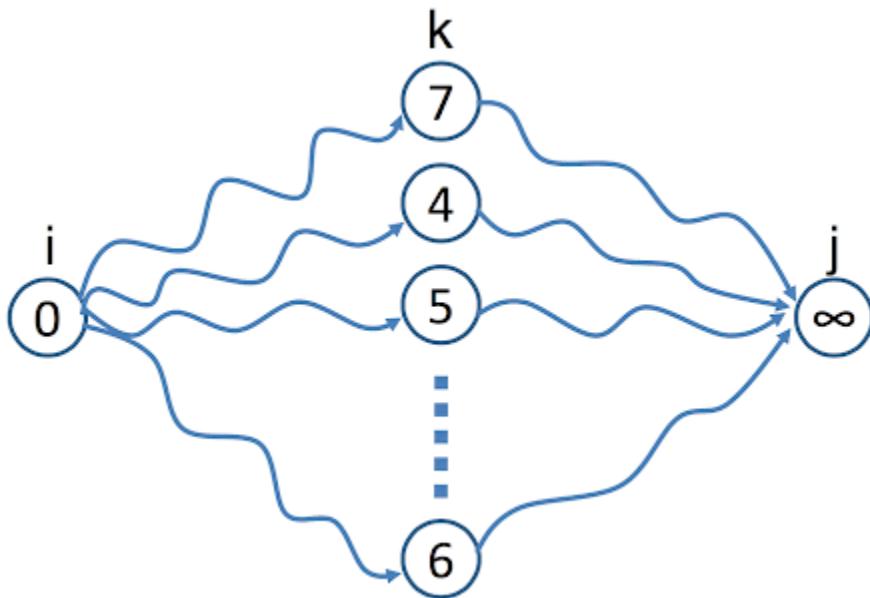


Parallel_Programming_HW3_Report

All-Pairs Shortest Path

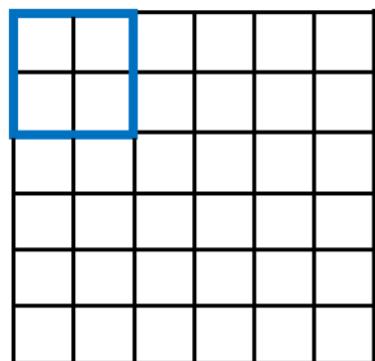
李浩榮 110062401



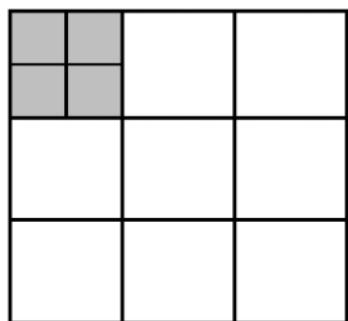
1. Implementation

這次的作業主要是會分成三個部分來實作 `All-Pairs Shortest Path` 的問題。其中會使用到 `Floyd-Warshall` 或 `Blocked Floyd-Warshall` 來做為實作的主要演算法。

1. hw3-1: 使用CPU
2. hw3-2: 使用一個GPU
3. hw3-3: 使用兩個GPU



$D_{(0,0)}$	$D_{(0,1)}$	$D_{(0,2)}$
$D_{(1,0)}$	$D_{(1,1)}$	$D_{(1,2)}$
$D_{(2,0)}$	$D_{(2,1)}$	$D_{(2,2)}$



Pivot block	Pivot row	Pivot row
Pivot column		
Pivot column		

(a) Phase 1

(b) Phase 2

(c) Phase 3

pivot		

(a) Round 1

(b) Round 2

(c) Round 3

			$D_{(0,2)}$
			$D_{(1,2)}$

(d) Phase 1

(e) Phase 2

(f) Phase 3

All-Pairs Shortest Path (CPU)

- 在hw3-1中我是使用 **Blocked Floyd-Warshall** 以及 **Pthread + OpenMP** 的 **hybrid** 版本來實作。
- 其中我使用了 dynamic 的寫法，讓已經執行完手上工作的threads能立刻去拿下一個row來繼續執行，減少等待其他threads的時間，讓整個流程更utilized。

```
void* worker(void* args){
    // printf("Hello World\n");
    int k,i,j;
    for(k=0; k<n; k++){
        current_i = 0;
        while(current_i < n){
            pthread_mutex_lock(&mutex);
            // do something
            if(current_i < n){
                i = current_i;
                current_i++;
            }
            pthread_mutex_unlock(&mutex);
            for(j=0; j<n; j++){
                if(Dist[i][j] > Dist[i][k]+Dist[k][j] && (Dist[i][k]!=INF && Dist[k][j]!=INF))
                    Dist[i][j] = Dist[i][k]+Dist[k][j];
            }
            pthread_barrier_wait(&thread_barrier);
        }
        pthread_exit(NULL);
    }
    return NULL;
}
```

