Homework 4: FlashAttention Report

1. Implementation

1.a 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.

1. 在 GPU memory 中分配
   - Q、K、V：大小為 $B \times N \times d$
   - m、l：大小為 $B \times N$

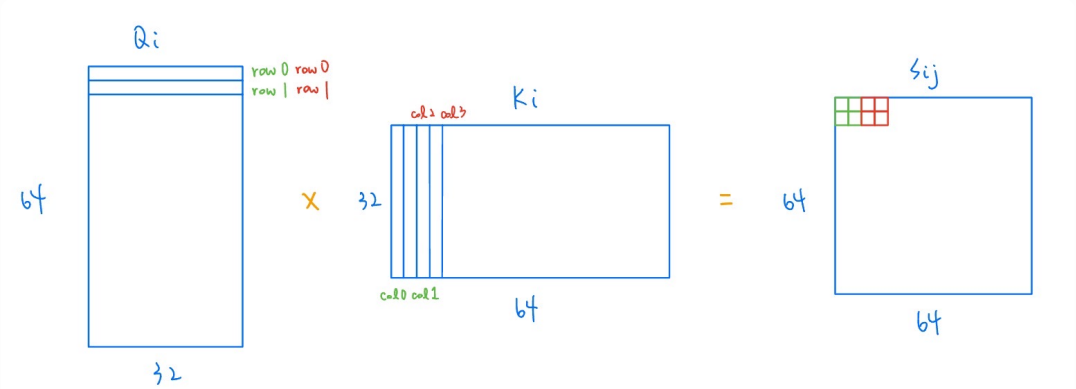
2. 我的程式中只有一個 kernel 且只 launch 一次即完成整個 attention 的計算，這邊先說明我是如何將整個 attention 的計算 mapping 到各個 block 的：
   - 將 Q、O 切成 $B_r \times d$ 的 row block Q_i、O_i
   - 將 K、V 切成 $B_c \times d$ 的 column block K_j、V_j
   - 這邊的"切"指的是每次處理的塊大小，不是真的有"切"的動作
   - 一個 block 對應到一個 Q_i，並在 kernel 中遍歷所有的 K_j、V_j，更新對應的 O_i

3. 為了 mapping 到整個矩陣範圍，我的 grid dimension 如下：

   ```
   dim3 gridDim(B*tr, 1);
   ```

   其中，B 就是 batch size，而 $tr = N / B_r$，表示 Q 跟 O 會被切成幾個 row block，透過這樣的設置，每個 block 對應到一個 row block，即可覆蓋完整的矩陣運算。

4. 再來，說明一下一個 block 中的處理流程：(我的 B_r 設為 64，B_c 也設為 64)
   - 載入 $64 \times d$ 的 row block Q_i 到 shared memory
   - for 迴圈遍歷所有的 column block，每個迴圈中流程如下：
     - 載入 $64 \times d$ 的 column block K_j 到 shared memory
     - 我的 block dimension 是 $(32, 32)$，也就是一個 block 有 32×32 個 threads，而我的 B_r 設為 64，B_c 也設為 64，所以，一個 thread 需要處理 Q_i 中的 2 個 row 與 K_j 中的 2 個 column，並對應到 S_{ij} 與 P_{ij} 中一個 2x2 的區塊，如下圖所示：



- thread (0, 0) 對應到 Q_i 的 $row0$、$row1$，以及 K_j 的 $col0$、$col1$，並計算
  - $S_{00} = row0 \cdot col0$
  - $S_{01} = row0 \cdot col1$
  - $S_{10} = row1 \cdot col0$
  - $S_{01} = row0 \cdot col1$
- thread (0, 1) 對應到 Q_i 的 row0、row1，以及 K_j 的 col2、col3，並計算
  - $S_{02} = row0 \cdot col2$
  - $S_{03} = row0 \cdot col3$
  - $S_{12} = row1 \cdot col2$
  - $S_{13} = row1 \cdot col3$
- 以此類推，整個 block 會完成 64×64 的 $S_{ij} = Q_i K_j^T \times \frac{1}{\sqrt{d}}$ 運算

