All-Pairs Shortest Path

Student ID 110590032
Name 林展毅

Implementation

Which algorithm do you choose in hw3-1?

Blocked Floyd-Warshall algorithm

How do you divide your data in hw3-2, hw3-3?

3-2

```
void block_FW(){
    int round = (padding_n + B - 1) / B;
    dim3 block(n_thread, n_thread), grid2(2, round - 1), grid3(round - 1, round
    cudaMalloc(&dev_Dist, Dist_size);
    cudaMemcpy(dev_Dist, Dist, Dist_size, cudaMemcpyHostToDevice);

    for(int r = 0; r < round; r++){
        cal1 <<<1, block>>> (dev_Dist, r, padding_n);
        cal2 <<<grid2, block>>> (dev_Dist, r, padding_n);
        cal3 <<<grid3, block>>> (dev_Dist, r, padding_n);
    }

    cudaMemcpy(Dist, dev_Dist, Dist_size, cudaMemcpyDeviceToHost);
    //cudaFree(dev_Dist);
};
```

用一個blocking factor(B)決定每round要處理多少節點，一個block有 `32*32=1024` 個thread，為了把shared memory(48KB)盡量用完，每個thread可以最多處理四個(integer)節點。

根據演算法每個階段要做的事情不同，第一階段要計算pivot，因此只需要計算一個block，第二階段要計算扣除pivot block以外的picot row和pivot column，第三階段則是把剩下的block計算完，實作可見上方的程式碼的kernel function。

3-3

```
void block_FW(){
    #pragma omp parallel num_threads(2)
    {
        int round = (padding_n + B - 1) / B;
        int id = omp_get_thread_num();
        int st = round / 2 * id;
        int offset = st * B * padding_n;
        int size = (round / 2) + (round % 2) * id;
        size_t cur_size = size * B * padding_n * sizeof(int);
        dim3 block(n_thread, n_thread), grid2(2, round - 1), grid3(size, round -

        cudaSetDevice(id);
        cudaMalloc(&dev_Dist[id], Dist_size);

        #pragma omp barrier
        cudaMemcpy(dev_Dist[id] + offset, Dist + offset, cur_size, cudaMemcpyHos

        for(int r = 0; r < round; r++){
            if((r >= st) && (r < st + size)){
                cudaMemcpyPeer(dev_Dist[!id] + (r * B * padding_n), !id, dev_Dis
            }

            #pragma omp barrier
            cal1 <<<1, block>>> (dev_Dist[id], r, padding_n);
            cal2 <<<grid2, block>>> (dev_Dist[id], r, padding_n);
            cal3 <<<grid3, block>>> (dev_Dist[id], r, padding_n, st);
        }

        cudaMemcpy(Dist + offset, dev_Dist[id] + offset, cur_size, cudaMemcpyDev
        //cudaFree(dev_Dist);
    }
};
```

我選擇用openmp實作兩張GPU的版本，很簡單地把資料分成上下兩部分並分配給兩張GPU同時計算，比較需要注意的部分是計算一輪後要互相傳輸資料。

其餘部分和3-2相同。

What’s your configuration in hw3-2, hw3-3? And why? (e.g. blocking factor, #blocks, #threads)

blocking factor：64
blocks：根據不同階段數量不同，上一題有解釋。
threads：1024

我的想法是，盡量讓GPU block對應到我自己設定的block，所以thread和GPU原本的配置相同，blocking factor和block上一題有解釋

另外，3-3只有第三階段用到的block數和3-2不同，因為資料被平分了。

How do you implement the communication in hw3-3?

`cudaMemcpyPeer` ，直接讓兩張卡傳輸，比在Host上做處理還快

Briefly describe your implementations in diagrams, figures or sentences.

3-1

```
void cal(int Round, int block_start_x, int block_start_y, int block_width, int b
    int block_end_x = block_start_x + block_height;
    int block_end_y = block_start_y + block_width;
    int k_start = Round * B, k_limit = min(k_start + B, n);
    __m128i ik, kj, ij;

    #pragma omp for collapse(2) schedule(auto)
    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++){
            int block_internal_start_x = b_i * B;
            int block_internal_end_x = min(block_internal_start_x + B, n);
            int block_internal_start_y = b_j * B;
            int block_internal_end_y = min(block_internal_start_y + B, n);

            for(int k = k_start; k < k_limit; k++){
                for(int i = block_internal_start_x; i < block_internal_end_x; i+
                    ik = _mm_set1_epi32(Dist[i][k]);
                    for(int j = block_internal_start_y; j < block_internal_end_y
                        ij = _mm_loadu_si128((__m128i*)&Dist[i][j]);
                        kj = _mm_loadu_si128((__m128i*)&Dist[k][j]);

                        _mm_storeu_si128((__m128i*)&Dist[i][j], _mm_min_epi32(_mm
                    }
                }
            }
        }
    }
}
```

