NTHU CS542200 Parallel Programming Homework 3: All-Pairs Shortest Path

112062591 李威辰

implementation

hw3-1

在作業3-1中我利用本堂課提供的sequential版的Blocked Floyd-Warshall algorithm利用OMP進行修改，在每一round中會有三個phase，然後每個phase會有要去計算對應block的shortest path，本次作業是利用#pragma omp parallel for schedule(dynamic)的方式去平行化找min path的計算。

```
void cal(
    int B, int Round, int block_start_x, int block_start_y, int block_width
    int block_end_x = block_start_x + block_height;
    int block_end_y = block_start_y + block_width;


    for (int k = Round * B; k < (Round + 1) * B && k < n; ++k){
        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
            for (int b_i = block_start_x; b_i < block_end_x; ++b_i)  {
                // To calculate original index of elements in the block (b_
                // For instance, original index of (0,0) in block (1,2) is
                int block_internal_start_x = b_i * B;
                int block_internal_end_x = (b_i + 1) * B;
                int block_internal_start_y = b_j * B;
                int block_internal_end_y = (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;

                #pragma omp parallel for schedule(dynamic) \\parallel
                for (int i = block_internal_start_x; i < block_internal_end
                    for (int j = block_internal_start_y; j < block_internal
                        if (Dist[i][k] + Dist[k][j] < Dist[i][j]) {
                            Dist[i][j] = Dist[i][k] + Dist[k][j];
                        }
                    }
                }
            }
        }
    }
}
```

hw3-2

本作業執行single GPU的CUDA平行化Blocked Floyd-Warshall algorithm，由於每一個round之間有data dependency，無法平行，所以這次主要平行每一個round中的3個phase函數的計算，以下說明以blocking factor = 2來計算以及每一個block都是32×32個threads。先將data分割成一群blocks，長寬都是tn=ceil(n/64)×64，然後開始進行計算，所以個block所拿到的資料量都是64×64。

```
void input(char *inFileName){
    FILE *file = fopen(inFileName, "rb");
    fread(&n, sizeof(int), 1, file);
    fread(&m, sizeof(int), 1, file);

    tn = ceil(n, 64) * 64;
    cudaMallocHost(&Dist, tn*tn*sizeof(int));
    for(int i = 0; i < tn; i++){
        for(int j = 0; j < tn; j++){
            Dist[i*tn+j] = (i==j&&i<n)?0:INF;
        }
    }

    int pair[3];
    for(int i = 0; i < m; i++){
        fread(pair, sizeof(int), 3, file);
        Dist[pair[0]*tn+pair[1]] = pair[2];
    }
    fclose(file);
}
```

下面開始就是此演算法的計算，先將host的data複製轉移到GPU上然後進行每一round的3個phase計算:

```
void block_FW(int B){
    cudaMalloc(&device_Dist, tn*tn*sizeof(int));
    cudaMemcpy(device_Dist, Dist, tn*tn*sizeof(int), cudaMemcpyHostToDevice
    int round = tn/64;
    dim3 num_thds(32, 32);
    dim3 num_blks_ph2(2, round-1);
    dim3 num_blks_ph3(round-1, round-1);
    for(int r = 0; r < round; r++){
        phase1 <<<1, num_thds>>> (B, r, device_Dist, tn);
        phase2 <<<num_blks_ph2, num_thds>>> (B, r, device_Dist, tn);
        phase3 <<<num_blks_ph3, num_thds>>> (B, r, device_Dist, tn);
    }
    cudaMemcpy(Dist, device_Dist, tn*tn*sizeof(int), cudaMemcpyDeviceToHost
    cudaFree(device_Dist);
}
```

phase 1:由於此phase主要在計算pivot block內的min path，所以只需用一個block的threads做計算，且每個threads去處理4個點的min path計算，分別是(blk_i+(i)),(blk_j+(j)),(blk_i+(i)),(blk_j+(j+32)),(blk_i+(i+32)),(blk_j+(j)),(blk_i+(i+32)),(blk_j+(j+32))，其中blk_j,blk_i是該block所對

應到的資料的block座標(phase1中為該round r的pivot block的座標)，i和j是thread的座標，也就是說每個threads各自處裡64×64 data中有4個32×32 data的正方形裡自己座標的資料(後面phase2，phase3也是相同的分割方式只是block資料的座標會不太一樣就不多做贅述)。
此處有用share memory進行優化，此一個block的資料是64×64，所以share memory的大小為64×64.

```
__global__ void phase1(int B, int r, int *device_Dist, int tn){
    __shared__ int s[64*64];
    int blk_i = r<<6, blk_j = r<<6;
    int i = threadIdx.y, j = threadIdx.x;

    #pragma unroll
    for(int x = 0; x < 2; x++){
        #pragma unroll
        for(int y = 0; y < 2; y++){
            s[(i+32 * y)*64+(j+32 * x)] = device_Dist[(blk_i+(i+32 * y))*tn
        }
    }
    __syncthreads();

    #pragma unroll 48
    for(int k = 0; k < 64; k++){
        #pragma unroll
        for(int x = 0; x < 2; x++){
            #pragma unroll
            for(int y = 0; y < 2; y++){
                s[(i+32 * y)*64+(j+32 * x)] = min(s[(i+32 * y)*64+(j+32 * x
            }
        }
        __syncthreads();
    }

    #pragma unroll
    for(int x = 0; x < 2; x++){
        #pragma unroll
        for(int y = 0; y < 2; y++){
            device_Dist[(blk_i+(i+32 * y))*tn+(blk_j+(j+32 * x))] = s[(i+32
        }
    }
}
```

