Parallel Programming HW3

108032053 陳凱揚

Implementation

Which algorithm do you choose in hw3-1?

我使用了Blocked Floyd-Warshall Algorithm來實作hw3-1。

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

- hw3-2

  首先我先將n*n的 `Dist` 做padding使得n為64的倍數 (64為我選擇的blocking factor)，接著以 64*64 筆data為單位送進一個GPU的blcok中執行。在每一個block中會有 32*32 個thread，每個thread會負責處理 4 筆data。

- hw3-3

  與hw3-2一樣，先做padding後，再以 64*64 筆data為單位送進一個GPU的block中執行，且每個thread處理 4 筆data。而因為有兩個GPU，我將所有data分為兩半，在phase3時，上半部的data交給GPU0做執行，下半部的交給GPU1做執行，並在每一round開始前傳遞當前pivot row的data給另一個GPU。

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

- hw3-2
  - blocking factor

    我設定的blocking factor為 64，這是因為每個GPU的block最多有 1024 個thread和 `48KB` 的share memory，需要盡量用盡所有資源。

    如果factor設為 32，每個thread會處理 1 筆資料，每筆資料需要 2 個int大小的share memory儲存data (`Dist[i][j] = Dist[i][k]+Dist[k][j]`)，也就是每個block會使用到 `4*2*32*32/1024 = 8KB` 的share memory，這樣會浪費掉很多空間。

    如果factor設為64，每個thread會處理 1 筆資料，每個block會使用 `32KB` 的share memory，比起factor為32時用了更多的資源。

  - blocks & threads

    這邊我的設定如下，每個GPU的block都對應到一個我們所切的block，而thread的數量固定為 32*32，每個thread處理 4 筆資料。

    而block的數量在不同phase時會不同，在phase1時，只有 1 個pivot blcok需要計算，所以只需要 1 個block；在phase2時，有pivot row和pivot column需要計算，且可以扣除pivot block，所以需要 2*(round-1) 個block；在phase3時，一樣扣除掉前面的部分，共需要 (round-1)*(round-1) 個block。

    ```
    int round = n/64;
    for(int r = 0; r < round; r++){
        phase1 <<<1, dim3(32, 32)>>> (B, r, Dist_GPU, n);
        phase2 <<<dim3(2, round-1), dim3(32, 32)>>> (B, r, Dist_GPU, n);
        phase3 <<<dim3(round-1, round-1), dim3(32, 32)>>> (B, r, Dist_GPU, n);
    }
    ```

- hw3-3
    - blocking factor & blocks & threads

      與hw3-2大致相同，只有在phase3時的block數量不同，需要切成上下兩半給 2 個GPU計算，若為奇數，會將多的那行row給下半部的GPU。

      ```
      int id = omp_get_thread_num(), round = n/64;
      int start = (round/2)*id, size = (round/2)+(round%2)*id;
      for(int r = 0; r < round; r++){
          int copy = (r>=start && r<start+size);
          cudaMemcpyPeer(Dist_GPU[!id]+(r*64*n), !id, Dist_GPU[id]+(r*64*n), id, copy*64*n*sizeof(int));
      #pragma omp barrier
          phase1 <<<1, dim3(32, 32)>>> (B, r, Dist_GPU[id], n);
          phase2 <<<dim3(2, round-1), dim3(32, 32)>>> (B, r, Dist_GPU[id], n);
          phase3 <<<dim3(size, round-1), dim3(32, 32)>>> (B, r, Dist_GPU[id], n, start);
      }
      ```

How do you implement the communication in hw3-3?

