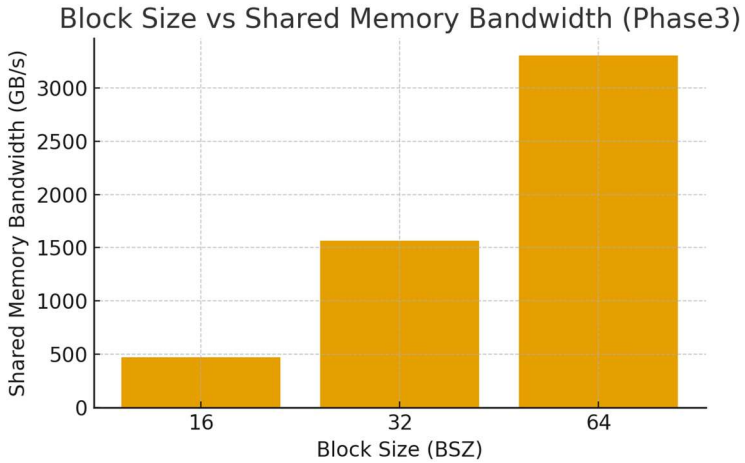
global:

BSZ	Global Load (GB/s)	Global store (GB/s)	Global BW (GB/s)
16	88.131	58.754	146.885
32	166.470	55.489	221.959
64	210.730	70.243	280.973



shared:

BSZ	Shared Load (GB/s)	Shared Store (GB/s)	Shared BW (GB/s)
16	355.12	118.37	473.49
32	1343.50	223.91	1567.41
64	3169.00	132.04	3301.04



c. Optimization (hw3-2)

以下比較cpu+openmp在apollo-origo的結果與gpu方法在apollo-gpu上的結果。

我先跑了cpu+openmp版本作為 baseline。這個版本大概要跑 15.55 秒。

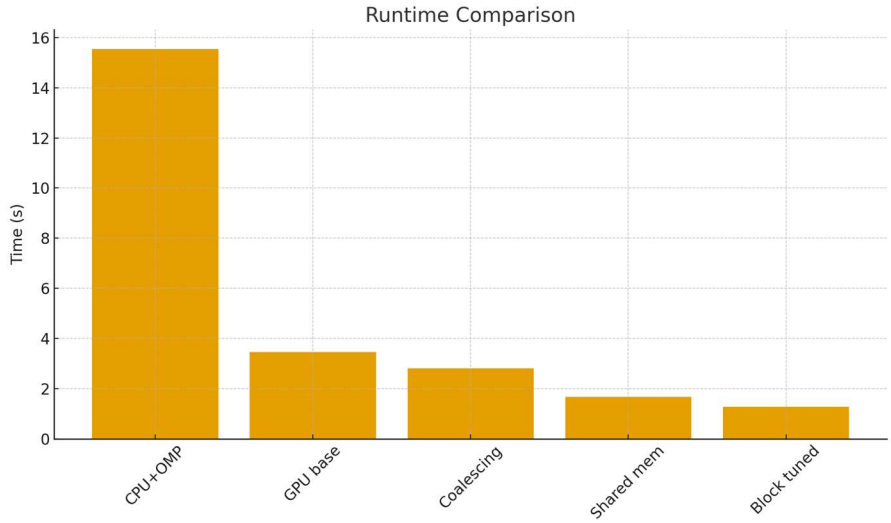
接著把 Floyd – Warshall 移到 GPU 上不做任何優化需要 3.461 秒。

之後我們調整 thread block 讓記憶體能 coalesced，只靠改善 memory access pattern，時間降到 2.813 秒。

再來加入 shared memory，把 pivot row/column 以及 tiles 暫存起來，減少 global memory 的來回讀取，變成 1.668 秒。

最後調整 block size、tile size，讓 SM 使用率更好，最終版本跑出 1.275 秒。

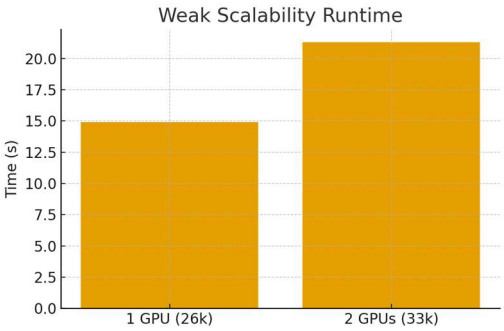
Version	Time (s)	Speedup vs CPU
CPU + OpenMP	15.55	1.00×
GPU baseline	3.461	4.49×
Coalescing access	2.813	5.53×
Shared mem + final	1.668	9.32×
Block size tune	1.275	12.20×