phase 2: 要執行與該round 的pivot block同行同列的block min path計算，此處需要2×(round-1)個block去做平行計算，分資料的方式是blockId.x=1的是處裡pivot row，blockId.x=0是處裡pivot cloumn，然後當blockId.y如果大於pivot block的y座標，pivot row裡的blk_j=blockId.y就要加一(blk_i即為pivot block 的y座標)，pivot column裡的blk_i=blockId.y就要加一(blk_j即為pivot block 的x座標)，這樣就分完每個block所需的資料了。
此處有用share memory進行優化，此一個block的資料是64×64，此處因計算需要，所以share memory的大小為2×64×64.

```
__global__ void phase2(int B, int r, int *device_Dist, int tn){
    __shared__ int s[2*64*64];
    int blk_i = (blockIdx.x*r+(!blockIdx.x)*(blockIdx.y+(blockIdx.y>=r)))<<
    int blk_j = (blockIdx.x*(blockIdx.y+(blockIdx.y>=r))+(!blockIdx.x)*r)<<
    int blk_p = r<<6;
    int i = threadIdx.y, j = threadIdx.x;

    int val0 = device_Dist[(blk_i+i)*tn+(blk_j+j)];
    int val1 = device_Dist[(blk_i+i)*tn+(blk_j+(j+32))];
    int val2 = device_Dist[(blk_i+(i+32))*tn+(blk_j+j)];
    int val3 = device_Dist[(blk_i+(i+32))*tn+(blk_j+(j+32))];


    #pragma unroll
    for(int x = 0; x < 2; x++){
        #pragma unroll
        for(int y = 0; y < 2; y++){
            s[(i+32*y)*64+(j+32*x)] = device_Dist[(blk_i+(i+32*y))*tn+(blk_
            s[4096+(i+32*y)*64+(j+32*x)] = device_Dist[(blk_p+(i+32*y))*tn+
        }
    }
    __syncthreads();
    #pragma unroll 48
    for(int k = 0; k < 64; k++){
        val0 = min(val0, s[i*64+k]+s[4096+k*64+j]);
        val1 = min(val1, s[i*64+k]+s[4096+k*64+(j+32)]);
        val2 = min(val2, s[(i+32)*64+k]+s[4096+k*64+j]);
        val3 = min(val3, s[(i+32)*64+k]+s[4096+k*64+(j+32)]);
    }

    device_Dist[(blk_i+i)*tn+(blk_j+j)] = val0;
    device_Dist[(blk_i+i)*tn+(blk_j+(j+32))] = val1;
    device_Dist[(blk_i+(i+32))*tn+(blk_j+j)] = val2;
    device_Dist[(blk_i+(i+32))*tn+(blk_j+(j+32))] = val3;
}
```

phase 3:要執行**除了**與該round 的pivot block同行同列的block min path計算，此處需要(round-1)×(round-1)個block去做平行計算，分資料的方式是若blockId.x>pivot block的x座標，blk_j=blockId.x要加一，若blockId.y>pivot block的y座標，blk_i=blockId.y要加一，這樣就分完每個block所需的資料了。。
此處有用share memory進行優化，此一個block的資料是64×64，此處因計算需要，所以share memory的大小為2×64×64.

```
__global__ void phase3(int B, int r, int *device_Dist, int tn){
    __shared__ int s[2*64*64];
    int blk_i = (blockIdx.x+(blockIdx.x>=r))<<6, blk_j = (blockIdx.y+(block
    int i = threadIdx.y, j = threadIdx.x;

    int val0 = device_Dist[(blk_i+i)*tn+(blk_j+j)];
    int val1 = device_Dist[(blk_i+i)*tn+(blk_j+(j+32))];
    int val2 = device_Dist[(blk_i+(i+32))*tn+(blk_j+j)];
    int val3 = device_Dist[(blk_i+(i+32))*tn+(blk_j+(j+32))];

    #pragma unroll
    for(int x = 0; x < 2; x++){
        #pragma unroll
        for(int y = 0; y < 2; y++){
            s[(i+32*y)*64+(j+32*x)] = device_Dist[(blk_i+(i+32*y))*tn+(blk_
            s[4096+(i+32*y)*64+(j+32*x)] = device_Dist[(blk_p+(i+32*y))*tn+
        }
    }

    __syncthreads();
    #pragma unroll 48
    for(int k = 0; k < 64; k++){
        val0 = min(val0, s[i*64+k]+s[4096+k*64+j]);
        val1 = min(val1, s[i*64+k]+s[4096+k*64+(j+32)]);
        val2 = min(val2, s[(i+32)*64+k]+s[4096+k*64+j]);
        val3 = min(val3, s[(i+32)*64+k]+s[4096+k*64+(j+32)]);
    }

    device_Dist[(blk_i+i)*tn+(blk_j+j)] = val0;
    device_Dist[(blk_i+i)*tn+(blk_j+(j+32))] = val1;
    device_Dist[(blk_i+(i+32))*tn+(blk_j+j)] = val2;
    device_Dist[(blk_i+(i+32))*tn+(blk_j+(j+32))] = val3;
}
```

