HW3 All-Pair Shortest Path

109062101 許佳綺

Implementation

hw3-1 (CPU)

使用基本的Floyd Warshall去實作，由於第k+1輪是 depends on 第k輪，所以只有每次的i, j這部分可以平行，我使用 `OpenMP` 並用 `schedule(static)` 讓每個thread去平分所有第i點到其他vertex的運算。

```
for(int k = 0; k < v; k++){
      #pragma omp parallel for schedule(static) num_threads(numOfThread)
          for(int i = 0; i < v; i++){
              for(int j = 0; j < v; j++){
                  dis[i][j] = min(dis[i][j], dis[i][k]+dis[k][j]);
              }
          }
  }
```

hw3-2 (Single GPU)

以下內容會以最終實作版本來闡述(包含Optimize的部分)

使用Block Floyd Warshall。我讓GPU中的每個gblock都負責一個Floyd Warshall的fblock，每個thread負責一個fblock 中的4個位置 $(i,j),(i+B/2,j),(i,j+B/2),(i+B/2,j+B/2)$，這是為了要最大化shared memory的使用。(後面解釋)

在phase1先做完該round的pivot block所含點之間的兩兩距離，因此單個gblock需要 $1*B*B$的shared memoryf去儲存一個pivot fblock所會用到的value，而phase2是處理與phase1做完的pivot fblock同row和同col的fblock，所以單個gblock需要 $3*B*B$的shared memory，而phase3是根據phase2做完的一個row fblock跟一個col fblock去算出位在該(row, col)的fblock，因此在這個階段單個gblock也需要$3*B*B$的shared memory。

而透過`cudaGetDeviceProperties`可以知道 `maxThreasPerBlock = 1024`，`sharedMemPerBlock = 49152`，而49152=3*4*64*64，一個integer是4 byte，在phase2跟phase3都需要4 * 3 * B * B所以我Block Factor(B) 選用64，number of Thread則使用32*32。

首先在Phase1的部分，我使用B*B的shared memory，先把每個thread所負責的4個位置的value都load到shared memory，並用syncthread()保證資料載入完畢，接著iterate過整個block pivot，去同時更新這四個位置。並且因為第k+1輪是depends on第k輪的結果，因此需要syncthreads()同步第k輪的所有結果。最後再將算好的位置存回`dist`中。

```
__global__ void block_FW_p1(int* dist, int round, int n){
    __shared__ int shr[Blocksize][Blocksize];
    int x = threadIdx.x; // col
    int y = threadIdx.y; // row

    int c = round * Blocksize + threadIdx.x;
    int r = round * Blocksize + threadIdx.y;
```

```
    shr[y][x] = dist[r * n + c];
    shr[y + Half][x] = dist[(r + Half) * n + c];
    shr[y][x + Half] = dist[r * n + (c + Half)];
    shr[y + Half][x + Half] = dist[(r + Half) * n + (c + Half)];

    __syncthreads();

    #pragma unroll 32
    for(int i = 0; i < Blocksize; i++){
        shr[y][x] = min(shr[y][x], shr[y][i] + shr[i][x]);
        shr[y + Half][x] = ...
        shr[y][x + Half] = ...
        shr[y + Half][x + Half] = ...
        __syncthreads();
    }

    dist[r * n + c] = shr[y][x];
    dist[(r + Half) * n + c] = shr[y + Half][x];
    dist[r * n + (c + Half)] = shr[y][x + Half];
    dist[(r + Half) * n + (c + Half)] = shr[y + Half][x + Half];
    return;
}
```

接著是Phase2的部分，我使用N/B個gblock去計算(會跳過Phase1負責的該block)，每一個thread負責一個與pivot 同row的fblock和一個與pivot 同col的fblock。和phase1類似，先把負責的position value載入，用syncthread()保證資料載入完畢，接著iterate過整個block，去同時更新這八個位置。並且因為第k+1輪是depends on第k輪的結果，因此需要syncthreads()同步第k輪的所有結果。最後再將算好的位置存回`dist`中。

接著是Phase3的部分，我使用N/B*N/B個gblock去計算(會跳過Phase1跟Phase2負責的blocks)，每一個thread負責一個fblock，並且需要用到跟該fblock同row跟同col的兩個fblock資料。先把需要的position value載入，用syncthread()保證資料載入完畢，接著iterate過整個block，去同時更新這四個位置。而因為phase3只depends on phase2運算的結果跟同一個點的運算結果，所以不需要syncthread去同步。最後再將算好的位置存回`dist`中。

而在這些流程中，我也做了像是memory padding將輸入的v補成是B的倍數，讓接下來存取的時候不用再有另外的if-else去判斷界線。以及使用`#pragma omp unroll`讓編譯時能將For-loop攤開優化效率。

hw3-3 (Multiple GPU)

hw3-3中我只在block_FW_p3做優化，因為phase1只需要算1個block，phase2需要做2*N/B - 1個block，phase3需要做(N/B)*(N/B) - 2*N/B)個block，因此最主要的運算都卡在這邊。