All-Pairs Shortest Path (Single GPU)

- 在hw3-2中我也是使用 **Blocked Floyd-Warshall** (block_FW) 來實作。

- 實際implement的做法是：

- pin 著 CPU 上的 Dist memory
- 使用 CUDA 在 GPU 上 allocate memory
- 複製 CPU 上的 Dist 到 GPU 上的 global memory 的 dst

```
cudaHostRegister(Dist, n*n*sizeof(int), cudaHostRegisterDefault);
cudaMalloc(&dst, n*n*sizeof(int));
cudaMemcpy(dst, Dist, n*n*sizeof(int), cudaMemcpyHostToDevice);
```

- 這裡pin 著CPU再做資料搬運的好處是可以加速memcpy。

- 再來是針對整個 block_FW 的流程，會由一個 for 迴圈去執行 round 次數的三個 Phase (Phase1, Phase2, Phase3)。

```
const int blocks = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 block_dim(32, 32, 1);
dim3 grid_dim(blocks, blocks, 1);

int round = ceil(n, B);
for (int r = 0; r < round; ++r) {
    Phase1<<<1, block_dim>>>(dst, r, n);
    Phase2<<<blocks, block_dim>>>(dst, r, n);
    Phase3<<<grid_dim, block_dim>>>(dst, r, n);
}
```

- Phase1 只會針對左上角的格子做一次update整個vertex weight的 matrix 數值 (以助教的範例圖為例)。所以只要準備一個block，且這個block裡面有的 threads數量是32*32的(至於為什麼是32, 後面會解釋)。
- Phase2 則是負責開始往同一層 (k) 的row和col延伸更新。所以會需要長度為blocks (也就是V/B後的結果) 的 32*32 threads數的空間。
- Phase3 則是處理剩下不等於 k 層的row和col們做update。所以會需要長度為 blocks * blocks 且每個小block中的threads數也是 32*32。

- 為什麼block_dim是設成(32, 32, 1)呢，因為對於CUDA block來說雖然每個block可能會被分配到不同的SM上，且threads的數量也可以任意，但是主要還是以32個threads為一個warp單位，所以這裡設計成32*32也是方便memory access的方便性，能夠滿足Occupancy Optimization。
- 對於三個Phase我準備了64*64的2D shared-memory讓Phase中blocks裡的threads可以同時使用，以加快存取時間。畢竟因為access GPU的shared-memory比access global memory還要來得快。
- 在我們使用的GPU GTX 1080中，它的最大threads一次最多只能開1024(32*32)個。如果我們的blocksize是64，那就會需要4096(64*64)個，很明顯不足夠，所以接下來的實作都是以32*32個threads為單位去執行，存取一樣使用64*64的shared-memory。最後每次iteration都會需要對shared-memory上的四個象限的點同時做update，亦即優化Instruction Level Parallelism(ILP)。
 - 四個象限點idx分別是(i, j), (i+32, j), (i, j+32), (i+32, j+32)等以此類推
- 同時也可以讓Coalesced memory access得以實現，回應SIMD的warp size unit。
- 首先是Phase1，第一步先把global memory的dist值複製到shared-memory上，再來展開64次的k loops去同時更新四個點的值。

```
#pragma unroll 16
for (int k = 0; k < 64; k++) {
    shared_memory[i][j] = min(shared_memory[i][j], shared_memory[i][k] + shared_memory[k][j]);
    shared_memory[i+32][j] = min(shared_memory[i+32][j], shared_memory[i+32][k] + shared_memory[k][j]);
    shared_memory[i][j+32] = min(shared_memory[i][j+32], shared_memory[i][k] + shared_memory[k][j+32]);
    shared_memory[i+32][j+32] = min(shared_memory[i+32][j+32], shared_memory[i+32][k] + shared_memory[k][j+32]);
    __syncthreads();
}
```

- 計算完後就再把目前Phase做好運算的shared_memory複製回去global memory的dist上。
- 剩下的Phase2和Phase3也是針對到對的index位置做運算即可。

All-Pairs Shortest Path (Multi GPU)

- 對於 multi GPU 的寫法其實也沒什麼太大的變動，主要是block_FW中使用OpenMP 開兩個 parallel threads (因為助教分配的設備資源就是一個server兩臺)。
 - 使用 **cudaSetDevice** 和 **cudaMalloc** 來讓threads 分配對應的 GPU。
 - 在進入round的迴圈之前，先從host 複製 yOffsetsize 長度的 Dist 資料到GPU上的global memory去運算，結束後再放回去 CPU上的memory。兩個GPU相互重複從CPU拿資料，運輸，最後再儲存回去。
 - 其餘的 Phase 內的工作都一樣。除了Phase3的block row的idx要更新，因為每次GPU拿到資料是memorybase+yOffset長度的，所以block的row idx要記得補上去即可。
-

