Implementation (seq-flashattention.c)

hw4_flash.cu

Describe how you implemented the FlashAttention forward pass using CUDA. Mention the algorithm's key steps, such as matrix blocking, SRAM usage, and how intermediate results like scaling factors (ℓ and m) were calculated.

- 多 Batch 一次性處理
  - 一次性把所有 BNd 的 Q, K, V, O 拷貝到 GPU (以及為 batch l[]、m[] 分配空間)，
  - GPU 裏面用一個函式 (flash_attention_all_batches) 把每個 batch 的注意力運算全部做完
  - 最後再把O一次性拷回 Host。

```
    // 3) 一次性拷 Q, K, V, O
    cudaMemcpy(d_Q, Q, B*N*d*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_K, K, B*N*d*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_V, V, B*N*d*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_O, O, B*N*d*sizeof(float), cudaMemcpyHostToDevice);
    // 4) 執行
    flash_attention_all_batches(br, bc);
    // 5) 把 O 拷回 Host
    cudaMemcpy(O, d_O, B*N*d*sizeof(float), cudaMemcpyDeviceToHost);
    // 6) 輸出
    output(argv[2]);
```

- SRAM usage & matrix blocking
  - QKDotAndScalarKernel 中使用tiled matrix multiplication技巧對d做分段（tilePos += 16），在 shared memory 中一次載入 16 個特徵維度，然後 partial dot product

```
+----------------------------+
|        QKDotAndScalarKernel        |
| Grid: (number of iBlock, jBlock)  |
| Block: (TILE_SIZE, TILE_SIZE)      |
+----------------------------+
            |
            v
+-------------------------------+
| 每個 Thread 負責一個 (i, j)    |
+-------------------------------+
            |
            v
+---------------------------------------+
| 1. 載入 Q 與 K 的 tile 到 Shared Memory |
|    - sQ[TILE_SIZE][TILE_SIZE]          |
|    - sK[TILE_SIZE][TILE_SIZE]          |
+---------------------------------------+
            |
            v
+-------------------------------+
| 2. 計算部分 Dot Product       |
|    - for(t = 0; t < TILE_SIZE; t++) |
|      sum += sQ[tidx][t] * sK[tidy][t] |
+-------------------------------+
            |
            v
+-------------------------------+
| 3. 總和與乘上 scalar          |
|    - out[i * bc + j] = sum * scalar  |
+-------------------------------+

```

- ℓ and m 計算
- 初始化 ℓ 和 m
  - initLMForBatch　kernel在每個 batch 的每一行開始計算前，將 ℓ 初始化為 0，m 初始化為 -∞