用openmp實作，並使用sse指令集，blocking factor設置為64。並collapse展開迴圈增加平行度。

3-2

我將三個階段分別實作成三個kernel function，一來比較直覺，二來測試時間比較方便。

首先要做padding讓資料對齊blocking factor的倍數。這裡比較複雜的是資料輸出時要用原本的n進行輸出。

在kernel function中，一次處理四筆資料會大大加快平行度，並且要使用unroll展開迴圈讓thread平行處理，最大的問題是要讓threads同步化，要盡量減少同步化的次數，以下以計算量最大的第三階段程式碼作範例說明。

(程式碼有點多，在下一頁)

```
__global__ void cal3(int *D, int r, int n){
    __shared__ int sm[n_sm_size];
    __shared__ int cp[n_sm_size];

    int b_i, b_j, b_k;
    int i, j;
    volatile int ik, kj;
    int tmp[4];

    // 剩下一大塊
    b_i = (blockIdx.x + (blockIdx.x >= r)) << 6;
    b_j = (blockIdx.y + (blockIdx.y >= r)) << 6;
    b_k = r << 6;
    i = threadIdx.y, j = threadIdx.x;
    ik = kj = (i * n_thread + j) * n_per_iter;

    tmp[0] = D[(b_i + i) * n + b_j + j];
    tmp[1] = D[(b_i + i) * n + b_j + j + n_thread];
    tmp[2] = D[(b_i + i + n_thread) * n + b_j + j];
    tmp[3] = D[(b_i + i + n_thread) * n + b_j + j + n_thread];

    sm[ik] = D[(b_i + i) * n + b_k + j];
    sm[ik + 1] = D[(b_i + i) * n + b_k + j + n_thread];
    sm[ik + 2] = D[(b_i + i + n_thread) * n + b_k + j];
    sm[ik + 3] = D[(b_i + i + n_thread) * n + b_k + j + n_thread];

    cp[kj] = D[(b_k + i) * n + b_j + j];
    cp[kj + 1] = D[(b_k + i) * n + b_j + j + n_thread];
    cp[kj + 2] = D[(b_k + i + n_thread) * n + b_j + j];
    cp[kj + 3] = D[(b_k + i + n_thread) * n + b_j + j + n_thread];

    __syncthreads();

    #pragma unroll n_thread
    for(int k = 0; k < n_thread; k++){
        ik = (i * n_thread + k) * n_per_iter;
        kj = (k * n_thread + j) * n_per_iter;
        tmp[0] = min(min(tmp[0], sm[ik] + cp[kj]), sm[ik + 1] + cp[kj + 2]);
        tmp[1] = min(min(tmp[1], sm[ik] + cp[kj + 1]), sm[ik + 1] + cp[kj + 3]);
        tmp[2] = min(min(tmp[2], sm[ik + 2] + cp[kj]), sm[ik + 3] + cp[kj + 2]);
        tmp[3] = min(min(tmp[3], sm[ik + 2] + cp[kj + 1]), sm[ik + 3] + cp[kj +
    }

    D[(b_i + i) * n + b_j + j] = tmp[0];
    D[(b_i + i) * n + b_j + j + n_thread] = tmp[1];
    D[(b_i + i + n_thread) * n + b_j + j] = tmp[2];
    D[(b_i + i + n_thread) * n + b_j + j + n_thread] = tmp[3];
};_i + i + n_thread) * n + b_j + j + n_thread] = tmp[3];
};
```

在sequential版本的程式碼中，要在迴圈中比較 `ik + kj < ij` 多次，我一次拿取四個節點，分別是i, j和i+32, j和i, j+32以及i+32, j+32，並為了加速我將ik和kj分別存入兩個陣列，並重新將四個節點排序放在一起，減少之後計算的memory access time。