1. Which algorithms do you choose in hw3-1?

- 在hw3-1當中，我是以 **Blocked Floyd-Warshall** 來當作整個程式的核心演算法來計算All-Pairs Shortest Path的問題。
- 我在hw3-1中使用了 **pthread + openmp** 的 **hybrid** 版本。

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

- 如前面implement中提到的，在hw3-2當中以 32×32 為threads unit去更新和計算，同時更新四個座標index的值，同時access shared-memory。
- 在hw3-3中，我讓block row為主軸，每次兩個GPU分別從CPU上拿了 $memorybase + row$ 長度的blocks資料來計算，在放回去CPU，反覆執行。

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

- blocking factor 大B 選擇 64，因為我們的資料(V/B)後有可能會少於32，那樣每次都會以小於 Warp size 的 32 去執行，在效率上會有欠缺。所以一律取大於資料量長度且32的倍數的 blocksize。
- number of blocks也已於Implement 那part中解釋了，不同的phase準備不同的blocks數量。
 - Phase1 準備一個block, block裡有 32×32 個threads。
 - Phase2 準備V/B且長度row個blocks, 每個單一小block裡也有 32×32 的threads。
 - Phase3 準備V/B - 1的平方的 grid blocks, 裡面的blocks也是 32×32 的threads。

```
const int blocks = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 block_dim(32, 32, 1);
dim3 grid_dim(blocks, blocks, 1);

int round = ceil(n, B);
for (int r = 0; r < round; ++r) {
    Phase1<<<1, block_dim>>>(dst, r, n);
    Phase2<<<blocks, block_dim>>>(dst, r, n);
    Phase3<<<grid_dim, block_dim>>>(dst, r, n);
}
```

2. Profiling Results (hw3-2)

- Profiling Results 針對 hw3-2 以下的幾個 metrics 來探討：
 - occupancy
 - sm efficiency
 - shared memory load/store throughput
 - global load/store throughput
- 使用 Nvidia profiling tools, 以 testcase c21.1 為例：

Phase1

Metrics Name	Min	Max	Avg
sm_efficiency	4.33%	4.66%	4.64%
achieved_occupancy	0.497841	0.498087	0.497993
shared_load_throughput	109.04GB/s	126.59GB/s	120.61GB/s
shared_store_throughput	35.790GB/s	37.752GB/s	37.534GB/s
gld_throughput	326.83MB/s	384.17MB/s	375.46MB/s
gst_throughput	559.31MB/s	593.27MB/s	585.42MB/s

Phase2

Metrics Name	Min	Max	Avg
sm_efficiency	85.25%	94.03%	92.44%
achieved_occupancy	0.957766	0.976177	0.963545
shared_load_throughput	1992.1GB/s	2301.6GB/s	2176.4GB/s
shared_store_throughput	1335.6GB/s	1385.3GB/s	1362.3GB/s
gld_throughput	11.622GB/s	13.556GB/s	13.118GB/s
gst_throughput	20.285GB/s	20.877GB/s	20.548GB/s

Phase3

Metrics Name	Min	Max	Avg
sm_efficiency	99.49%	99.72%	99.64%
achieved_occupancy	0.927586	0.929306	0.928539
shared_load_throughput	3154.9GB/s	3509.9GB/s	3378.7GB/s
shared_store_throughput	257.47GB/s	261.31GB/s	259.24GB/s
gld_throughput	16.470GB/s	18.777GB/s	18.586GB/s
gst_throughput	63.908GB/s	64.687GB/s	64.337GB/s