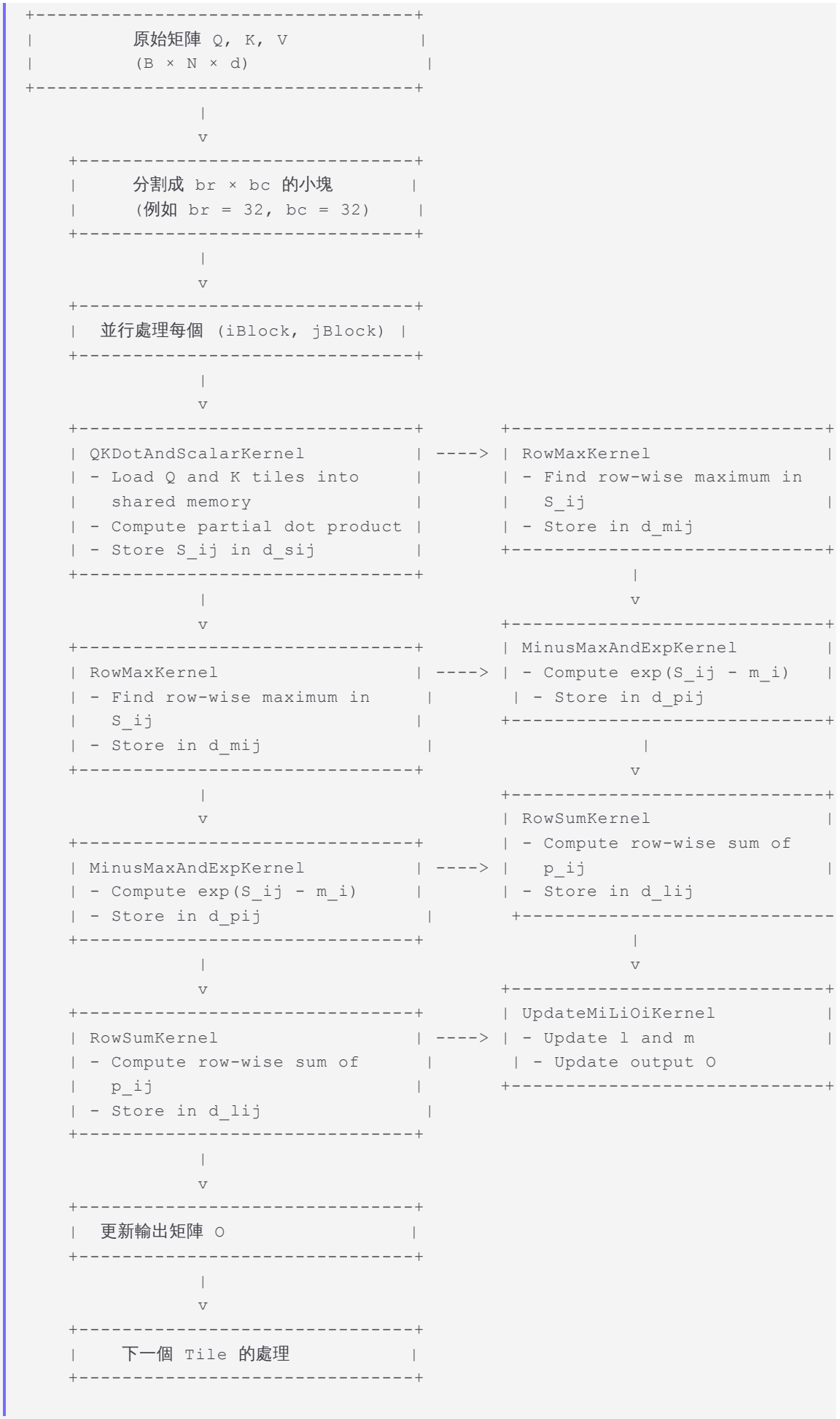
```
__global__ void initLMForBatch(float *d_l, float *d_m, int bIdx, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x; // [0..N)
    if(i < N){
        int offset = bIdx*N + i;
        d_l[offset] = 0.0f;     // l=0
        d_m[offset] = -FLT_MAX; // m = -∞
    }
}
```

- 分塊計算與 ℓ、m 的更新
  - 整個 QK^T矩陣被分割成若干個小區塊（tiles），每個區塊的大小為 br×bc（例如 32×32）。對於每個區塊，執行以下步驟：

1. 計算區塊內的注意力得分 Sij `QKDotAndScalarKernel<<<grid, block>>>(d_sij, d_Q, d_K, bIdx, iBlock, jBlock, br, bc, N, d, scale);`
2. 找出區塊內每一行的最大值 mij `RowMaxKernel<<<br,1>>>(d_mij, d_sij, br, bc);`
3. 計算exp(Sij−mi) `MinusMaxAndExpKernel<<<grid, block>>>( d_pij, d_sij, d_mij, br, bc);`
4. 計算區塊內每一行的累加和lij `RowSumKernel<<<br,1>>>(d_lij, d_pij, br, bc);`
5. 更新全局的 ℓ 和 m並更新輸出O `UpdateMiLiOiKernel<<<grid, block>>>(d_m, d_l, d_O, d_pij, d_mij, d_lij, d_V, bIdx, iBlock, jBlock, br, bc, N, d);`

Explain how matrices Q, K, and V are divided into blocks and processed in parallel.

1. divided into blocks：
   將 Q 和 K 劃分為 br × bc 的塊。
   每個 CUDA Block 處理一個 Q 和 K 的塊對（iBlock, jBlock）。
2. load into Shared Memory：
   每個 Thread 負責載入 TILE_SIZE × TILE_SIZE 的子塊 sQ 和 sK 到共享記憶體，以實現高效的點積計算。sQ 和 sK 是共享記憶體中的暫存區，用於存放當前塊的 Q 和 K 的子塊。
   點積計算：每個 Thread 在共享記憶體中計算部分點積，然後累加到總和 sum 中。經過多次載入子塊 sQ 和 sK，完成整個點積的計算。
3. 寫回結果：將計算結果 sum 乘以縮放因子 scalar，然後寫回全域記憶體中的輸出矩陣 d_sij。