一開始會由CPU將整張圖複製給兩張GPU，接者再每輪中，我使用`cudaMemcpyPeer()`去讓兩個GPU互相溝通，基本上我把Phase3的運算部分切割成上下兩part，在第i輪時將負責第i個row的block的資料複製到另一個GPU上。並使用#pragma omp barrier等待所有同步，再進行接下來的運算。

```
// sudo code
#pragma omp parallel num_threads(2)
{
```

```
    int id = omp_get_thread_num();
    cudaSetDevice(id);
    cudaDeviceEnablePeerAccess(!id, 0);
    cudaMalloc(&(ddist[id]), n * n * sizeof(int));
    cudaMemcpy(ddist[id], Dist, n * n * sizeof(int), cudaMemcpyHostToDevice);

    dim3 num_blocks_p3(B, B / 2);
    int row_offset = 0;
    if(id){
        row_offset = B / 2;
        if(B & 1) num_blocks_p3.y++;
    }

    for(int i = 0; i < B; i++){
        if(!id && i < B / 2){
            cudaMemcpyPeer(0 -> 1);
        }else if(id && i >= B / 2){
            cudaMemcpyPeer(1 -> 0);
        }
        #pragma omp barrier
        block_FW_p1<<<num_blocks_p1, num_threads>>>(ddist[id], i, n);
        block_FW_p2<<<num_blocks_p2, num_threads>>>(ddist[id], i, n);
        block_FW_p3<<<num_blocks_p3, num_threads>>>(ddist[id], i, n, row_offs
    }
    cudaMemcpy(Device(GPU[id]) -> Host(CPU));
}
```

Profiling Results

這邊取最大計算量的 `block_FW_p3` 來profile

block_FW_p3	Min	Max	Avg
Achieved_Occupancy	0.917496	0.920175	0.918862
sm_efficiency	99.07%	99.74%	99.65%
shared_load_throughput	2771.2GB/s	2854.1GB/s	2826.0GB/s
shared_store_throughput	261.87GB/s	268.30GB/s	265.86GB/s
gst_throughput	58.412GB/s	59.786GB/s	59.341GB/s
gld_throughput	19.003GB/s	19.214GB/s	19.047GB/s

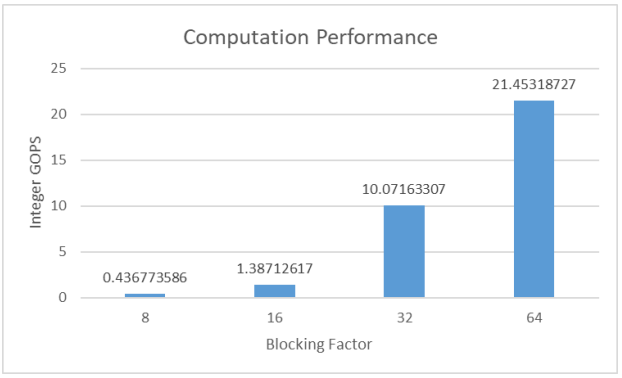
Experiment & Analysis

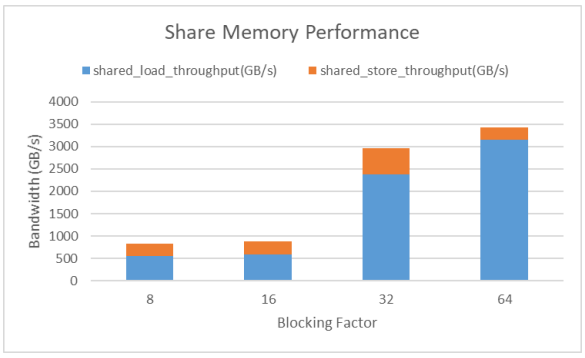
System Spec

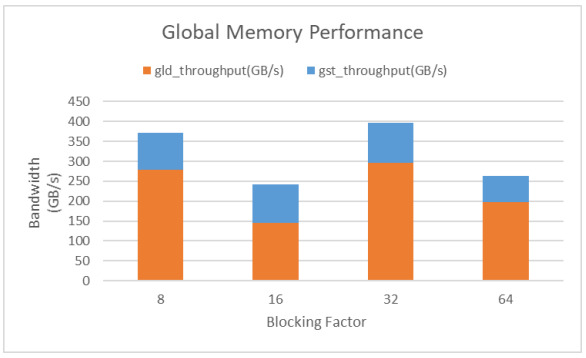
課程所提供的Hades環境

計算IO時間使用 `clock_gettime()` ，計算kernel function時間則使用 `nvprof` 的 `–metrixs` , `–print-gpu-summary` 。