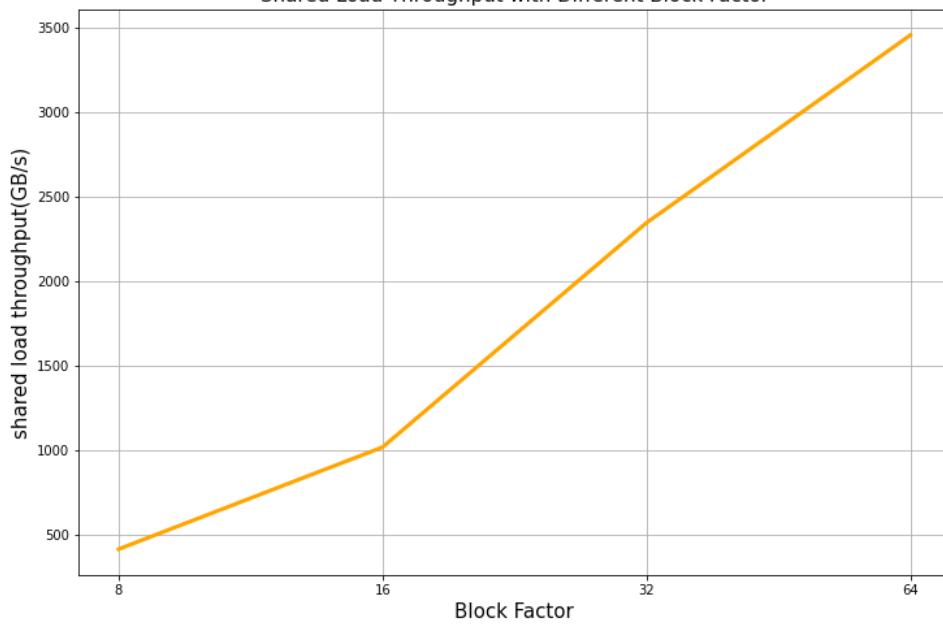
3. Experiments & Analysis

- System Spec
 - 我使用助教分配的 hades.cs.nthu.edu.tw 做為我的測試server。
- Blocking Factor
 - 以 testcase c21.1 為例。且為了方便，這裡的數值是以Phase3的average value為衡量。
 - 可以看到從圖中也顯示了Blocking Factor 在 64是最好的選擇。

Blocking Factor	8	16	32	64
shared load throughput (GB/s)	413.31	1016.2	2343.5	3454.2
shared store throughput (GB/s)	228.83	584.28	721.82	263.12
glb throughput (GB/s)	19.690	18.906	19.187	18.850
gst throughput (GB/s)	57.302	144.98	89.827	72.310
Integer Instructions	5.894e+8	8.903e+8	1.653e+9	3.003e+9
Total Time(s)	2.634	1.123	0.906	0.805