- 在 flash attention 中，需要計算局部的 row max m_{ij} 跟 row sum l_{ij} 來逐步更新成完整的 softmax 結果，因此，block 內的 threads 會先針對自己負責的 2x2 S_{ij} 做局部的 `fmaxf`；接著再使用 `warpReduceMax`，進行 warp 級別的最大值運算，找出該 thread 負責的 2 個 row 中個別的最大值。

```
float row_val0 = fmaxf(val_sij_00, val_sij_01);
float row_max0 = warpReduceMax(row_val0);

float row_val1 = fmaxf(val_sij_10, val_sij_11);
float row_max1 = warpReduceMax(row_val1);
```

其中，`warpReduceMax` 就是在一個 warp（32 threads）內，進行「找最大值」的運算。由於在同一個 warp 裏面，可以透過 CUDA 的特殊指令 `__shfl_down_sync` 進行 threads 之間的資料傳遞，而不需要透過 global memory，因此效率較高。

實際作法是：

1. 每個 thread 先有一個初始值（例如某個計算出來的 S_{ij}）。

2. 接著，程式內用迴圈不斷把「相鄰距離」(offset) 變成 16、8、4、2、1，並在每一次迭代時，用 `__shfl_sync(...)` 將另一個 thread 的值取回來，和自己手上的值比較，更新出最大的數字。
3. 最後，32 個 threads 就會 reduce 出最大值。

```
__device__ float warpReduceMax(float val) {
    #pragma unroll
    for (int offset = 16; offset > 0; offset >>= 1) {
        float other = __shfl_down_sync(0xffffffff, val, offset);
        val = fmaxf(val, other);
    }
    return val;
}
```

- reduce 完後，該 warp 中只會有第 0 個 thread 有值，所以，需要廣播給該 warp 中所有的 threads。

```
float warp_max0 = __shfl_sync(0xffffffff, row_max0, 0);
float warp_max1 = __shfl_sync(0xffffffff, row_max1, 0);
```

- 有了 row-wise 最大值之後，每個 thread 就能對自己負責的四個 data 計算 $P_{ij} = \exp(S_{ij} - max_i))$ 以防止數值 overflow。

- 計算 row sum l_{ij}，block 內的 threads 會先針對自己負責的 2x2 P_{ij} 做局部的加法；接著再使用 `warpReduceSum`，進行 warp 級別的 summation 運算，找出該 thread 負責的 2 個 row 中個別的總和，然後廣播給該 warp 中所有的 threads。

- 將「舊有的 partial sum / partial max」與「本次 column block 貢獻的值」整合，也就是更新 m 跟 l，這樣才能維持 softmax 結果的正確性，這邊因為我是將 m 跟 l 存在 register 中，而不是 shared memory，所以每個 thread 都要更新自己負責的那兩個 row 的 m 跟 l，令當前這個 column block 針對該 row 產生的「局部最大值」為 $warp\_max$，局部總和為 $warp\_sum$。公式如下：

$$m_i^{new} = max(m_i^{old}, warp\_max)$$

$$l_i^{new} = e^{m_i^{old} - m_i^{new}} \times l_i^{old} + e^{warp\_max - m_i^{new}} \times warp\_sum$$