為了減少同步化時間，我開啟一個tmp陣列放入最後要存取的值，這樣就不用每次k迴圈就同步化一次。

最後，我在k迴圈中將原本應該進行64次的迴圈展開成32次，畢竟資料沒有依賴性。

3-3

前面都有提到，只是把資料用openmp分配給兩張GPU。

Profiling

```
nvcc -std=c++11 -O3 -Xptxas="-v" -arch=sm_61 -lineinfo -G -lm -o hw3-2 hw3-2.cu

nvprof --metrics \
achieved_occupancy,\
sm_efficiency,\
shared_load_throughput,shared_store_throughput,\
gld_throughput,gst_throughput \
./hw3-2 testcase out
```

testcase為c20.2
env：hades02 with single GPU

結果：

```
==28241== Profiling application: ./hw3-2 testcase out
==28241== Profiling result:
==28241== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: cal1(int*, int, int)
         63                        achieved_occupancy                        Achieved Occupancy    0.493070    0.496799    0.493461
         63                             sm_efficiency                   Multiprocessor Activity       4.97%       4.98%       4.98%
         63                    shared_load_throughput             Shared Memory Load Throughput  17.002GB/s  19.500GB/s  19.323GB/s
         63                   shared_store_throughput            Shared Memory Store Throughput  7.6218GB/s  8.7414GB/s  8.6619GB/s
         63                            gld_throughput                    Global Load Throughput  30.018MB/s  34.428MB/s  34.115MB/s
         63                            gst_throughput                   Global Store Throughput  30.018MB/s  34.428MB/s  34.115MB/s
    Kernel: cal2(int*, int, int)
         63                        achieved_occupancy                        Achieved Occupancy    0.880611    0.884196    0.882182
         63                             sm_efficiency                   Multiprocessor Activity      89.05%      89.56%      89.35%
         63                    shared_load_throughput             Shared Memory Load Throughput  278.30GB/s  318.78GB/s  316.49GB/s
         63                   shared_store_throughput            Shared Memory Store Throughput  6.9576GB/s  7.9695GB/s  7.9122GB/s
         63                            gld_throughput                    Global Load Throughput  2.6091GB/s  2.9886GB/s  2.9671GB/s
         63                            gst_throughput                   Global Store Throughput  890.58MB/s  0.9962GB/s  0.9890GB/s
    Kernel: cal3(int*, int, int)
         63                        achieved_occupancy                        Achieved Occupancy    0.911158    0.912503    0.911801
         63                             sm_efficiency                   Multiprocessor Activity      99.54%      99.79%      99.66%
         63                    shared_load_throughput             Shared Memory Load Throughput  350.14GB/s  360.83GB/s  358.95GB/s
         63                   shared_store_throughput            Shared Memory Store Throughput  8.7535GB/s  9.0209GB/s  8.9739GB/s
         63                            gld_throughput                    Global Load Throughput  3.2826GB/s  3.3828GB/s  3.3652GB/s
         63                            gst_throughput                   Global Store Throughput  1.0942GB/s  1.1276GB/s  1.1217GB/s
```

Experiment & Analysis

System Spec

hades02

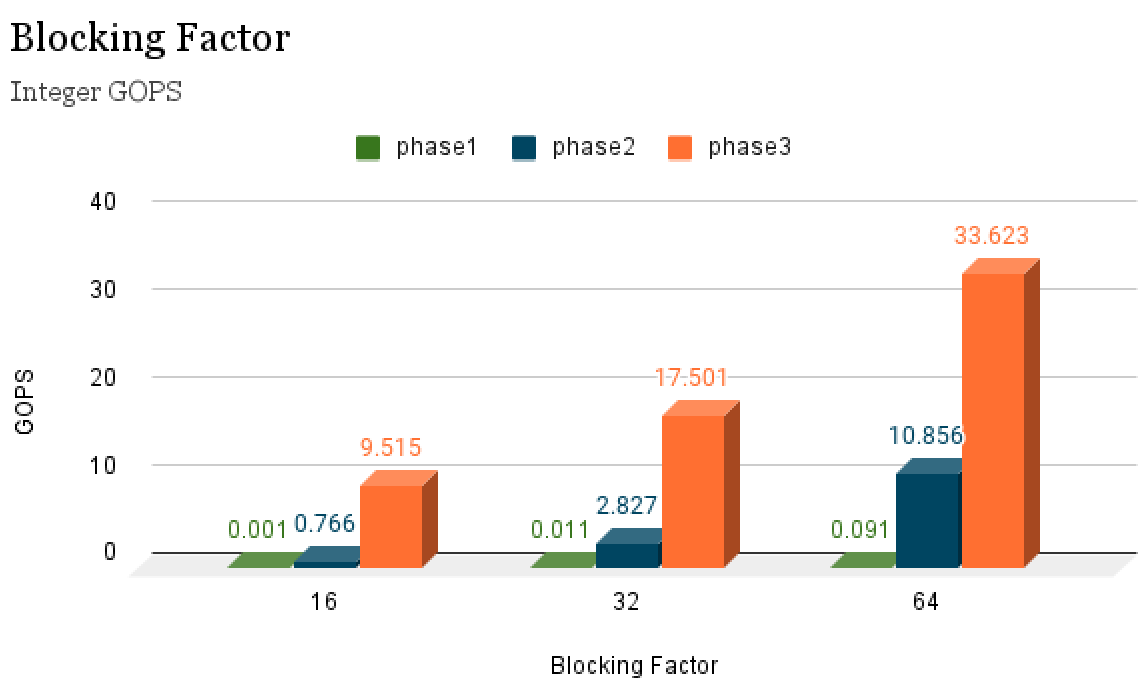
Blocking Factor

```
nvprof --metrics \
inst_integer,\
shared_load_throughput,shared_store_throughput,\
gld_throughput,gst_throughput \
./hw3-2 testcase out
```