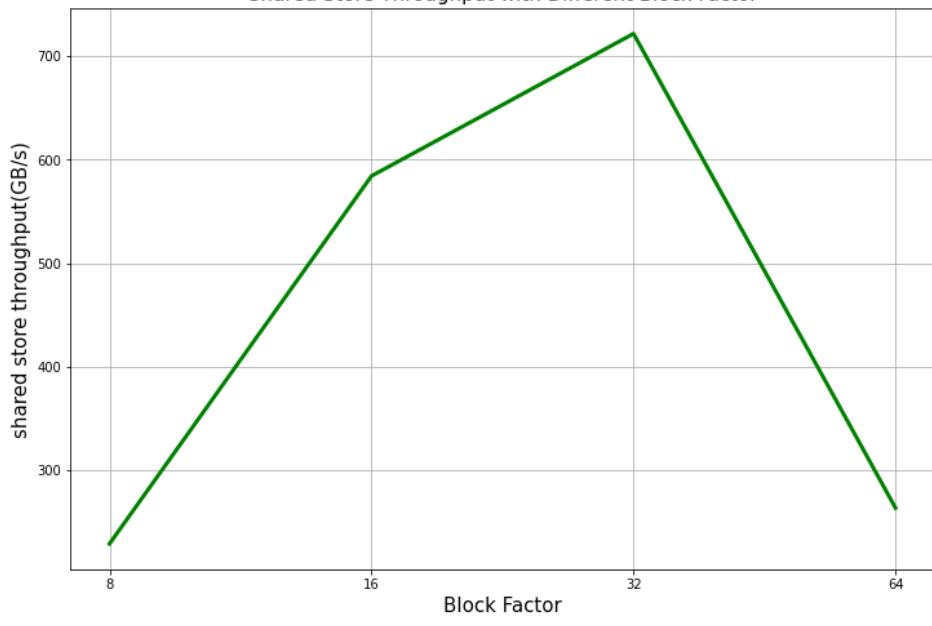
◦

Shared Load Throughput with Different Block Factor



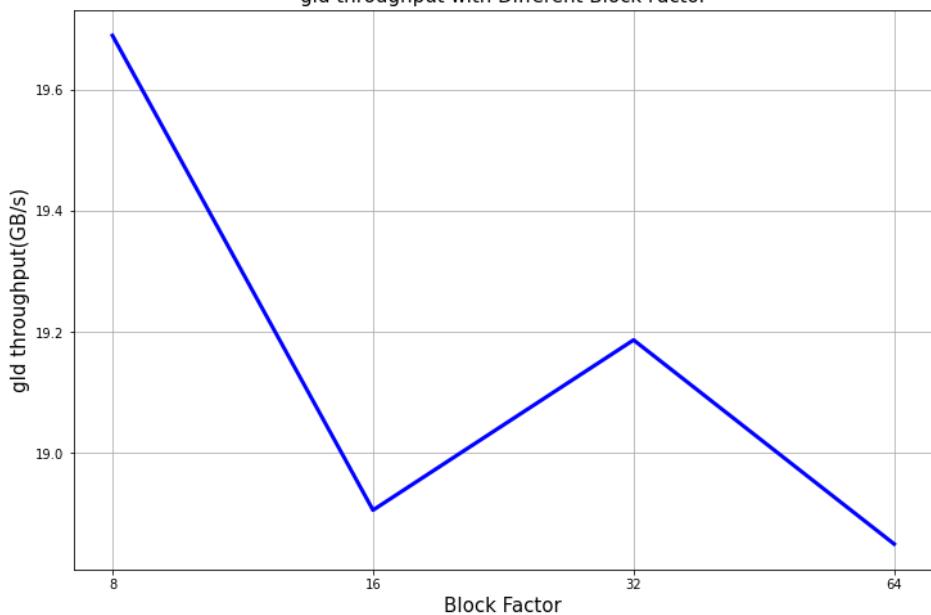
O

Shared Store Throughput with Different Block Factor