我使用 `cudaMemcpyPeer` 在 2 個GPU間傳遞資料，而不是透過很慢的Host來傳遞資料。在每個round，都會將負責此部分的GPU將pivot row傳遞給另一個GPU，詳細原因在下面implement的部分會再說明。

```
void block_FW(int B){
#pragma omp parallel num_threads(2)
{
    int id = omp_get_thread_num(), round = n/64;
    int start = (round/2)*id, size = (round/2)+(round%2)*id;
    cudaSetDevice(id);
    cudaMalloc(&Dist_GPU[id], n*n*sizeof(int));
#pragma omp barrier
    cudaMemcpy(Dist_GPU[id]+(start*64*n), Dist+(start*64*n), size*64*n*sizeof(int), cudaMemcpyHostToDevice);
    for(int r = 0; r < round; r++){
        int copy = (r>=start && r<start+size);
        cudaMemcpyPeer(Dist_GPU[!id]+(r*64*n), !id, Dist_GPU[id]+(r*64*n), id, copy*64*n*sizeof(int));
#pragma omp barrier
        phase1 <<<1, dim3(32, 32)>>> (B, r, Dist_GPU[id], n);
        phase2 <<<dim3(2, round-1), dim3(32, 32)>>> (B, r, Dist_GPU[id], n);
        phase3 <<<dim3(size, round-1), dim3(32, 32)>>> (B, r, Dist_GPU[id], n, start);
    }
    cudaMemcpy(Dist+(start*64*n), Dist_GPU[id]+(start*64*n), size*64*n*sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(Dist_GPU[id]);
}
}
```

Briefly describe your implementations in diagrams, figures or sentences.

- hw3-1
    - OpenMP

      在hw3-1裡，我採取與seq.cc差不多的架構，只是在cal function裡將block以OpenMP做平行化，並將schedule的方式設為dynamic。此外，我將block factor設為64，這是實驗過不同數字後得到最佳的設定。

    - SIMD

      我在迴圈最內層做計算時，採用SIMD的方式，一次可以做 4 個計算。

```
void cal(
    int B, int Round, int block_start_x, int block_start_y, int block_width, int block_height) {
    int block_end_x = block_start_x + block_height;
    int block_end_y = block_start_y + block_width;
    int k_start = Round*B, k_end = min((Round+1)*B, n);

    #pragma omp parallel for num_threads(cpu_num) schedule(dynamic)
    for (int b_i = block_start_x; b_i < block_end_x; ++b_i) {
        int block_internal_start_x = b_i*B, block_internal_end_x = min((b_i+1)*B, n);
        for (int b_j = block_start_y; b_j < block_end_y; ++b_j) {
            int block_internal_start_y = b_j*B, block_internal_end_y = min((b_j+1)*B, n);
            for (int k = k_start; k < k_end; ++k) {
                for (int i = block_internal_start_x; i < block_internal_end_x; ++i) {
                    __m128i SIMD_ik = _mm_set1_epi32(Dist[i][k]);
                    for (int j = block_internal_start_y; j+3 < block_internal_end_y; j += 4) {
                        __m128i SIMD_l = _mm_loadu_si128((__m128i*)&Dist[i][j]);;
                        __m128i SIMD_r = _mm_add_epi32(SIMD_ik, _mm_loadu_si128((__m128i*)&Dist[k][j]));
                        _mm_storeu_si128((__m128i*)&Dist[i][j], _mm_min_epi32(SIMD_l, SIMD_r));
                    }
                    int j = block_internal_start_y+(block_internal_end_y-block_internal_start_y)/4*4;
                    while(j < block_internal_end_y){
                        Dist[i][j] = min(Dist[i][j], Dist[i][k]+Dist[k][j]);
                        j++;
                    }
                }
            }
        }
    }
}
```

- hw3-2
    - Padding & Pin memory

      如同前面所說，將 `Dist` 的大小padding成 64 的倍數，並使用 `cudaMallocHost` 來allocate memory，用以pin住這整塊記憶體。