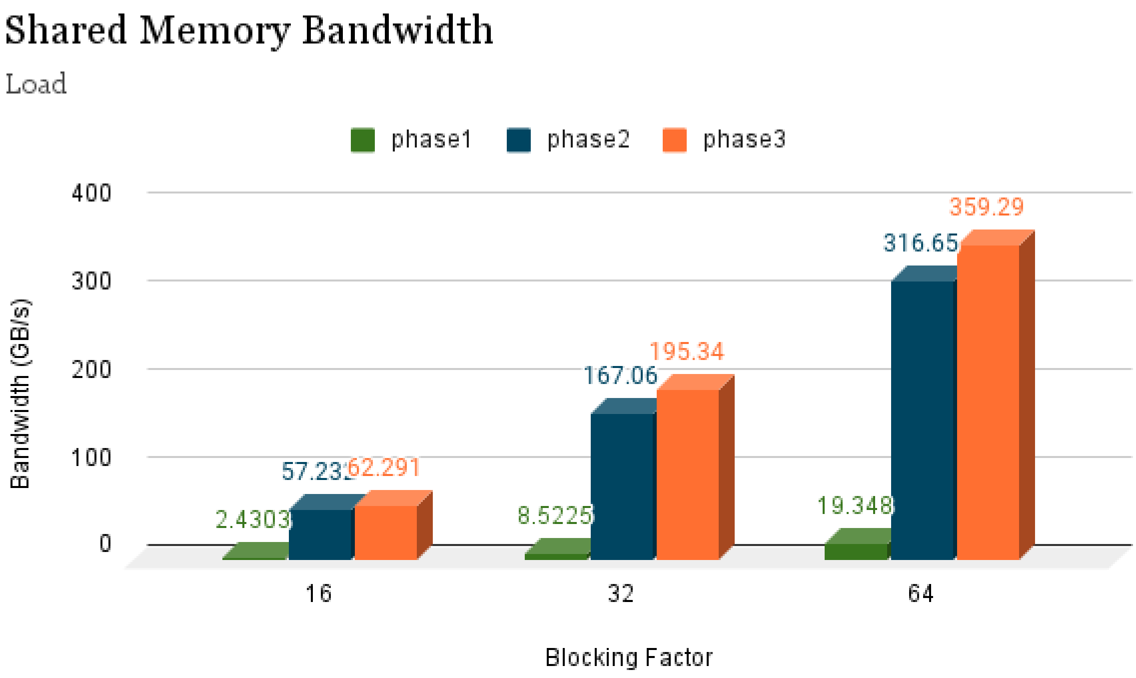
更改blocking factor時我只有更改以下變數，算法一樣是一個thread一次處理四個節點。