Blocking Factor

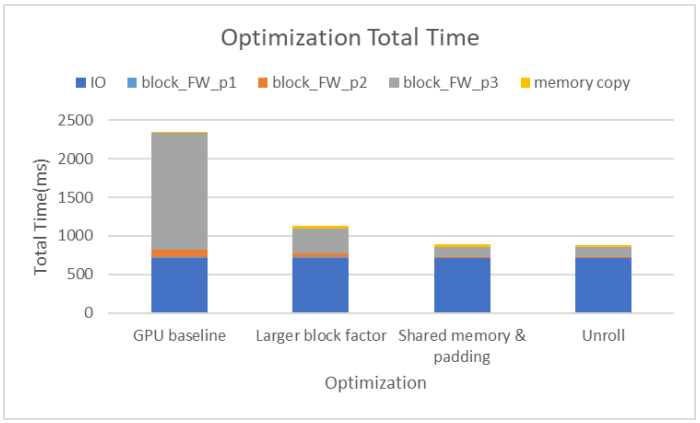






可以看到隨著block factor的遞增，GOPS也會跟得增加。而shared memory performance也跟Blocking Factor成正相關，代表blocking factor的增加能更善用share memory。而global memory performance 目前不太確定再block factor=16跟64時 global 的load throughput會下降，推測可能跟硬體架構有關(?)。

Optimization

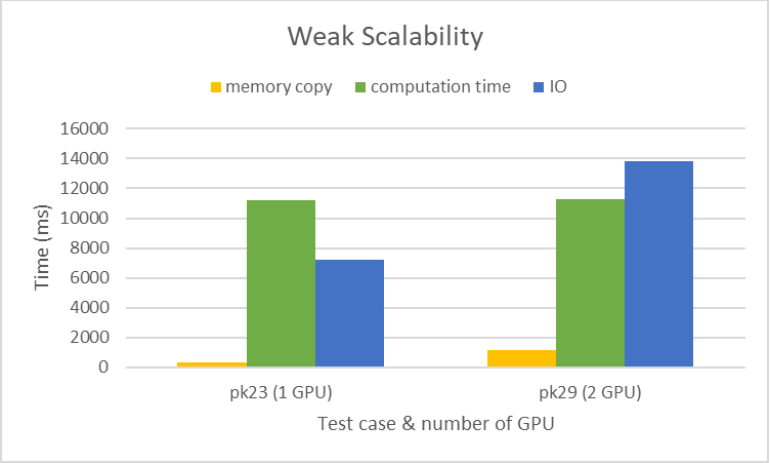


可以看到灰色部分(block_FW_p3) 會是bottle neck，因此一開始我將Block Factor從32改成64去優化後可以看到灰色部分可以減少非常多，更好的運用所有資源。接著做share memory跟padding的優化，就是將每個thread負責的計算從1個position變成4個position，又可以看到灰色部分能夠再減少1/2左右，而橘色部分(block_FW_p2)也會下降蠻多的。後續就是unroll的優化，可以再降低一點點執行時間。總體而言，IO時間比較固定，因此對computation部分做優化而能達到不錯的效果。

Weak Scalability

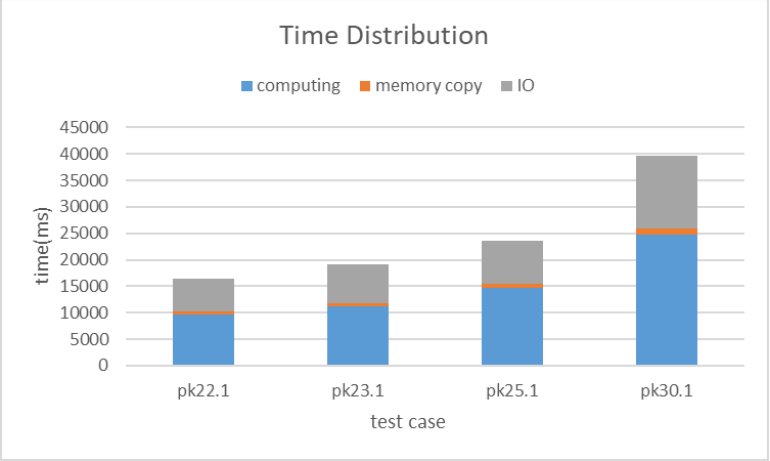
用 single GPU 跑 pk23 → v = 22973，用 2 GPU 跑 pk29 → u = 28911。

因為Weak Scalability是想驗證 在每個processor拿到的工作量相等下增加processor數量 的 執行時間，因此當computational resource 上升兩倍，problem size也需要從 $v^3 = 1.2124201e+13$ 變成 $u^3 = 2.4165141e+13$ (大概兩倍) (Floyd Warshell的time complexity是$O(n^3)$)。



可以從上圖看到，從1GPU變成2GPU，隨著計算量也上升兩倍，computation time還是能維持得差不多，總共的時間如果不包含IO時間還是能看出scalability還是很不錯的。

Time Distribution



我使用不同vertex size的test case去測試，可以看到不論是computation time 還是IO time都是隨著input增加而遞增，total time會跟input成正相關。

Experience & Conclusion

這次作業我卡最久的就是算位置，因為block floyd warshall每個phase的切法不太一樣，並且還有算每個thread負責的位置，我想這次的GPU作業讓我不僅僅是對這個演算法有更多的了解，同時也花了不少時間再理解硬體架構，讓成是能最大化硬體提供的資源。cuda的平行跟切割的邏輯需要非常清晰，平時也需要一些profiler的輔助來知道現在的優化是否有成效，透過這次作業讓我學到更多分析的方式，也對撰寫cuda程式有了更多了解。感謝助教與教授的用心出題及處理server問題。