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

n_origin = n;
n += 64-((n%64+64-1)%64+1);
cudaMallocHost(&Dist, n*n*sizeof(int));
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        Dist[i*n+j] = (i==j&&i<n_origin)?0:INF;
    }
}
```

- Kernel & Share Memory

  我將有dependency的 3 個phase寫成 3 個kernel來執行，每個kernel的block數就是要計算的block數量，每個block有 32*32 個thread，每個thread負責 4 筆資料。

  以下以phase3為例，每個thread負責的data會是此block中的 `(i, j), (i, j+32), (i+32, j), (i+32, j+32)` ，其中 `i` , `j` 為 `threadIdx.y` , `threadIdx.x` 。首先會開一個大小為 2*64*64 大小的int陣列，負責儲存所有這個block會access到的memory， `s` 的前半部儲存在做floyd-warshall時的 `Dist[i][k]` 所屬的那個block，後半部儲存 `Dist[k][j]` 所屬的那個block，只是在程式碼中會將所有東西壓成 1 個 1D 陣列，所以看起來會比較複雜。接著在for迴圈開始計算前，需要做 `__syncthreads()` ，確保所有thread都將資料先讀進了share memory中。

  phase1和phase2也與phase3大同小異，只是在陣列編碼的部分有些微不同，另外在phase1時，因為 `Dist[i][j]` , `Dist[i][k]` , `Dist[k][j]` 皆屬於同一塊block，所以在每次迴圈中必須將答案寫回share memory，並且做 `__syncthreads()` 。

```
int round = n/64;
for(int r = 0; r < round; r++){
    phase1 <<<1, dim3(32, 32)>>> (B, r, Dist_GPU, n);
    phase2 <<<dim3(2, round-1), dim3(32, 32)>>> (B, r, Dist_GPU, n);
    phase3 <<<dim3(round-1, round-1), dim3(32, 32)>>> (B, r, Dist_GPU, n);
}
```

```
__global__ void phase3(int B, int r, int *Dist_GPU, int n){
    __shared__ int s[2*64*64];
    int b_i = (blockIdx.x+(blockIdx.x>=r))<<6, b_j = (blockIdx.y+(blockIdx.y>=r))<<6, b_k = r<<6;
    int i = threadIdx.y, j = threadIdx.x;

    int val0 = Dist_GPU[(b_i+i)*n+(b_j+j)];
    int val1 = Dist_GPU[(b_i+i)*n+(b_j+(j+32))];
    int val2 = Dist_GPU[(b_i+(i+32))*n+(b_j+j)];
    int val3 = Dist_GPU[(b_i+(i+32))*n+(b_j+(j+32))];

    s[i*64+j] = Dist_GPU[(b_i+i)*n+(b_k+j)];
    s[i*64+(j+32)] = Dist_GPU[(b_i+i)*n+(b_k+(j+32))];
    s[(i+32)*64+j] = Dist_GPU[(b_i+(i+32))*n+(b_k+j)];
    s[(i+32)*64+(j+32)] = Dist_GPU[(b_i+(i+32))*n+(b_k+(j+32))];

    s[4096+i*64+j] = Dist_GPU[(b_k+i)*n+(b_j+j)];
    s[4096+i*64+(j+32)] = Dist_GPU[(b_k+i)*n+(b_j+(j+32))];
    s[4096+(i+32)*64+j] = Dist_GPU[(b_k+(i+32))*n+(b_j+j)];
    s[4096+(i+32)*64+(j+32)] = Dist_GPU[(b_k+(i+32))*n+(b_j+(j+32))];

    __syncthreads();
    #pragma unroll
    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)]);
    }

    Dist_GPU[(b_i+i)*n+(b_j+j)] = val0;
    Dist_GPU[(b_i+i)*n+(b_j+(j+32))] = val1;
    Dist_GPU[(b_i+(i+32))*n+(b_j+j)] = val2;
    Dist_GPU[(b_i+(i+32))*n+(b_j+(j+32))] = val3;
}
```

- hw3-3
  - Dependency in phase3

    hw3-3與hw3-2幾乎相同，只是在phase3時，1 個GPU只需負責一半的data，因為和phase3有dependency的資料只有pivot row和pivot column，以pivot row為界線，可以發現下半部的pivot column並不會被計算到，所以其實只需要在每次計算前，負責計算此pivot row的GPU將此pivot row傳送給另一個GPU，這樣會被使用到的整個pivot row就會是正確，而上半部的pivot column則在之前就傳送過，所以也會是正確。這樣子在計算phase3時，所需的dependency皆為正確的，也因此可以計算出正確的答案。