最後就將device_Dist的資料用cudaMemcpy 傳回host再進行output。

hw3-3

原則上input和output都一樣這裡就不贅述，這裡將GPU的id分為0和1，id=0會被分到前面size=(round/2)×64×tn個資料，id=1會被分到剩餘資料，然後進行每一round迴圈，進行三階段的運算，裡面的內容全部都和hw3-2一樣這裡就不贅述(包含block number, thread number等)，只有phase 3的block size 從(round-1, round-1)變成(size, round-1)此計算兩個GPU沒有data dependency且各自裡面的資料在phase3都要計算所以要同時各自算自己的部分，然後跑完一次迴圈完成phase123就用cudaMemcpyPeer去傳遞各自運算完的結果給彼此。
這裡要注意的事情是當前round/2的計算結果只有id=0的GPU是對的因為他拿到前半部分資料，反之後round/2的計算結果id=1亦然，舉個很直覺的例子在第一輪中算phase1，要算block(0, 0)的min path需用GPU id =0去計算而GPU id =1的會去計算block(round/2, round/2)的結果但根本不對因為每round間都跟前一個round有data dependency，所以是錯的，這裡為了解決這個問題，在cudaMemcpyPeer前面加上if(r<round/2)和else就可以確保前半0給1後半1給0。

```
void block_FW(int B){
    #pragma omp parallel num_threads(2)
    {
        int thd_id = omp_get_thread_num(), round = tn/64;
        int offset = (round/2)*thd_id, size = round/2;
        if(thd_id&&round%2)size++;
        cudaSetDevice(thd_id);
        cudaMalloc(&device_Dist[thd_id], tn*tn*sizeof(int));
        #pragma omp barrier
        cudaMemcpy(device_Dist[thd_id]+(offset*64*tn), Dist+(offset*64*tn),
        for(int r = 0; r < round; r++){
            if(r<round/2)cudaMemcpyPeer(device_Dist[1]+(r*64*tn), 1, device
            else cudaMemcpyPeer(device_Dist[0]+(r*64*tn), 0, device_Dist[1]
            #pragma omp barrier
            dim3 num_thds(32, 32);
            dim3 num_blks_ph2(2, round-1);
            dim3 num_blks_ph3(size, round-1);
            phase1 <<<1, num_thds>>> (B, r, device_Dist[thd_id], tn);
            phase2 <<<num_blks_ph2, num_thds>>> (B, r, device_Dist[thd_id],
            phase3 <<<num_blks_ph3, num_thds>>> (B, r, device_Dist[thd_id],
        }
        cudaMemcpy(Dist+(offset*64*tn), device_Dist[thd_id]+(offset*64*tn),
        cudaFree(device_Dist[thd_id]);
    }
}
```

implementation

以下都在本堂課提供之Apollo GPU平台做測試

profiling result

testcase c10.1

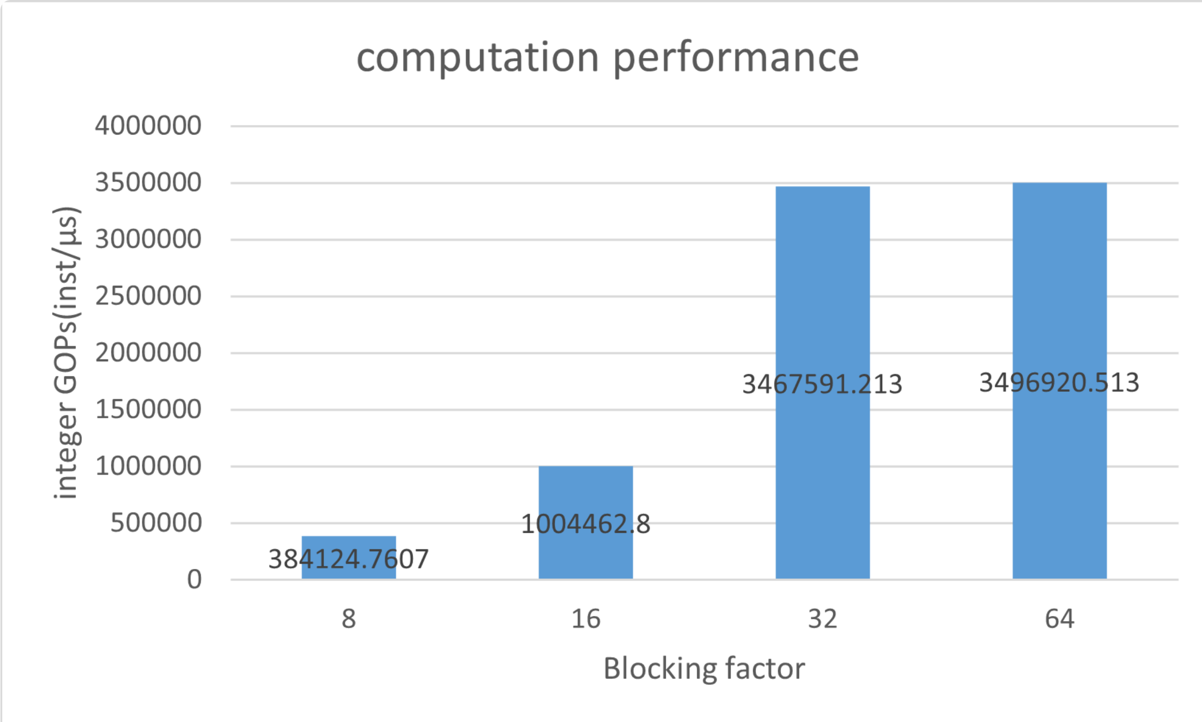
blocking factor 64

64	kernel 1	kernel 2	kernel 3
Achieved Occupancy	0.497965	0.486942	0.824094
Multiprocessor Activity	0.045	0.444	0.6806
share_load(GB/sec)	68.445	356.52	428.51
share_store(GB/sec)	23.052	7.4274	8.9273
global_load(GB/sec)	0.36315	245.1	294.6
global_store(GB/sec)	0.36315	237.68	285.67