原始矩陣 Q, K, V
(B × N × d)
分割成 br × bc 的小塊
(例如 br = 32, bc = 32)
並行處理每個 (iBlock, jBlock)
QKDotAndScalarKernel
- Load Q and K tiles into shared memory
- Compute partial dot product
- Store S_ij in d_sij
RowMaxKernel
- Find row-wise maximum in S_ij
- Store in d_mij
MinusMaxAndExpKernel
- Compute exp(S_ij - m_i)
- Store in d_pij
RowSumKernel
- Compute row-wise sum of p_ij
- Store in d_lij
更新輸出矩陣 O
下一個 Tile 的處理
RowMaxKernel
- Find row-wise maximum in S_ij
- Store in d_mij
MinusMaxAndExpKernel
- Compute exp(S_ij - m_i)
- Store in d_pij
RowSumKernel
- Compute row-wise sum of p_ij
- Store in d_lij
UpdateMiLiOiKernel
- Update l and m
- Update output O

Describe how you chose the block sizes B_r and B_c and why.

我選擇 block size Br=32 Bc=32以下是我的理由

- 減少 Global Memory 讀寫次數、提升平行度
  - 一次處理 32 條 row (或 column) 可以讓 GPU kernel 在一個區塊 (iBlock, jBlock) 裡做足夠多的運算，減少 kernel 呼叫的 overhead。
  - 同時 32 通常與 warp size (32 threads) 有對齊的優勢，對 row-based kernel 有助於 coalescing。
- 配合後續 Shared Memory Tiling (TILE_SIZE=16)：
  - 在 QKDotAndScalarKernel 裏面，我們又進一步將每次要用的 d-dimension(特徵維度)分成 tiles 大小 = 16。
  - 這樣 32 x 32 的 S 區塊，在 kernel 內部就會用 16 x 16 的 tile 兩次 (因為 32 ÷ 16 = 2) 掃過 row / column。
- 考量 Shared Memory 容量：
  - QKDotAndScalarKernel 在每次 16 維特徵要載入 shared memory 的量為 16×16 16×16（對 Q 及 K 各需要一塊），共 512 個 float（約 2KB），對大多數 GPU 而言相當輕鬆。
    若將 tile 設太大，可能導致 shared memory 不足或是 occupancy(同時執行的 block 數量) 降低。若設太小，則 kernel 的呼叫數量又會變多，效率也會下降。

Specify the configurations for CUDA kernel launches, such as the number of threads per block, shared memory allocation, and grid dimensions.

1. QKDotAndScalarKernel<<<grid, block>>>
   - dim3 block(TILE_SIZE, TILE_SIZE) = (16, 16)。每個 block 共有 256 threads
   - dim3 grid((br+TILE_SIZE-1)/TILE_SIZE, (bc+TILE_SIZE-1)/TILE_SIZE) = (⌈32/16⌉, ⌈32/16⌉)=(2,2)，因為一次要覆蓋 32 x 32 的 S 區塊，故需要 2 x 2 = 4 個 block 來並行處理。
   - Shared memory: 需要的空間主要是 sQ[16][16] 與 sK[16][16]，共 512 個 float，外加一點額外暫存 (幾乎可以忽略不計)。
2. RowMaxKernel<<<br,1>>>
   - 一個 block 負責一個 row。這裡 br=32 => gridDim=32，blockDim=1。
     每個 row 會在該 block 的單一 thread 裏面直接做一個 for-loop 找最大值
3. MinusMaxAndExpKernel<<<grid, block>>>

- dim3 block(8,8)，dim3 grid((br+7)/8, (bc+7)/8)。這個 kernel 需要同時遍歷整個 br x bc 矩陣，因此用 2D block (8,8)。每個 block 有 64 threads，(br+7)/8=4 => grid 可能是 (4,4) = 16 blocks；16 blocks x 64 threads = 1024 threads。

4. RowSumKernel<<<br,1>>>
   - 與 RowMaxKernel 一樣做法：一個 block 負責一整 row 的加總。

5. UpdateMiLiOiKernel<<<grid, block>>>
   - dim3 block(32)、dim3 grid((br+31)/32) = 1 (因為 br=32)。
     即 32 threads 負責 32 個 row 的更新，同時在 kernel 裏面再用一個 for 迴圈把 pij * V 累積到 O。

Justify your choices and how they relate to the blocking factors and the SRAM size.