```
void block_FW(int B){
#pragma omp parallel num_threads(2)
{
    int id = omp_get_thread_num(), round = n/64;
    int start = (round/2)*id, size = (round/2)+(round%2)*id;
    cudaSetDevice(id);
    cudaMalloc(&Dist_GPU[id], n*n*sizeof(int));
#pragma omp barrier
    cudaMemcpy(Dist_GPU[id]+(start*64*n), Dist+(start*64*n), size*64*n*sizeof(int), cudaMemcpyHostToDevice);
    for(int r = 0; r < round; r++){
        int copy = (r>=start && r<start+size);
        cudaMemcpyPeer(Dist_GPU[!id]+(r*64*n), !id, Dist_GPU[id]+(r*64*n), id, copy*64*n*sizeof(int));
#pragma omp barrier
        phase1 <<<1, dim3(32, 32)>>> (B, r, Dist_GPU[id], n);
        phase2 <<<dim3(2, round-1), dim3(32, 32)>>> (B, r, Dist_GPU[id], n);
        phase3 <<<dim3(size, round-1), dim3(32, 32)>>> (B, r, Dist_GPU[id], n, start);
    }
    cudaMemcpy(Dist+(start*64*n), Dist_GPU[id]+(start*64*n), size*64*n*sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(Dist_GPU[id]);
}
}
```

Profiling Results

- Occupancy, Sm efficiency, Shared memory load/store throughput, Global load/store throughput

以下為使用 `c21.1` 為測資做profiling的結果，可以看到在phase2和phase3處理的資料比較龐大，不管在occupancy、sm efficiency、各種throughput上都比只處理 1 個block的phase1來的大上許多，將資源更加充分利用。