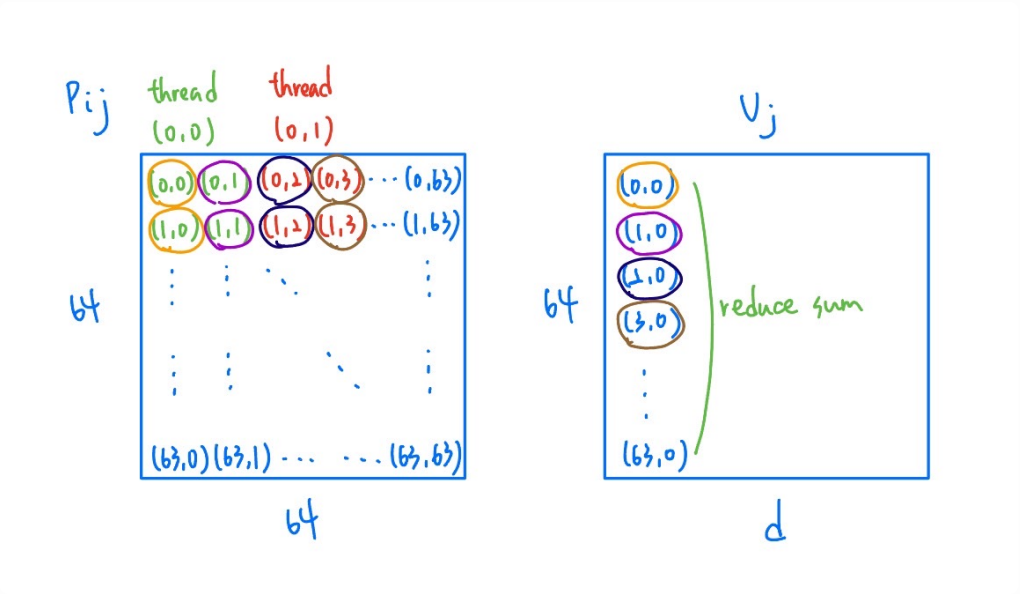
- 將 P_{ij} 與 V_j 相乘，並更新到 O_i，這邊的 mapping 方式比較複雜，以下圖為例，64×64 的 P_{ij} 與 $64 \times d$ 的 V_j 相乘，為了得到 O_{00}，必須將 P_{ij} 的 row0 與 V_j 的 column0 做內積，但是，P_{ij} 的 data 是分別存在每個 thread 的 registers 中，沒辦法由一個 thread 來處理 O_{00} 的計算，所以需要每個 thread 計算一部分的 partial sum，再將 32 個 threads 計算出的 sum reduce 成 O_{00} 的值。以 thread (0, 0) 為例，

thread (0, 0) 有 P_{00}、P_{01}、P_{10}、P_{11} 的值，所以 thread (0, 0) 需要負責 O_i 的 row0 及 row1 的 partial sum，因為計算這兩個 row 都需要用到 thread (0, 0) 有的資料。具體來說，第一個 iteration，會做以下計算：

- thread (0, 0) : $P_{00} \times V_{00} + P_{01} \times V_{10}$
- thread (0, 1) : $P_{02} \times V_{20} + P_{03} \times V_{30}$
- ...
- thread (0, 31) : $P_{0,62} \times V_{62,0} + P_{0,63} \times V_{63,0}$
- 這樣一個 warp 的 threads 就有 O_{00} 所需要的 partial sum 了 (令其為 partial_sum)，再用 `warpReduceSum` 取得全部的總和，再用以下公式：

$$O_i^{new} = \frac{e^{(m_i^{old} - m_i^{new})} \times l_i^{old} \times O_i^{old} + e^{(warp\_max - m_i^{new})} \sum_j (P_{ij} \times V_j)}{l_i^{new}}$$

- 以上是計算出 O_{00} 的方式，而第一個 iteration 會計算出 O_i 的第一個 column，然後 iteration 會沿著 d 方向進行，所以會逐步計算出 O_i 的每個 column，進而更新完整個 O_i。



- 這樣一個 column block 的 iteration 就結束了，再來就是要遍歷完所有的 column，注意每個 iteration 之間有 所以在迴圈的最後要使用 `__syncthreads()` 確保該 block 的所有 thread 都已完成該迴圈，然後等每個 block 都完成運算整個 attention 的 forward pass 就計算完成了。

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

這邊 1.a 應該有提到了，再把重點附上：

- 我的程式中只有一個 kernel 且只 launch 一次即完成整個 attention 的計算，這邊先說明我是如何將整個 attention 的計算 mapping 到各個 block 的：
  - 將 Q、O 切成 $B_r \times d$ 的 row block Q_i、O_i
  - 將 K、V 切成 $B_c \times d$ 的 column block K_j、V_j
  - 一個 block 對應到一個 Q_i，並在 kernel 中遍歷所有的 K_j、V_j，更新對應的 O_i
- 為了 mapping 到整個矩陣範圍，我的 grid dimension 如下：

  ```
  dim3 gridDim(B*tr, 1);
  ```