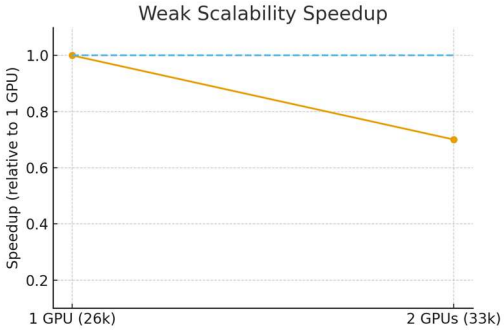


d. Weak scalability (hw3-3)

在weak scalability實驗中, 我使用p26k1, p33k1的測資去測試1-gpu, 2-gpu的weak scalabilty, 這是因為根據floyd-warshell的algo, 會需要大約2^(⅓)大小的node數差別，實驗結果如下: 可以看到有大約30%的損失。但整體效果仍相當好。

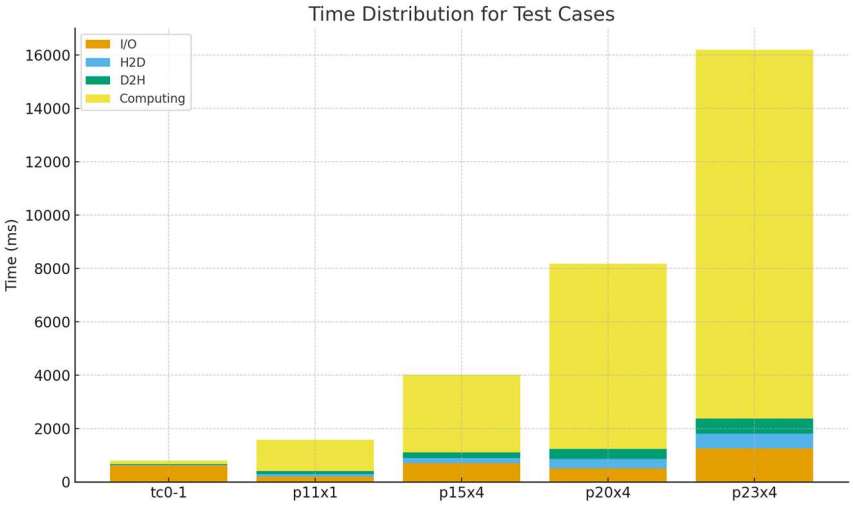
GPUs	Problem Size (V)	Time (s)	Speedup
1 GPU	26,000	14.912	1.00
2 GPUs	33,000	21.291	0.70





e. Time Distribution (hw3-2)

可以看出程式主要的瓶頸仍然是 computing，隨著測資變大，computing逐漸成為程式最多的部分 顯示gpu計算上面應該還有方法可以繼續壓縮才對；H2D / D2H 隨著測資變大有微幅增加，不過差異並不像 computing那樣強烈。



4. AMD GPU Porting and Analysis

a. implementation

我將原本的 CUDA 程式改成 HIP，以便在 AMD MI210 上執行。移植過程主要是把 CUDA 的 API 換成 ROCm 的 HIP API，其餘程式結構幾乎不變。

1.API 對應替換

cudaMalloc → hipMalloc

cudaFree → hipFree

cudaMemcpy → hipMemcpy

cudaMemcpyHostToDevice / cudaMemcpyDeviceToHost → HIP 對應版本

kernel 呼叫語法 <<< >>> 保持相同

2. 標頭檔替換

#include <cuda_runtime.h> → #include <hip/hip_runtime.h>

3. Kernel 內容維持不變

thread/block 索引、__shared__、__syncthreads() 都能直接在 hip 上使用

4. 編譯方式變更

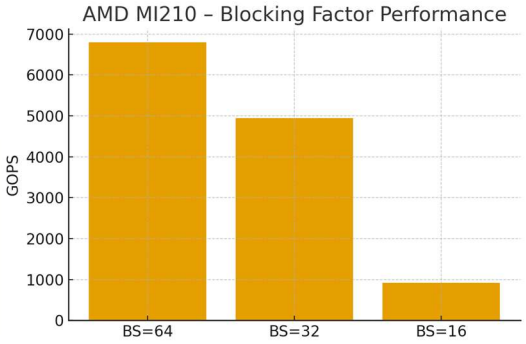
使用 hipcc --offload-arch=gfx90a

b. Experiment & Analyze

blocking-factor

同樣使用c21.1的測資做block實驗，如下所示，值得注意的是gops普遍遠大於cuda的情況，這個源自於MI210 是資料中心的 GPU,整數運算能力,HBM 記憶體頻寬,CU 數量都大於1080，所以 floyd-warshell 算起來會快很多。