```
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GTX 1080 (0)"
    Kernel: phase3(int, int, int*, int)
         79                        achieved_occupancy                        Achieved Occupancy    0.924328    0.927549    0.926039
         79                             sm_efficiency                   Multiprocessor Activity      99.52%      99.72%      99.62%
         79                     shared_load_throughput             Shared Memory Load Throughput  3213.5GB/s  3277.2GB/s  3249.2GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  133.90GB/s  136.55GB/s  135.39GB/s
         79                            gld_throughput                    Global Load Throughput  200.84GB/s  204.83GB/s  203.08GB/s
         79                            gst_throughput                   Global Store Throughput  66.948GB/s  68.275GB/s  67.693GB/s
    Kernel: phase1(int, int, int*, int)
         79                        achieved_occupancy                        Achieved Occupancy    0.499183    0.499220    0.499204
         79                             sm_efficiency                   Multiprocessor Activity       4.53%       4.63%       4.59%
         79                     shared_load_throughput             Shared Memory Load Throughput  111.29GB/s  129.07GB/s  127.39GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  37.676GB/s  43.696GB/s  43.126GB/s
         79                            gld_throughput                    Global Load Throughput  593.54MB/s  688.39MB/s  679.40MB/s
         79                            gst_throughput                   Global Store Throughput  593.54MB/s  688.39MB/s  679.40MB/s
    Kernel: phase2(int, int, int*, int)