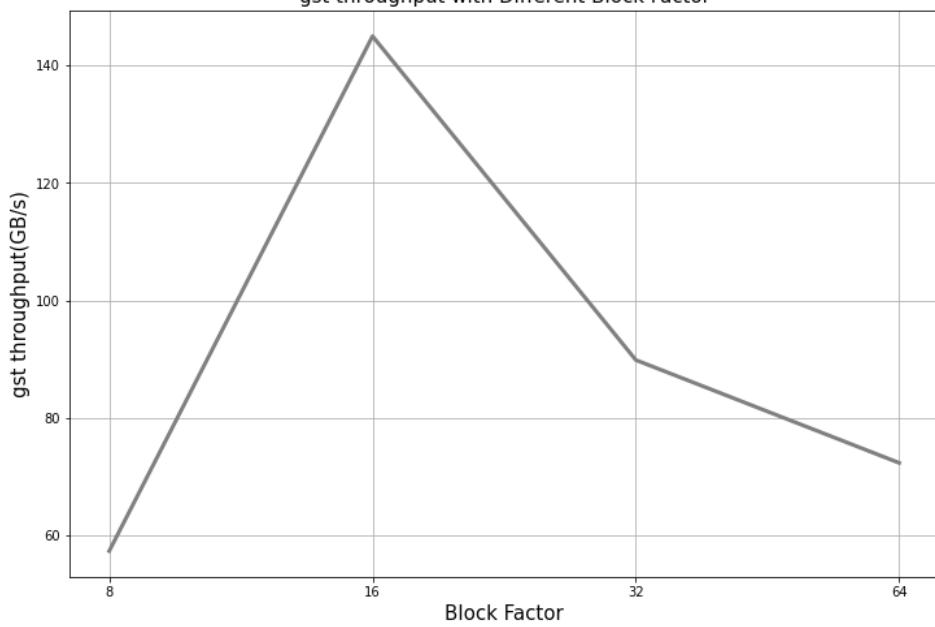
O

gld throughput with Different Block Factor

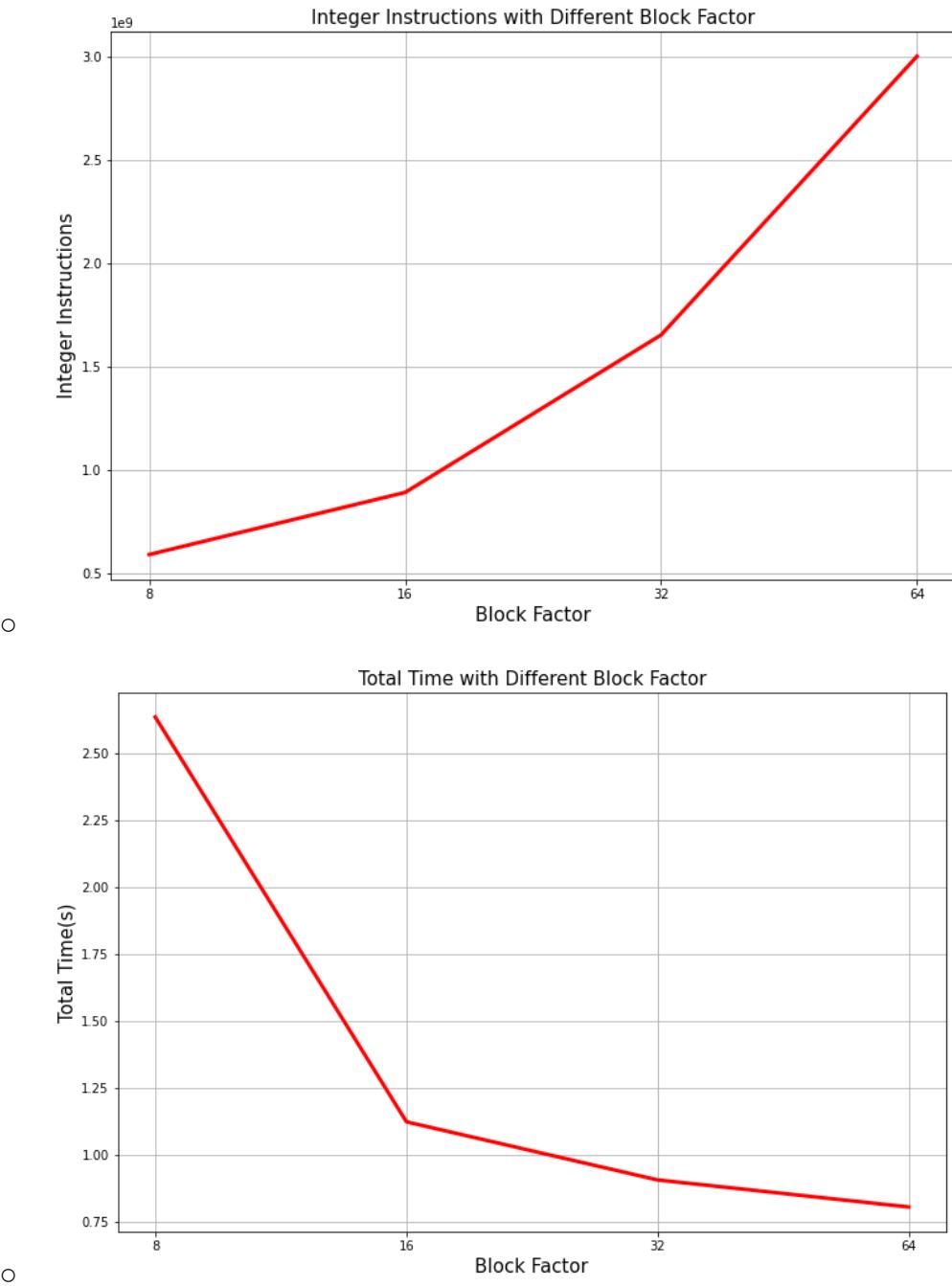


O

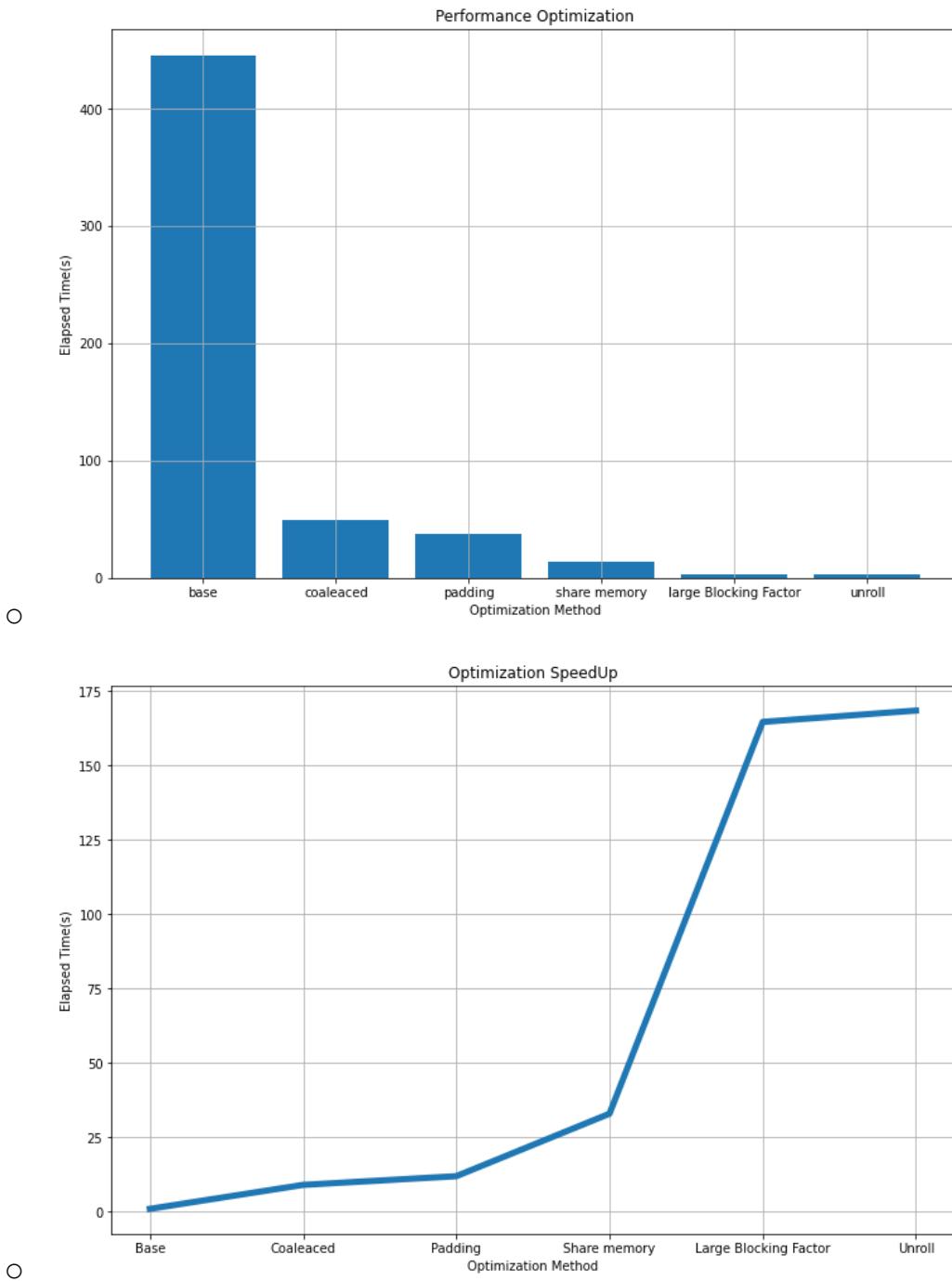
gst throughput with Different Block Factor