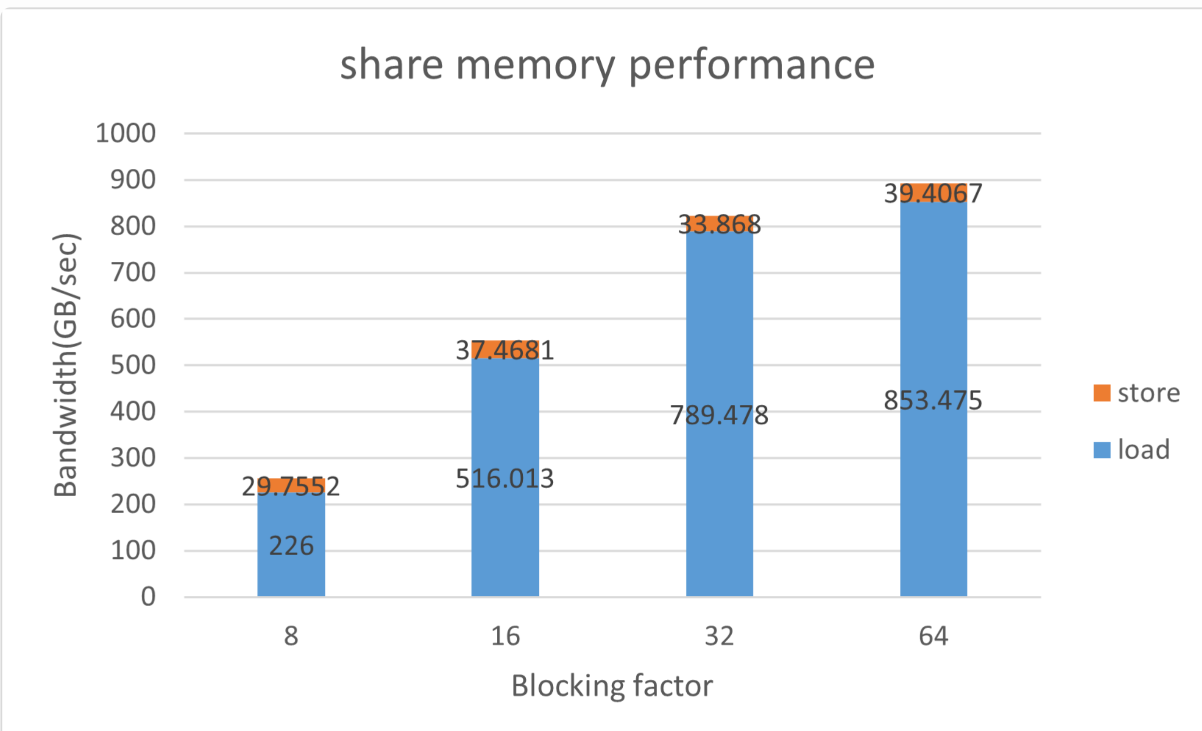
這是用nvprof做的結果，其中Achieved Occupancy(GPU上執行核心的使用率)和Multiprocessor Activity(GPU活動時間與總時間的比例)是用來看GPU的利用資源比例，所以期望越接近1越好，可以看到在kernel 1(phase1)和2(phase2)的函數利用率只有不到50%，而kernel3(phase3)好很多，原因就是phase1、phase2函數所需的計算threads量不多導致的。
而由bandwidth的數據發現此程式share memory中load data的次數較多，而global都差不多。

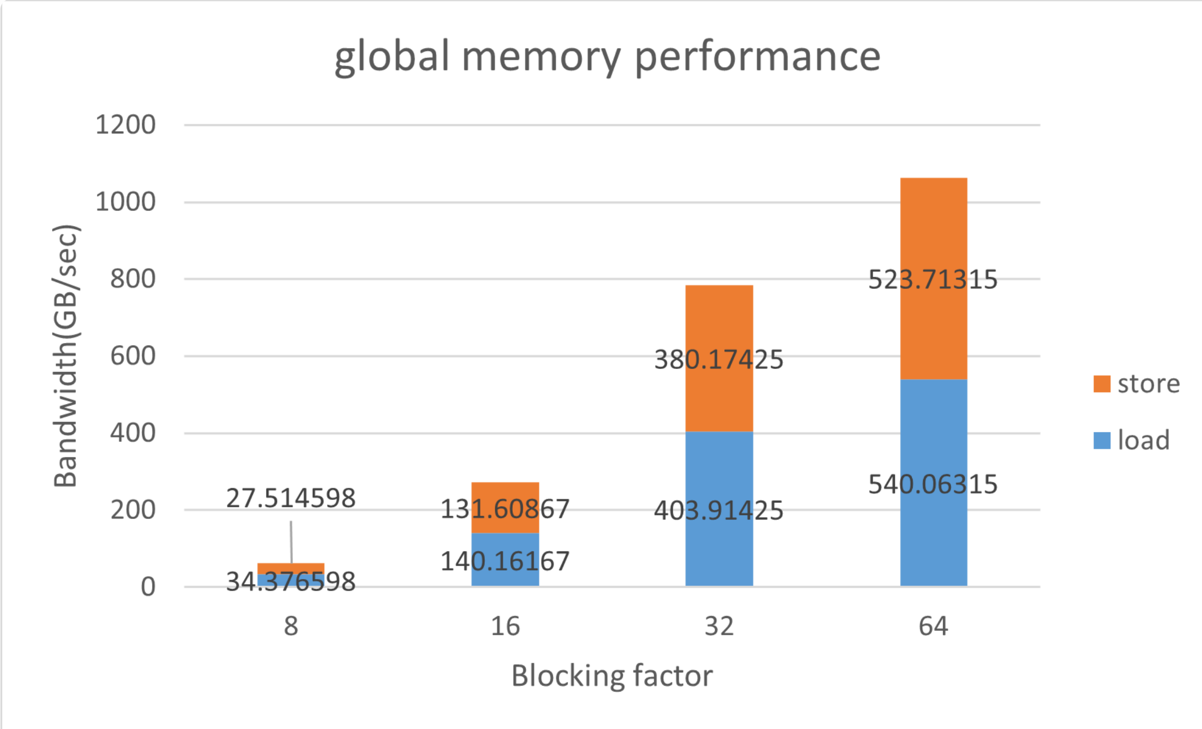
Blocking Factor (hw3-2)

這是用nvprof做的結果，以下利用testcase c10.1進行測試在不同blocking factor下 Integer GOPS 和 global/shared memory bandwidth有什麼不同:



可以看到在blcking factor越大每單位時間的整數運算就越多，符合預期，但32和64之間的成長不明顯，有可能是程式碼邏輯的問題造成其bandwidth差不多，像是32時每個threads的運算已經飽和就不會再增加。

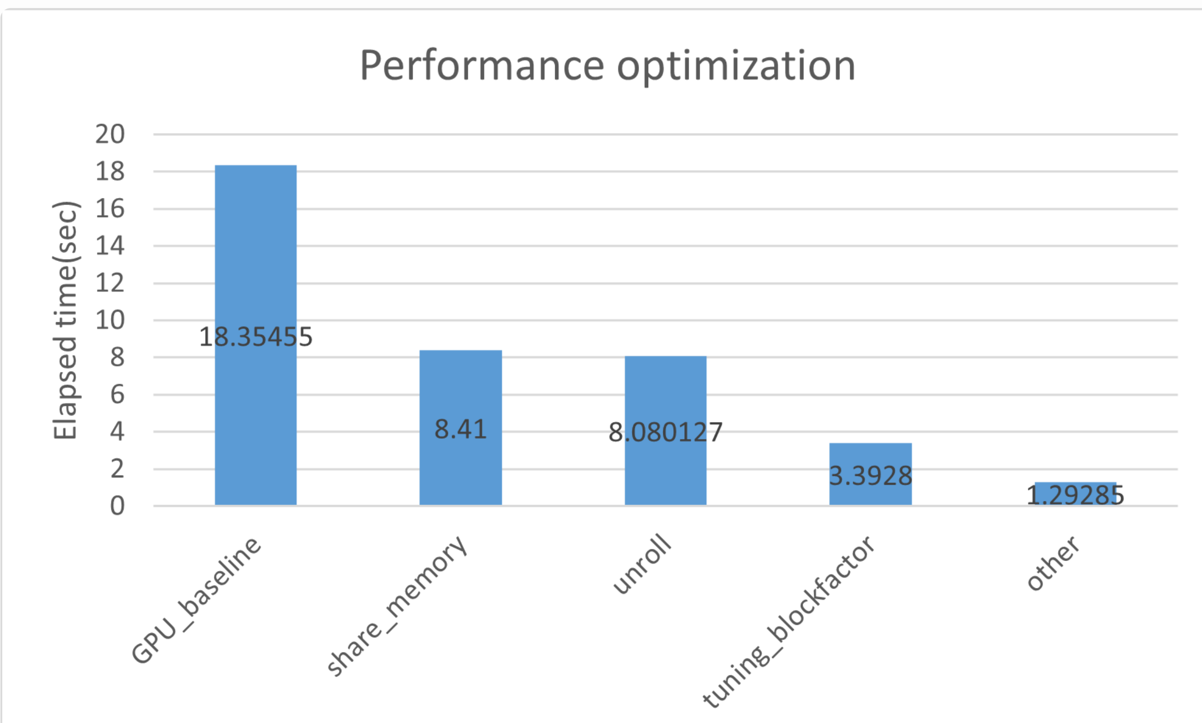




可以看到在blcking factor越大每單位時間的share memory 和global資料傳輸就越多，符合預期，只有share memory在32和64之間的成長不明顯。

Performance optimization (hw3-2)

這是用nvprof做的結果，下圖是每一部優化的結果，使用testcase p11k1:



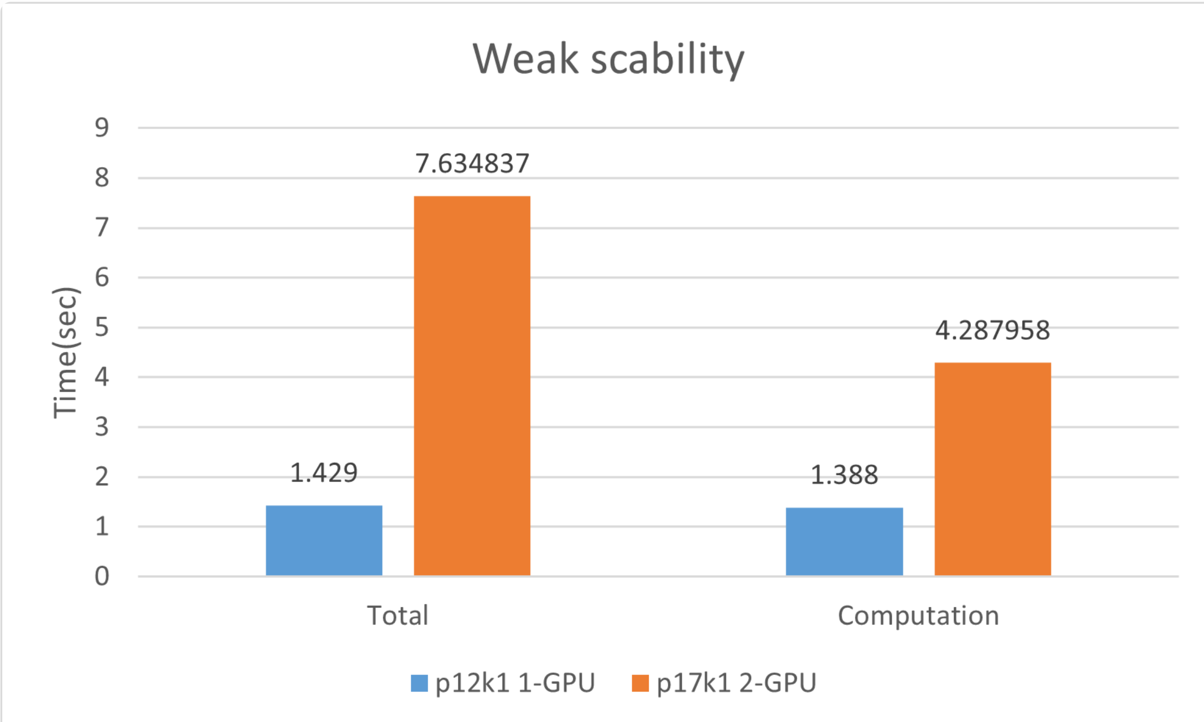
other: 調整一些內容，例如將計算用到的變數存取直接改為數字存取、將乘64改為向左移<<<6、將unroll的大小調整64變為48。

tunning_block_factor:調整blocking factor發現64為最快。

可以看到期從原本的18.4降至1.3將近14倍的加速結果還不錯。

Weak scalability (hw3-3)