         79                        achieved_occupancy                        Achieved Occupancy    0.892943    0.921808    0.904688
         79                             sm_efficiency                   Multiprocessor Activity      85.93%      91.81%      89.55%
         79                     shared_load_throughput             Shared Memory Load Throughput  2705.0GB/s  2938.0GB/s  2800.3GB/s
         79                    shared_store_throughput            Shared Memory Store Throughput  112.71GB/s  122.42GB/s  116.68GB/s
         79                            gld_throughput                    Global Load Throughput  169.06GB/s  183.62GB/s  175.02GB/s
         79                            gst_throughput                   Global Store Throughput  56.353GB/s  61.208GB/s  58.341GB/s
```

Experiment & Analysis

System Spec

我的測試是在課程所提供的 `hades` 上做的。

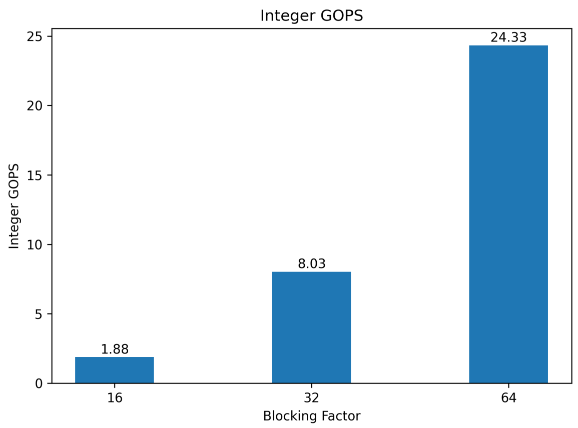
Blocking Factor

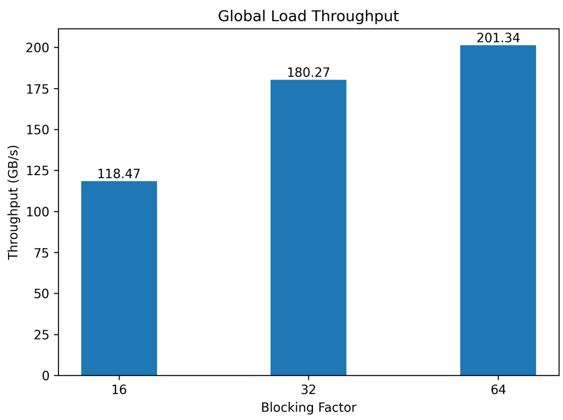
以下為phase3的數據，並以 `c21.1` 為測資所測量。

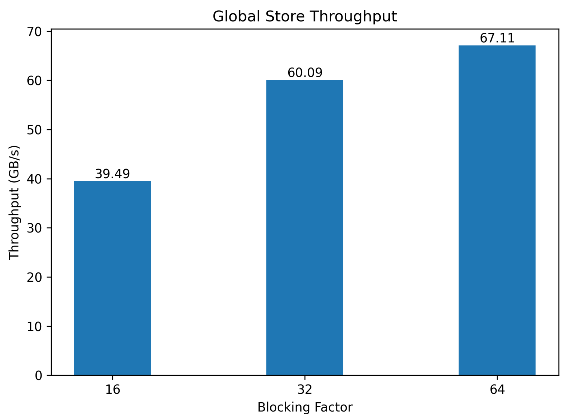
- Method of measurement

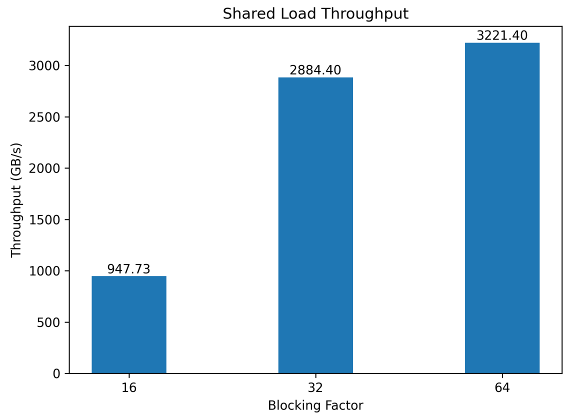
```
IN=/home/pp22/share/hw3-2/cases/c21.1
OUT=/dev/null
NVPROF_METRIC=inst_integer,shared_load_throughput,shared_store_throughput,gld_throughput,gst_throughput
for i in {16,32,64};
do
    echo -e "\n############### Blocking Factor: $i ###############"
    echo -e "\n########## Time ##########"
    srun -p prof -N1 -n1 --gres=gpu:1 nvprof ./hw3-2-$i $IN $OUT
    echo -e "\n########## Metric ##########"
    srun -p prof -N1 -n1 --gres=gpu:1 nvprof -m $NVPROF_METRIC ./hw3-2-$i $IN $OUT
done
```

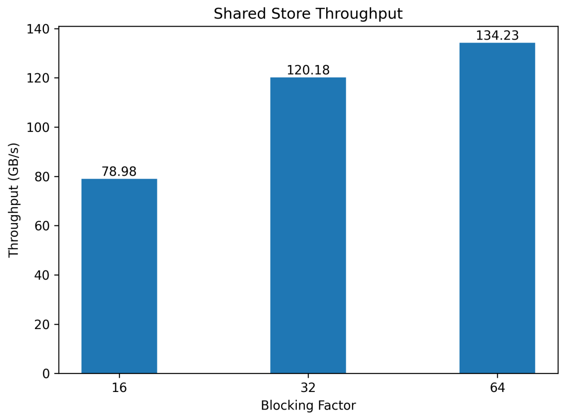
- Result











Optimization

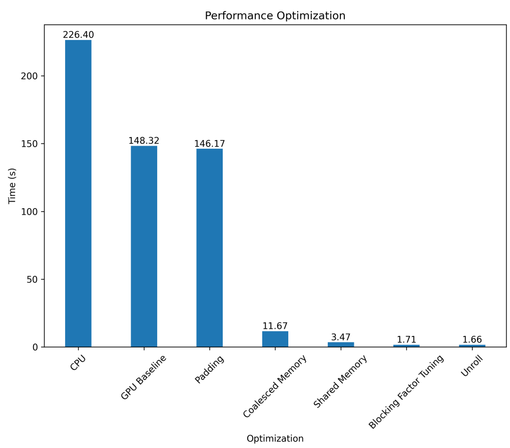
以下數據是以 `p11k1` 為測資所測量出來的。

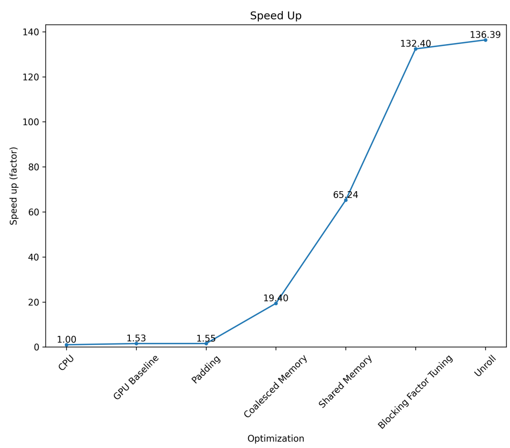
- Method of measurement