O



- Optimization



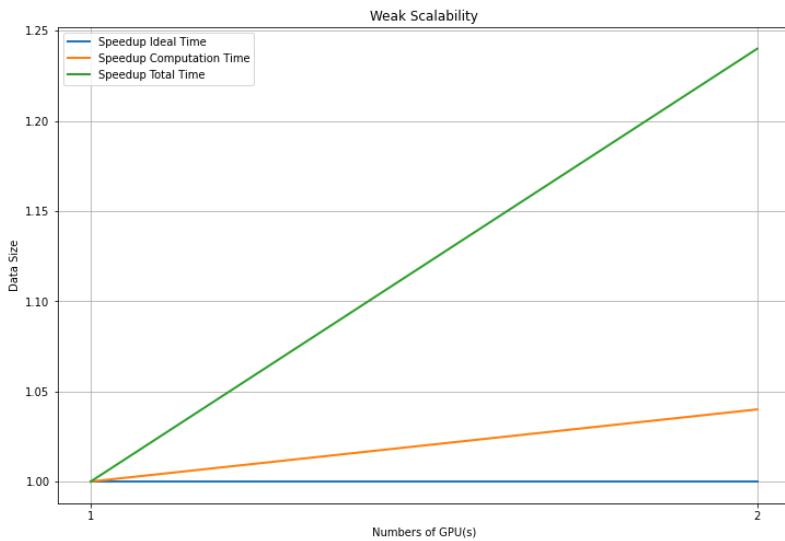
- Weak scalability

- weak scalability的用意是探討通過增加data size的情況下，也增加processors的數量的情況下，能加速多少。
- 我這裡是使用p19和p24，兩者V^3資料量差了一倍
 - p19: (original_n: 18947, padding_n: 19008)

- p24 (original_n: 24000, padding_n: 24064)
- $24064^3 / 19008^3 = 2.029$ (倍)

Weak Scalability	1GPU	2GPU
Speedup Ideal Time	1	1
Speedup Computation Time	1	1.04
Speedup Total Time	1	1.23

○



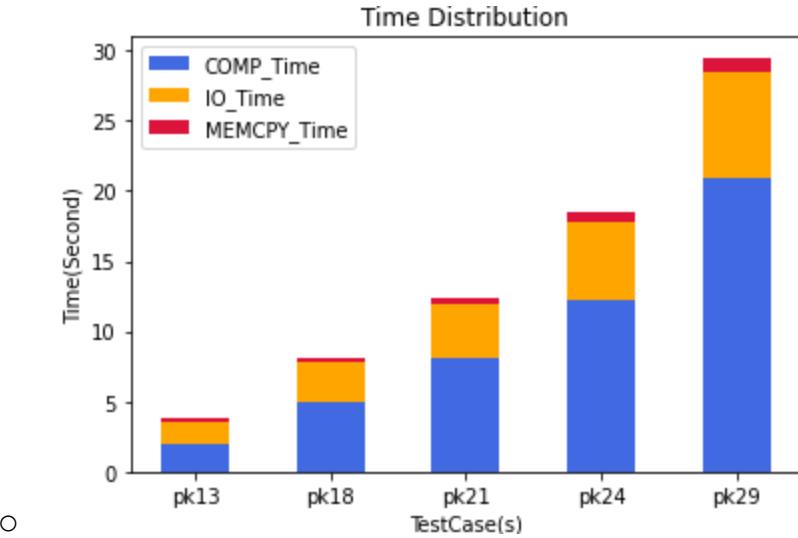
○

● Time Distribution

- 我是根據Nvidia 的 NVPROF 顯示的數據畫成圖。

Testcase	Comp_Time(μs)	IO_Time(μs)	Memcpy_Time(μs)
pk13	1.592	0.210	1.986
pk18	2.874	0.389	4.890
pk21	3.794	0.544	8.086
pk24	5.491	0.713	12.266
pk29	7.522	1.039	20.953