- Tiling 大小設為 16，可保證在大多數 GPU 上能同時容納多個 block 執行，而不會過度佔用 shared memory。16 x 16 x 4 bytes x 2（Q 與 K） ≈ 2 KB，對支援 48KB~64KB shared memory 的 SM 來說負擔很小。這樣就能增加 occupancy（同時執行的 block/warp 數量），提升整體吞吐量。

- br=32,bc=32 RowMaxKernel, RowSumKernel 這類對 br=32 行做操作的 kernel，可以輕鬆地用 32 threads (或 32 blocks with 1 thread) 來平行化。32 跟 warp size 同步，若要改成一行對應一個 warp，也容易做對齊，不會有 thread 閒置的問題。若把 br, bc 設得太大，如 64, 128，則單次需要用到的 shared memory 就會變多，甚至要搭配更複雜的 tiling 設計。同時 block 數量減少，也有可能降低 GPU 核心的使用率。

- 以 tile (16) 的寬度做讀寫，可以保證同一 row 或同一列上的連續 thread 會讀取相鄰記憶體位置，減少對 global memory 的無效率存取。

- 示意圖

```
整體 NxN (舉例 N=8, br=4, bc=4)

   iBlock=0           iBlock=1
+---------+---------+
|  (0,0)  |  (0,1)  |   <- jBlock=0,1
+---------+---------+
|  (1,0)  |  (1,1)  |
+---------+---------+

每一塊(4x4)內部 (QKDot) 又用 tile(16維) 在 d維度上做部分 dot product
```

Implementation (seq-attention.c)

hw4_seq.cu

kernel配置

- QKDotAndScalarKernel<<<blocksPerGrid, threadsPerBlock>>>
  - threadsPerBlock = 1024，blockDim.x = 1024。
  - 每個 block 內有 32 個 warp (WARPS_PER_BLOCK = 32)。
    blocksPerGrid = (N * N + WARPS_PER_BLOCK - 1) / WARPS_PER_BLOCK， 使得每個 warp 負責一個 (i,j) 的位置。
  - Shared memory `__shared__ typename WarpReduce::TempStorage warp_temp_storage[32]`
- SoftMaxKernel<<<N, 256>>>
  - Grid 維度：gridDim.x = N，表示每個 block 負責一個 row。
  - Block 維度：blockDim.x = 256，即每個 row 用 256 個 thread 做並行。
  - Shared memory: `__shared__ float sdata[256]; __shared__ float row_max, row_sum;`
- MulAttVKernel<<<blocksPerGridMV, threadsPerBlockMV>>>
  - threadsPerBlockMV = 256。
  - blocksPerGridMV = (N * d + threadsPerBlockMV - 1) / threadsPerBlockMV。
  - 每個 thread 負責計算一個 (i,j) 輸出，對應要對一整 row (i) 的 attention row 乘以 V 在第 j 維的向量做內積。

warp_reduce 讓整體效能大幅提升的關鍵

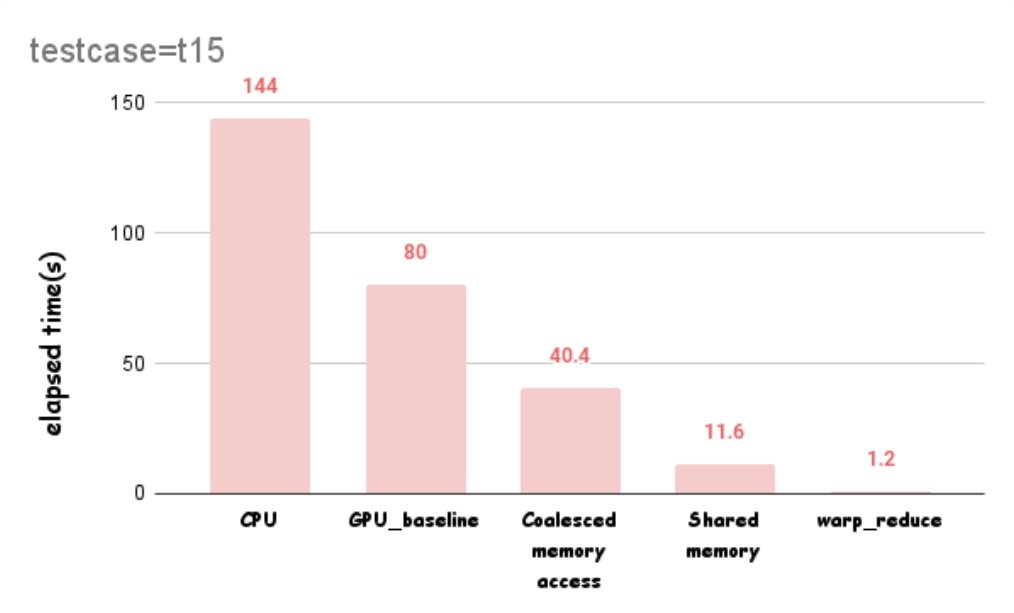
- SoftMaxKernel
  - 每個thread都會在自己分配到的元素中計算得出局部最大local_max以及總和local_sum，接著透過share memory sdata[]進行tree-reduction，在這個過程中，會比較所有thread在自己區域計算出來的local_max從中挑選出最大的存在sdata[0]，以及將所有local_sum加起來成為row_sum存在sdata[0]，最後再將d_att元素/row_sum 完成計算
  - Warp-level reduce 將整個 warp 的 Thread 作為一個單位進行同步和 reduce 操作，減少了跨 Thread Block 的同步需求，降低了共享記憶體的競爭。
  - 利用 warp 的內部同步特性（即同一 warp 的 Thread 是同步執行的），避免了全局同步的延遲，提升了 reduce 操作的效率。
- QKDotAndScalarKernel
- 每個warp中的每個thread負責分擔一部份的dot product因此每個warp會有自己計算的局部sum，透過WarpReduce將所有warp的計算出來的局部sum加總在一起，並由thread 0 將值寫回global memory中。