```
struct timeval start, end;
gettimeofday(&start, NULL);
// ...
gettimeofday(&end, NULL);
double time = (double)(US_PER_SEC*(end.tv_sec-start.tv_sec)+(end.tv_usec-start.tv_usec))/US_PER_SEC;
printf("%.2lf\n", time);
```

```
IN=/home/pp22/share/hw3-2/cases/p11k1
OUT=/dev/null
for i in {CPU,GPU_Baseline,Padding,Coalesced_Memory,Shared_Memory,Blocking_Factor_Tuning,Unroll};
do
    echo -e "\n############### Optimization: $(python -c "print(' '.join('$i'.split('_')),end='')") ###############"
    echo -e "\n########## Time ##########"
    srun -N1 -n1 --gres=gpu:1 ./hw3_$i $IN $OUT
done
```

- Result





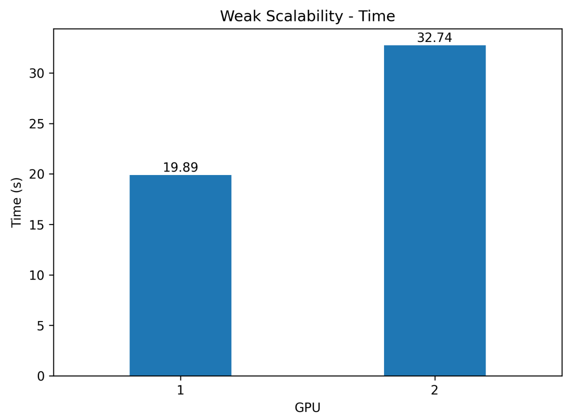
Weak Scalability

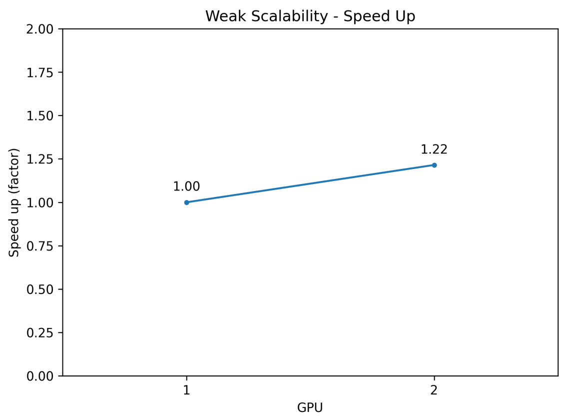
以下數據是以 `p25k1` 和 `p35k1` 為測資所測量出來的

- Method of measurement