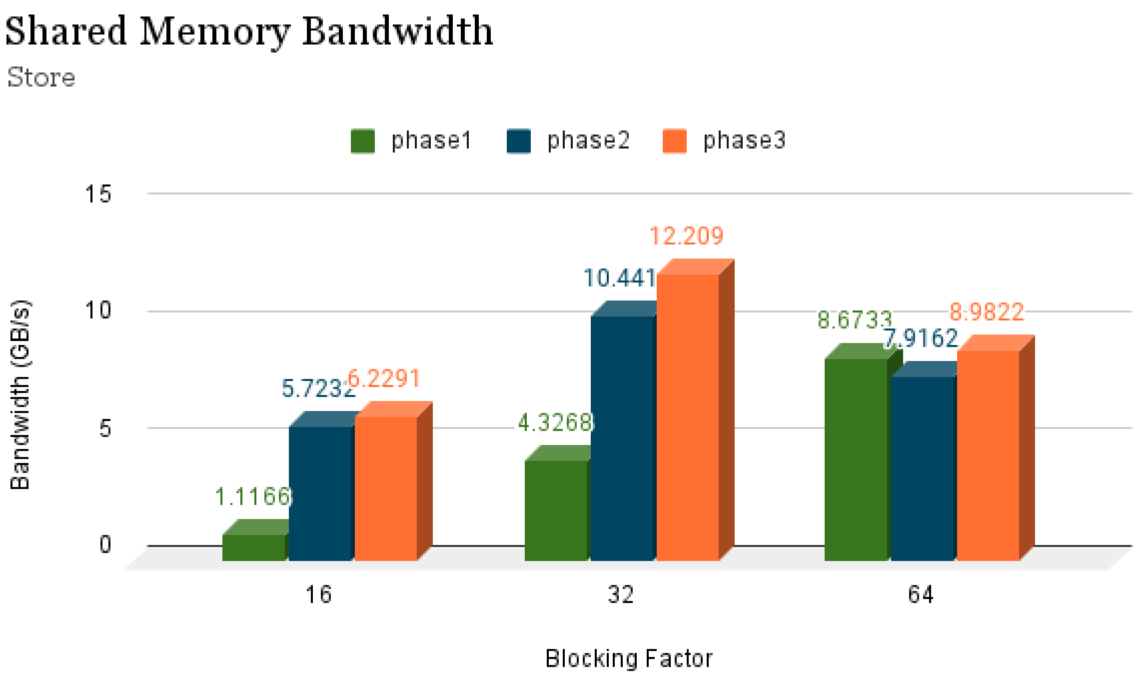
```
const int INF = ((1 << 30) - 1);
const int B = 64;
const int n_thread = 32;
const int n_per_iter = 4;
const int n_sm_size = 4096;
const int shift = 6;
```