BSZ	N	Total Kernel Cnts	Time (ms)	GOPS
64	5000	493,039	36.810	6791.567
32	5000	3,869,893	50.598	4940.915
16	5000	30,664,297	272.492	917.457



接著我也使用nvprof, rocmprof來觀察在不同block size下, phase3的平均執行時間(ms) 發現amd在執行phase3的時候平均時間不同於nv是在64是最佳, rocm反而是在32的時候有最佳phase3的平均執行時間(ms)。推測是因為MI210 的shared mem 結構、wavefront64 的執行模型，使得在phase3 中 64×64 block 反而造成更高的 shared mem 壓力與 occupancy 降低，所以最佳 block size 落在 32。

NV

Blocking Factor	Phase 3 平均時間 (ms)	Grid Size (launch blocks)
16	3.12	97,344
32	1.78	24,336
64	1.36	6,084

AMD

Blocking Factor	Phase 3 平均時間 (ms)	Work-groups
16	0.8488	6,230,016
32	0.2695	6,230,016
64	0.333	6,230,016

one optimization

在本次 amd gpu版本中，我嘗試加入 rocm 專屬的 kernel attribute:

```
__attribute__((amdgpu_flat_work_group_size(1024, 1024)))
```

此屬性用於明確告知 amdgpu 後端編譯器:此 kernel 的 work-group size 固定為 1024 threads 對應到程式的 block 為 32×32

根據AMD 官方文件,對於大型work-group，這個提示可以讓編譯器更精準配置 VGPR,LDS，以及排程 wavefront，理論上可以提升occupancy 或減少資源保守配置。

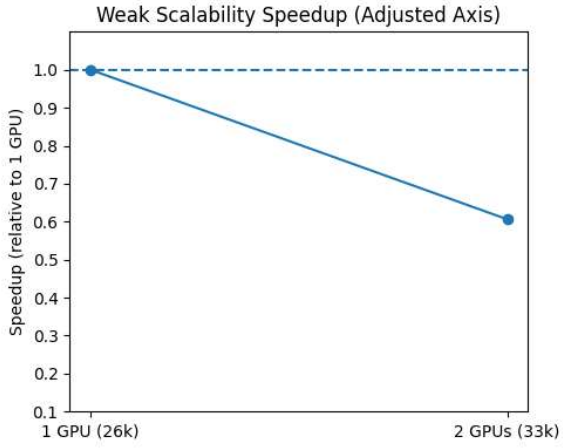
版本	Compute time (ms)
Baseline	35.328
加入 flat_work_group_size	35.370

但發現差異極小,可視為無明顯幫助。猜測MI210 compiler 已經能從實際 launch config 推斷 block size，所以加入額外指示無明顯幫助。

weak-scaling

在amd weak scalability實驗中, 我跟cuda一樣使用p26k1, p33k1的測資去測試1-gpu, 2-gpu的weak scalabilty，這裡紀錄的時間為kernel的computation time, 因為發現io time在amd的情況下是bottle neck，最後實驗結果如下: 可以看到有大約40%的損失比cuda的略慢。

GPUs	Problem Size (V)	Time (s)	Speedup
1 GPU	26,000	6.19	1.00
2 GPUs	33,000	10.21	0.61



5. Experience & conclusion

透過這次 CUDA/HIP 平行程式作業，我更加理解 GPU 加速的核心要素，包括 block size、記憶體取用模式與 shared memory 的影響。實驗中發現 block size 對效能有明顯作

用，而 coalesced access、shared memory都能穩定降低執行時間。在 multi-GPU 版本中，雖然運算可分散到多張 GPU 上，但資料交換與同步會成為新的開銷來源，需要特別處理。此外，在amd 實驗中也發現 大型 APSP 問題的 I/O 成本遠高於預期，對總執行時間影響明顯。整體而言，這次作業讓我更清楚 GPU 程式效能並非只由 kernel 計算決定，也必須兼顧資料布局、傳輸成本與硬體特性，才能真正達到最佳表現。