```
# 1 GPU, n1 = 25000, n1*n1 = 6.25e8
IN1=/home/pp22/share/hw3-2/cases/p25k1
# 2 GPU, n2 = 34921, n2*n2 = 12.19e8 ≈ 2*n1*n1
IN2=/home/pp22/share/hw3-3/cases/p35k1
OUT=/dev/null
echo -e "\n############### Weak Scalability: 1 GPU ###############"
echo -e "\n########## Time ##########"
srun -N1 -n1 --gres=gpu:1 ./hw3-2 $IN1 $OUT
echo -e "\n############### Weak Scalability: 2 GPU ###############"
echo -e "\n########## Time ##########"
srun -N1 -n2 --gres=gpu:2 ./hw3-3 $IN2 $OUT
```

- Result



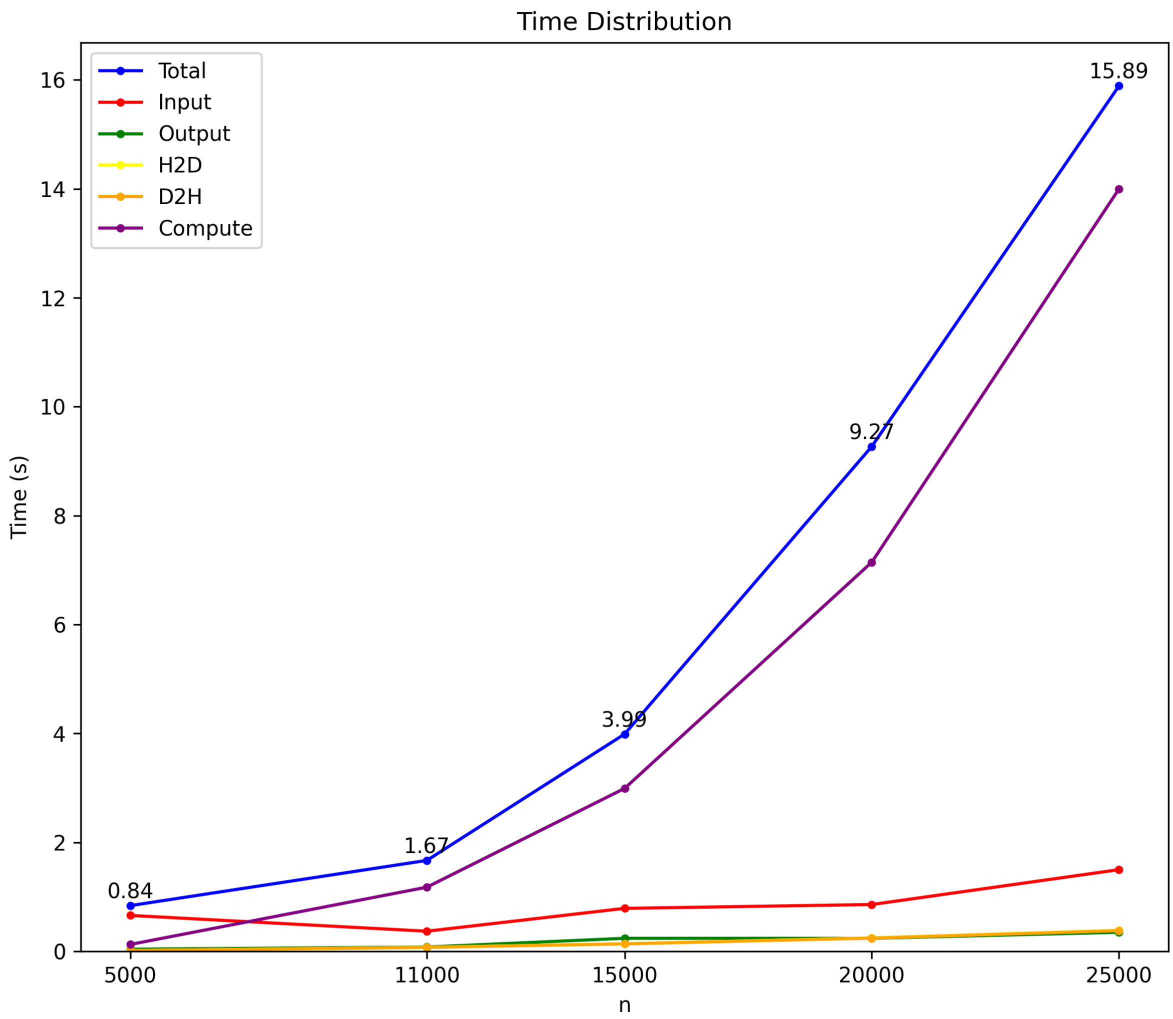


Time Distribution

- Method of measurement

```
DIR=/home/pp22/share/hw3-2/cases/
IN=(c21.1 p11k1 p15k1 p20k1 p25k1)
N=(5000 11000 15000 20000 25000)
OUT=/dev/null
for i in $(seq 0 4);
do
    echo -e "\n############### Time Distribution: ${IN[$i]} - N=${N[$i]} ###############"
    echo -e "\n########## Time ##########"
    srun -N1 -n1 --gres=gpu:1 ./hw3-2 $DIR${IN[$i]} $OUT
    echo -e "\n########## Metric ##########"
    srun -p prof -N1 -n1 --gres=gpu:1 nvprof ./hw3-2 $DIR${IN[$i]} $OUT > $OUT
done
```

- Result



Experience & conclusion

What have you learned from this homework?

我覺得這次的作業讓我對GPU更加熟悉，尤其是在對GPU的架構和他的計算模式，不然很難對cuda做優化，因為和平常寫程式只需注意時間、空間複雜度非常不一樣，cuda的重點都在memory access上，而且還有很多層memory需要一一去優化，有時也需要做trade off。此外，優化後的cuda程式常常都很難讀，也加深了繼續優化的難度，必須對整個架構很清楚，才有辦法找到效能瓶頸。