Integer GOPS

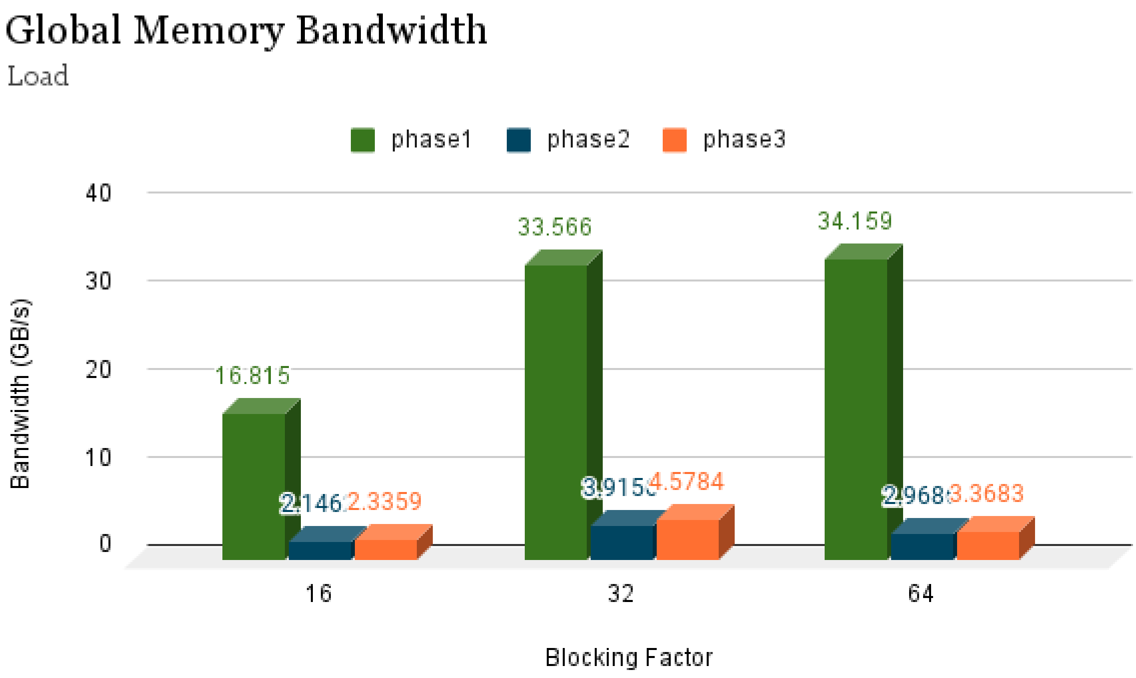


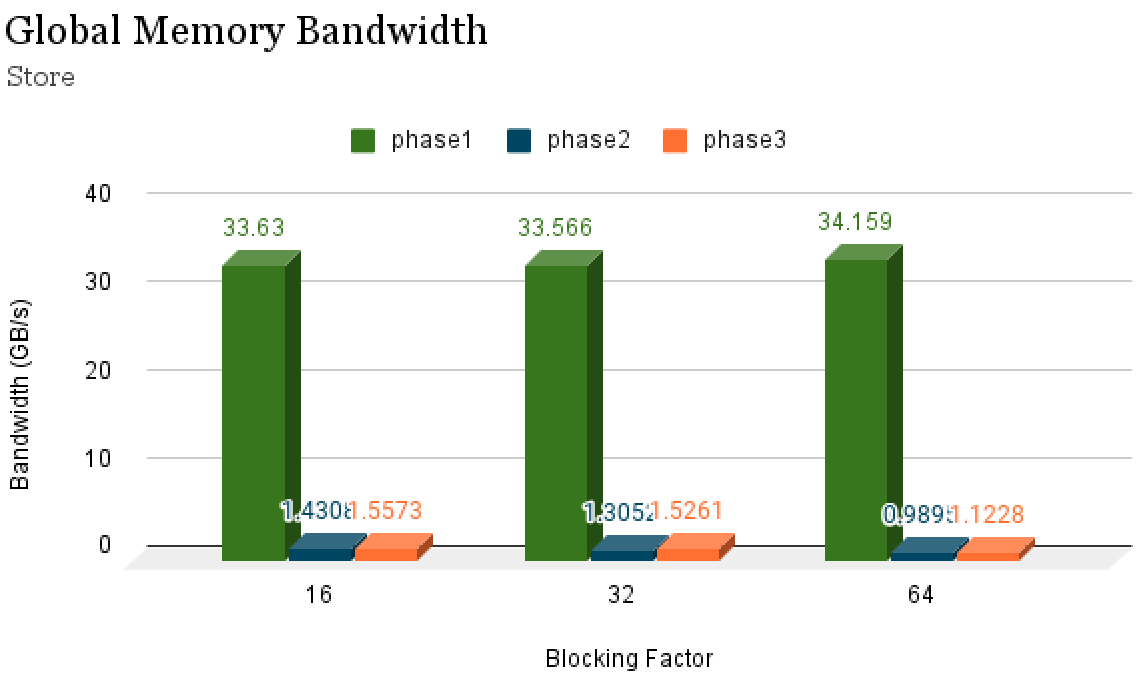
Shared Memory Bandwidth





Global Memory Bandwidth



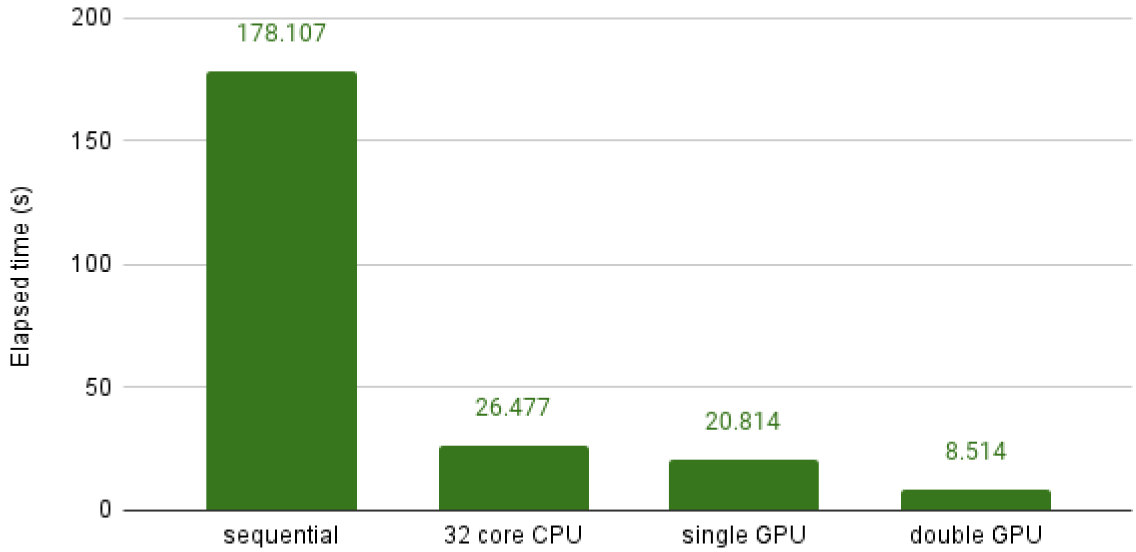


Optimization