○



- Others

- 其他值得注意的點或者優化的心得如下：
- 盡量以完整的數字表示乘法的結果。例如在update weight的時候，我們會需要對四個index做計算，有些人會為了code style寫得比較好懂就寫成 $32*32$ 或 $64*64$ 等。直接以 $2048, 4096$ 來表示，會讓程式減少計算乘法的時間，代價是code的可讀性會降低。
- 乘除可以試看看用shift operator來代替乘除運算子。
- unroll 32 的效果比 unroll 64 更好。在 tune unroll的最佳值時，發現 unroll 32 的時間會比較短，我覺得應該是因為 unroll 後讓 64 個一起做反而會更容易出現有些還沒做完的情況，導致時間上沒有被utilized, unroll太小反而就失去 unroll 一起做的意義了，所以32是這個problem最好的值，且也符合warp的設計，更有一致性。
- 在GPU上盡量減少branch的出現，如(if, switch 等)，因為GPU注重平行化，若有越多branch的可能性，平行化的工作會變得困難且漸漸失去意義。
- 使用內建的 minimum define function 來找出較小的值會比使用三元運算子來得快。雖然我們都知道要避免branch，但是對於三元運算子我們還是可以用其他optimized的function來代替實現。
- 從上述的實驗我們不難看出memcpy的時間佔總時間最少，Computation的部分佔最多。除了這兩個可以優化以外，I/O Time也可以做優化。例如如下的方式直接用一次for迴圈來代替兩次for迴圈，做法是三個memory位置一起assign。

```

for (int block_i = 0; block_i < n; ++block_i) {
    int IN = block_i * n;
    #pragma GCC ivdep
    for (int j = 0; j < block_i; ++j) {
        Dist[IN + j] = INF;
    }
    #pragma GCC ivdep
    for (int j = block_i + 1; j < n; ++j) {
        Dist[IN + j] = INF;
    }
}

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

```



```

void output(char *outFileName) {
    FILE *outfile = fopen(outFileName, "w");
    for (int block_i = 0; block_i < N; ++block_i) {
        fwrite(&Dist[block_i * n], sizeof(int), N, outfile);
    }
    fclose(outfile);
}

```

- 除了code的寫法以外，選對compiler也是很重要，對於這個case來說有人說c++ 的 vector 在這會有相當不錯的效果。這裡使用g++ 會比 gcc 來得適合。
- Command
 - srun -p prof -N1 -n1 --gres=gpu:1 nvprof --metrics [metrics] ./hw3-2
`./cases/c21.1 ./cases/c21.1_myout.out`
 - sm_efficiency
 - achieved_occupancy
 - gld_throughput
 - gst_throughput
 - shared_load_throughput
 - shared_store_throughput
 - inst_integer
 -

4. Experience & Conclusion

-
- 這次學到了怎麼在GPU上做 data 和 memory 的分配，且也學到了 warp 和 blocks 和 GPU 一次 block 的 threads 上限等硬體限制與設計架構。

- 由此可見若有足夠清楚的硬體背景了解，再從軟體方面設計好的記憶體分配分流是可以大大優化整個流程的。
- 這次的實驗總結出的 bottleneck 很明顯是出在 memory 的 access上，接下來若有機會的話，可以往這個方向繼續做優化應該會是最有效的。
- 有些優化方法其實之前根本不會遇到，如compiler的flag, pragma unroll, CUDA API等，主要也是看了蠻多人的寫法，都一一認識，拿來嘗試看看，發現原來還有這種好東西，就直接拿進來code使用了xD。
- 這次的作業難度其實不低，實作起來雖然步驟不多，但是因為是在GPU上要 global memory的空間大小和資料的記憶體搬運才是關鍵(例如我就是空間沒有開好，有時候會跑到奇怪的地方，或者少掉一些值，且這個作業又是 iteration蠻多次的，trace debug起來真的不是開玩笑xD)
- hw3-3的部分，雖然我的寫法是丟回CPU，再讓準備好的GPU threads去跟CPU要，但是這樣的寫法聽起來就有點大費周章，如果可以全部平均都分配到所有GPU，且做GPU間的peer to peer交換也許會更快。