Profiling Results

```
==1083929== Profiling application: ./hw4 /home/pp24/pp24s036/hw4/testcases/t05 t05.out
==1083929== Profiling result:
==1083929== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: MulAttVKernel(float*, float const *, float const *, int, int)
         80                        achieved_occupancy                     Achieved Occupancy    0.396673    0.401983    0.399924
         80                            sm_efficiency                Multiprocessor Activity      76.62%      81.58%      79.57%
         80                    shared_load_throughput           Shared Memory Load Throughput  0.00000B/s  0.00000B/s  0.00000B/s
         80                   shared_store_throughput           Shared Memory Store Throughput  0.00000B/s  0.00000B/s  0.00000B/s
         80                            gld_throughput                    Global Load Throughput  685.15GB/s  768.81GB/s  736.03GB/s
         80                            gst_throughput                   Global Store Throughput  1.0705GB/s  1.2013GB/s  1.1500GB/s
    Kernel: SoftMaxKernel(float*, int)
         80                        achieved_occupancy                     Achieved Occupancy    0.879829    0.886809    0.883280
         80                            sm_efficiency                Multiprocessor Activity      85.33%      87.37%      86.38%
         80                    shared_load_throughput           Shared Memory Load Throughput  194.46GB/s  200.45GB/s  197.04GB/s
         80                   shared_store_throughput           Shared Memory Store Throughput  123.75GB/s  127.56GB/s  125.39GB/s
         80                            gld_throughput                    Global Load Throughput  141.43GB/s  145.78GB/s  143.31GB/s
         80                            gst_throughput                   Global Store Throughput  94.286GB/s  97.190GB/s  95.537GB/s
    Kernel: QKDotAndScalarKernel(float*, float const *, float const *, float, int, int)
         80                        achieved_occupancy                     Achieved Occupancy    0.771443    0.776553    0.774516
         80                            sm_efficiency                Multiprocessor Activity      98.64%      98.88%      98.81%
         80                    shared_load_throughput           Shared Memory Load Throughput  0.00000B/s  0.00000B/s  0.00000B/s
         80                   shared_store_throughput           Shared Memory Store Throughput  0.00000B/s  0.00000B/s  0.00000B/s
         80                            gld_throughput                    Global Load Throughput  243.09GB/s  244.69GB/s  243.63GB/s
         80                            gst_throughput                   Global Store Throughput  30.387GB/s  30.586GB/s  30.454GB/s
```

由圖中可以發現由於SoftMaxKernel,QKDotAndScalarKernel中我有實作warp_reduce以及tree-reduction因此這使得他們資源利用率均比其他kernel來的高，achieved_occupancy和sm_efficiency均高，此外SoftMaxKernel我有使用share memory因此可以發現他的shared memory load/store throughput均比其他kernel來的高。

Experiment & Analysis



程式使用share memory和warp_reduce能有比較大的效能優化

Experience & conclusion

- What have you learned from this homework?

- 了解到SRAM對於效能提升的重要性，掌握了將大型矩陣劃分為小區塊（tiles）進行計算的方法，學會warp-level reduce 技術