時間都是用gettimeofday()測出來的。

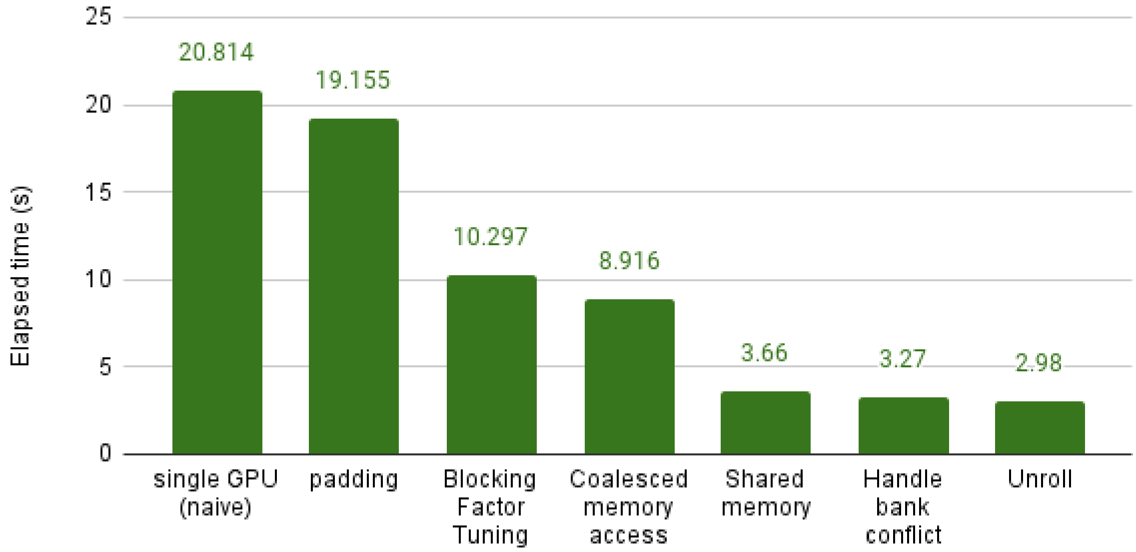
Optimization between naive version of different setup

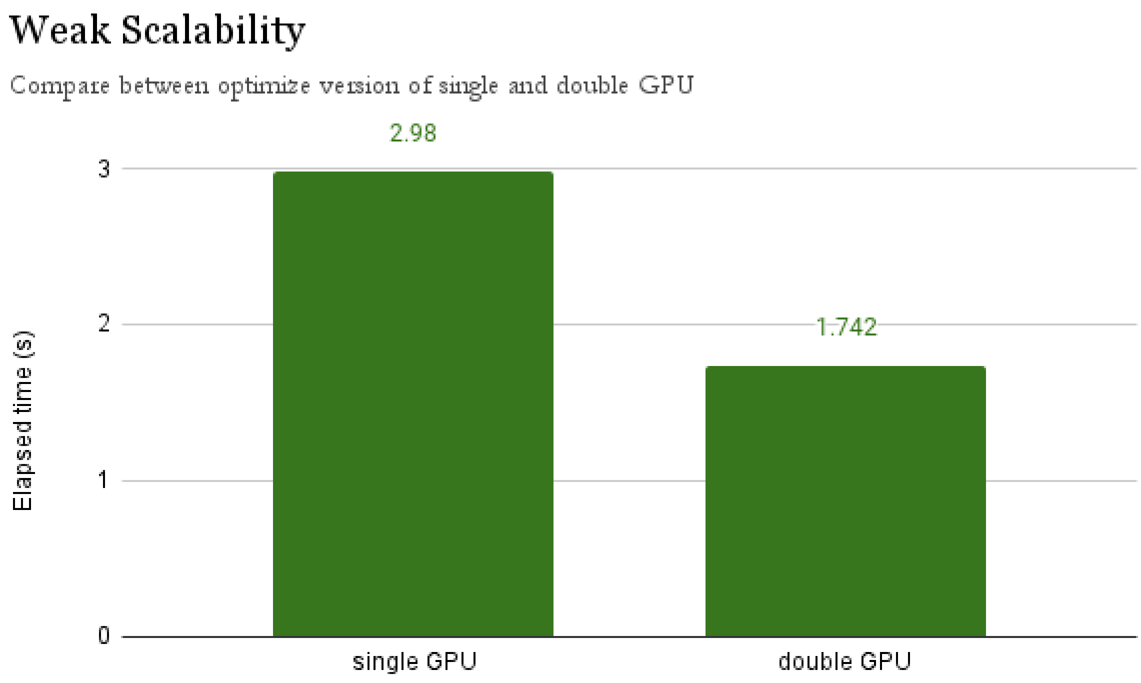
Base on sequential version with testcase c20.2