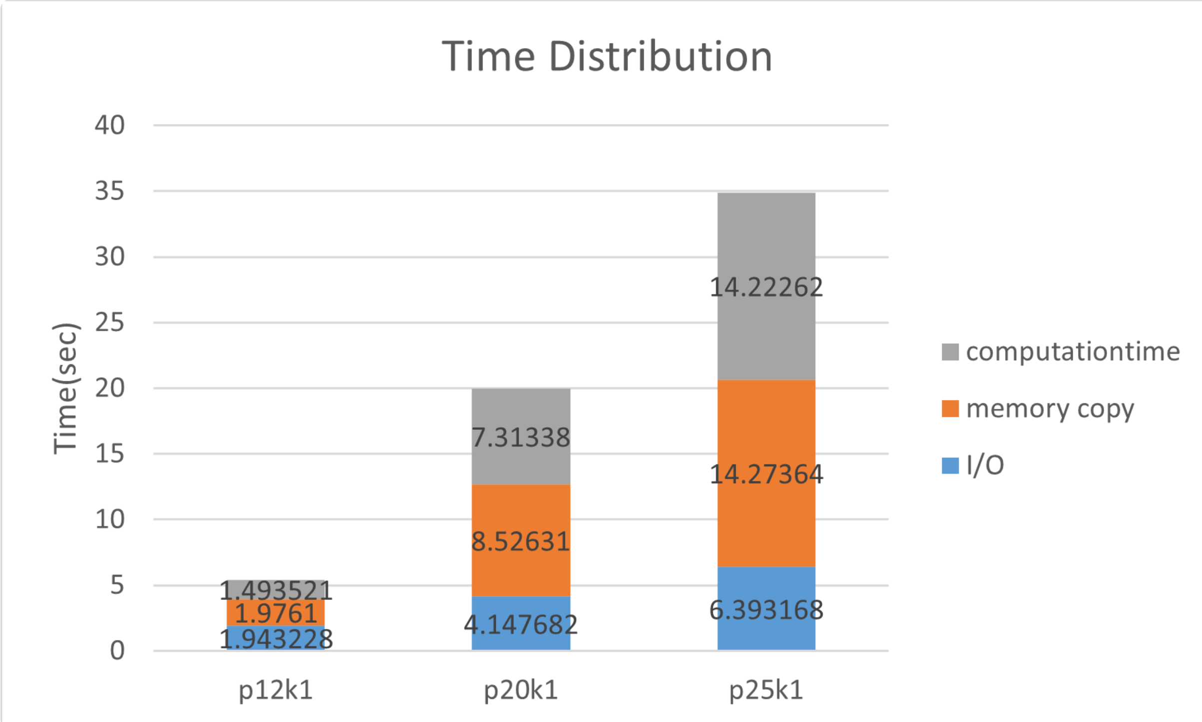
這裡用#include <sys/time.h>中的gettimeofday來進行測量，這裡探討2-GPU和1-GPU的程式效能比較，看performance有沒有一樣，這裡選用testcase p12k1、 p17k1來測試(因為vertex平方資料數量17比12大約兩倍):



可以發現計算時間(掛在三個phase的round迴圈外)和總時間2-GPU都比較多，可能因為中間運算兩個GPU需要進行資料傳輸cudaMemcpyPeer導致整體計算速度嚴重下滑而使總時間增加。

Time Distribution (hw3-2)

這裡用#include <sys/time.h>中的gettimeofday來進行測量，使用testcase p12k1、p20k1、p25k1進行測式:

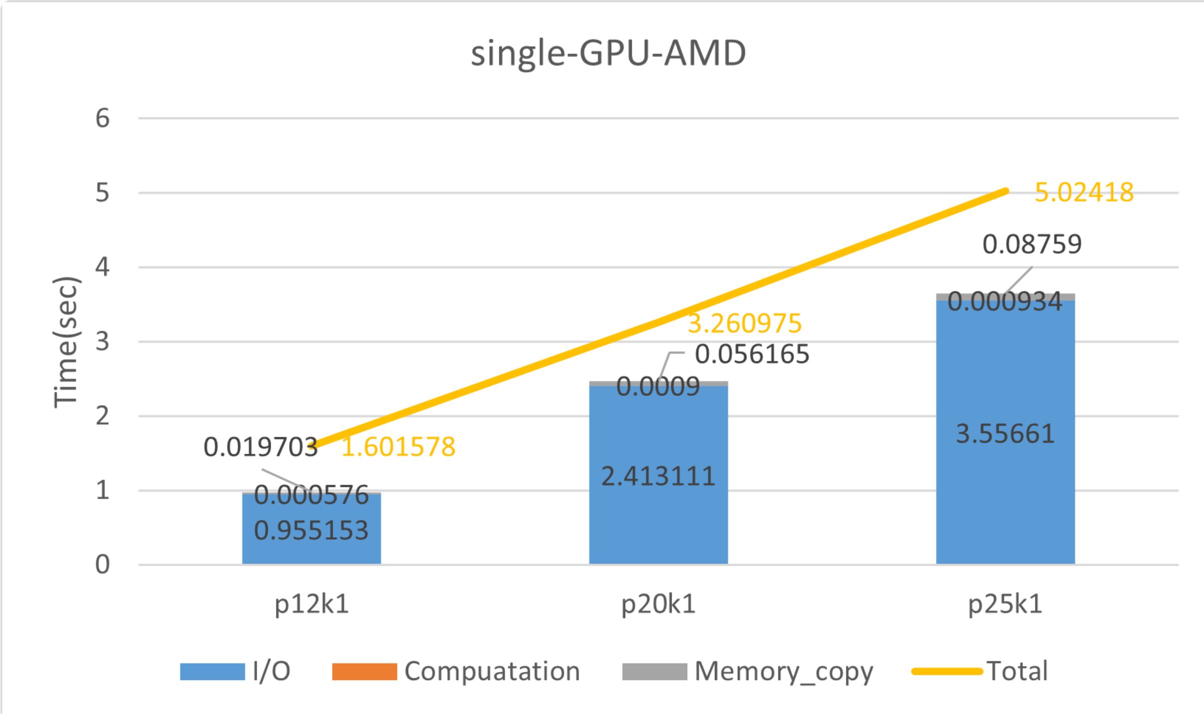


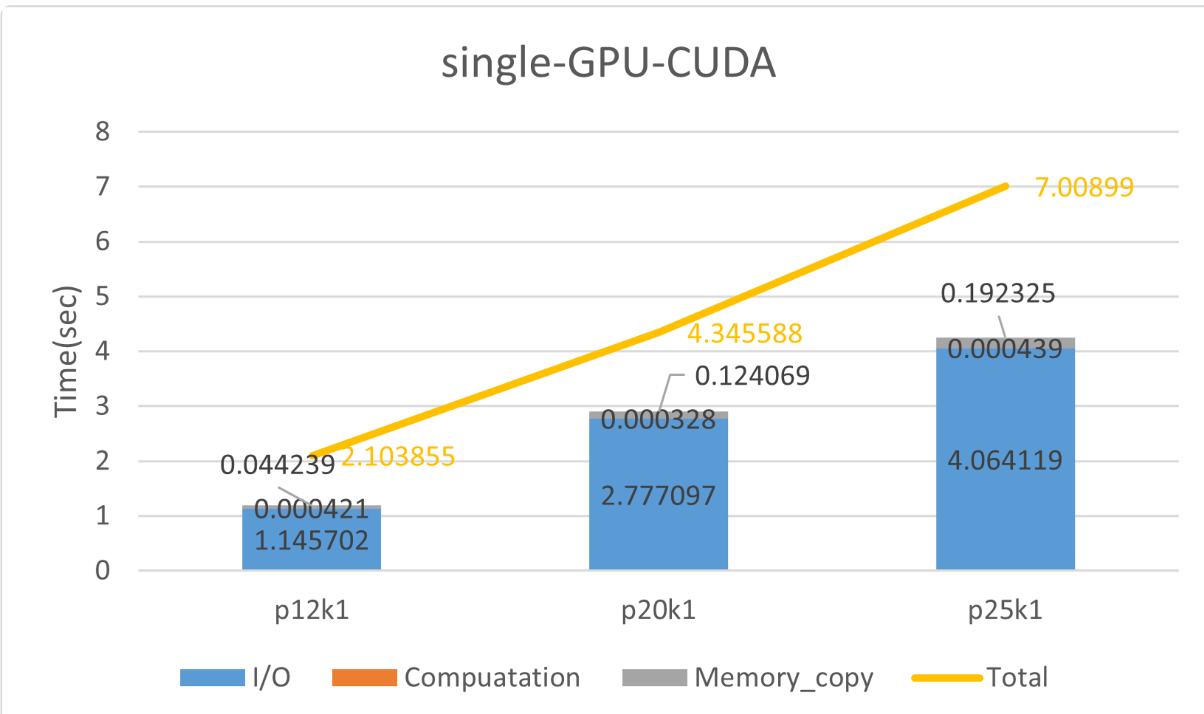
由圖可以知道I/O、memory copy、computation time都隨著testcase的資料變大而變大，其中memory copy、computation time最為明顯，符合預期的可能性，memory copy的時間原本就是平行程式的一個效能瓶頸，或許可以進一步優化記憶體存取之類的地方用padding或memory coalescing的方式存取global data。

Experiment on AMD GPU

這裡比較AMD GPU 的程式和CUDA的差異性，這裡用#include <sys/time.h>中的gettimeofday來進行測量I/O、memory copy、computation time、total time，使用testcase p12k1、p20k1、p25k1進行測式:

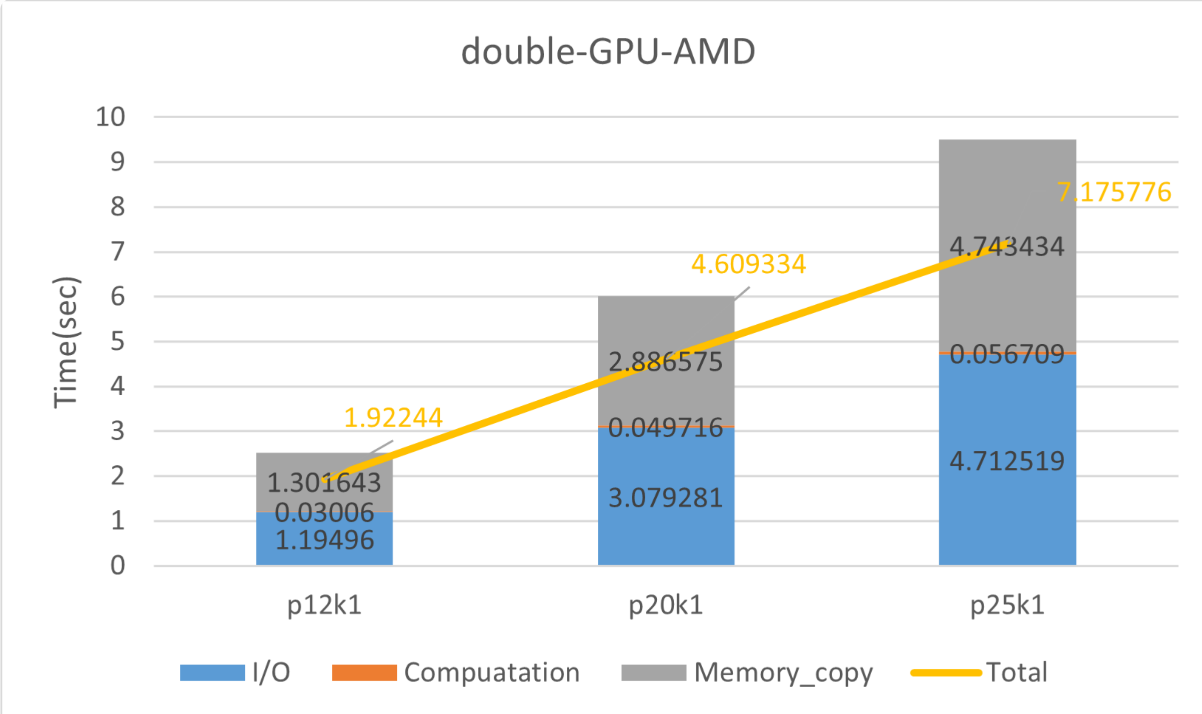
CUDA V.S. AMD(hw3-2 single GPU)

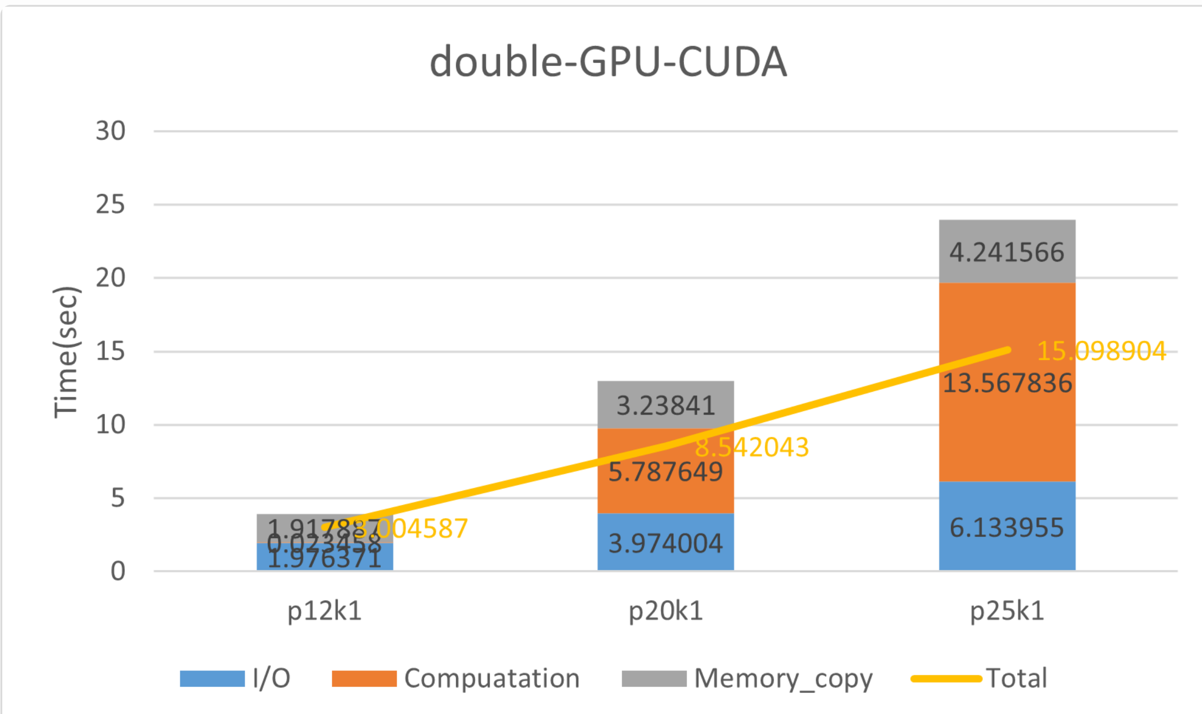




由圖可以發現，雙方的時間佔比最大的是I/O>>>memory copy>computation time，而且可以發現AMD的I/O以及memory copy的時間都比CUDA小，所以造成總體運行時間比較好。

CUDA V.S. AMD(hw3-3 double GPU)





CUDA的時間佔比最大的是I/O>computation time>memory copy，並且這裡的總時間比三個加總來的小是因為測量時memory copy有一部分包含在computation time重疊到了，所以使computation time很大，所以應該還是I/O>memory copy>computation time。由圖可以發現AMD的總運行時間比較少，因為AMD的I/O以及memory copy的時間都比CUDA小，，所以造成總體運行時間比較好。

Experience & conclusion

這次作業發現更改一些簡單的operation會導致程式的效能改變，並且GPU-Host 間memcopy的時間佔據會讓程式變得很慢，所以還有很多可以優化的地方。

並且這個作業也讓我認識AMD這個東西，發現他做I/O、memcopy等operation會快不少，使原本的平行程式效能顯著增加。

這次作業讓我學習到如何用CUDA將程式平行化，裡面有非常多的細節要注意，不然會導致performance不佳甚至答案是錯的，最讓我頭疼的是優化的部分，以前存來沒想過一個資料存取或一個branch的if條件會使運算速度顯下降，甚至是share memory和global data的存取速度也有影響，這次真的讓我學到了很多有關程式效能東西。