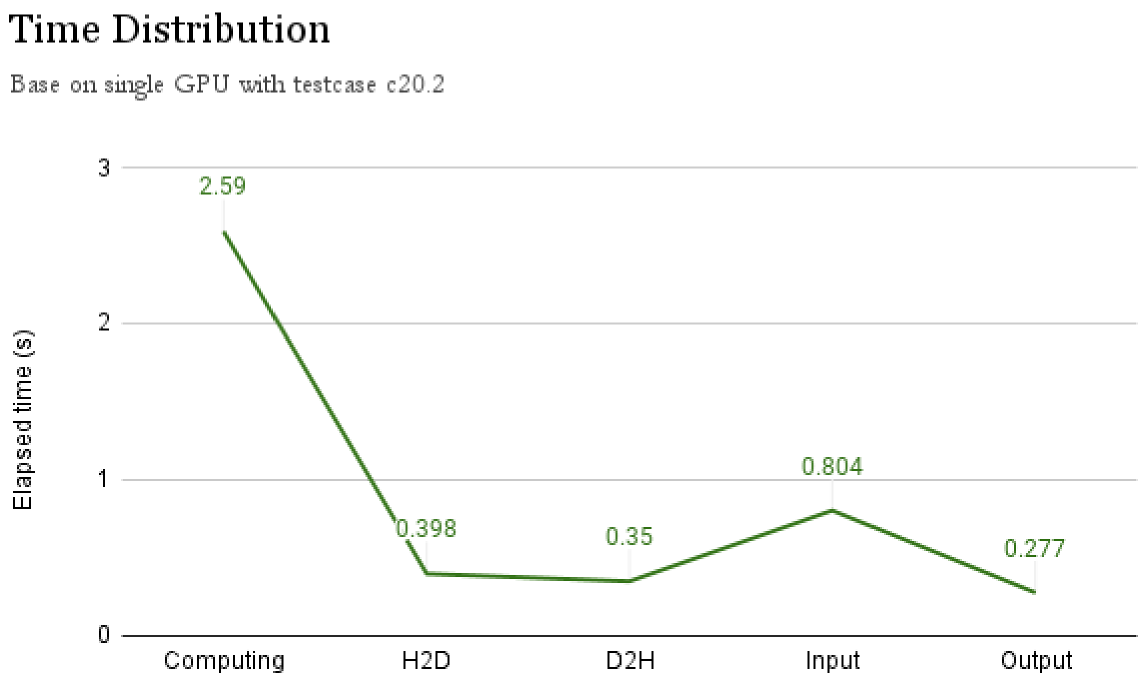


Optimization on Single GPU version

Base on naive single gpu version with testcase c20.2







Experience & Conclusion

What have you learned from this homework?

我學到cuda的同步化非常重要，以及如何善用shared memory。另外，資料的預處理對平行也是非常重要的，這次題目中的blocking factor的選擇和padding都大大影響了performance，還有memory access的優化則是最後要注意的點，預處理後如果能進一步優化memory access就能將computing以外的時間壓低許多。

最困難的點應該是debug，根本看不出來哪裡寫錯了，因為資料都被處理過了，頭很痛。

Feedback

love <3