  其中，B 就是 batch size，而 $tr = N / B_r$，表示 Q 跟 O 會被切成幾個 row block，透過這樣的設置，每個 block 對應到一個 row block，即可覆蓋完整的矩陣運算。

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

一開始我選擇 $(B_r, B_c) = (32, 32)$，因為：

- 我的 block dimension 也是 $(32, 32)$，這樣一個 thread 剛好負責 S_{ij} 與 P_{ij} 的一個元素，較好實作。

後來我將 (BR, BC) 擴大為 $(64, 64)$，因為：

- 我想實驗看看，如果將一個 block 對應到 (64×64) 的區塊，讓一個 thread 處理 S_{ij} 與 P_{ij} 中一個 2x2 的區域（4 個元素），會不會比較快，當時我實作的版本沒有用到 shared memory，所以加大 (BR, BC) 並不會對 shared memory 有影響，但會增加到使用的 registers 數量，這樣的 trade-off 是，可能一個 block 能同時執行的 block 數會減少，而且一個 thread 的工作量變四倍，但可以減少整體 block 數量，也就是沿著 column block 的 iteration 次數會減少一半，實際測試後發現幾乎在所有測資上都變快了，所以後來就選用 $(64, 64)$ 作為 (BR, BC)。
- 另外，我後來也有實驗，在有使用 shared memory 的情況下（將 Q_i、K_j load 到 shared memory），將 (BR, BC) 設為 $(64, 64)$ 也會比將 (BR, BC) 設為 $(32, 32)$ 還快，但是，將 (BR, BC) 設為 $(64, 64)$ 最多只能 load Q、K、V、O 矩陣的其中三個，因為在測資中，d 可能為 32 或 64，若為 64，三個 $(64, 64)$ 矩陣會剛好填滿一個 block 最多能用的 shared memory 空間，而且這樣會有 bank conflict 的問題，也無法透過 $(64, 64 + 1)$ 來緩解 bank conflict，shared memory 會爆掉，所以我最後只有將 Q、K load 到 shared memory，並用 $(64, 64 + 1)$ 來緩解 bank conflict。
- 會走上這條路是因為我先嘗試了將 (BR, BC) 擴大為 $(64, 64)$，才加入 shared memory 的優化，不然我覺得將 (BR, BC) 設為 $(32, 32)$，然後將 Q、K、V、O 全部載入 shared memory 可能是更好的做法，後續也還有更多優化空間，比如在計算 $P_{ij} \times V_j$ 時，如果 V_j 是從 global memory 讀取的，我目前的讀取 pattern 是 non coalesced 的，若 V_j 能載入 shared memory 就不會有這個問題了。

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

1. **Threads per block**：
   使用 $(32, 32)$ 的 block dimension，總共 1024 個 threads。

2. **Grid dimensions (gridDim)**：
   在這個作業的 Input specification 中：

   $$N \in \{128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768\}$$

   $N / 64$ 一定會整除，因此 tr (幾個 row block) 與 tc (幾個 column block) 為：

   - $tr = N / 64$
   - $tc = N / 64$

   我將 gridDim 設為 $(B \times tr, 1)$，平行處理每個 batch，每個 block 處理一個 batch 中的一個 row block 以及所有 column block。

```
dim3 blockDim(32, 32);
dim3 gridDim(B*tr, 1);

FlashAttentionKernel<<<gridDim, blockDim>>>(
    d_Q, d_K, d_V, d_O, d_m, d_l, B, N, d, tr, tc, inv_sqrt_d
);
cudaDeviceSynchronize();
```

3. **Shared Memory Allocation**：

   - $Q_i : (64, 64 + 1)$
   - $K_j : (64, 64 + 1)$

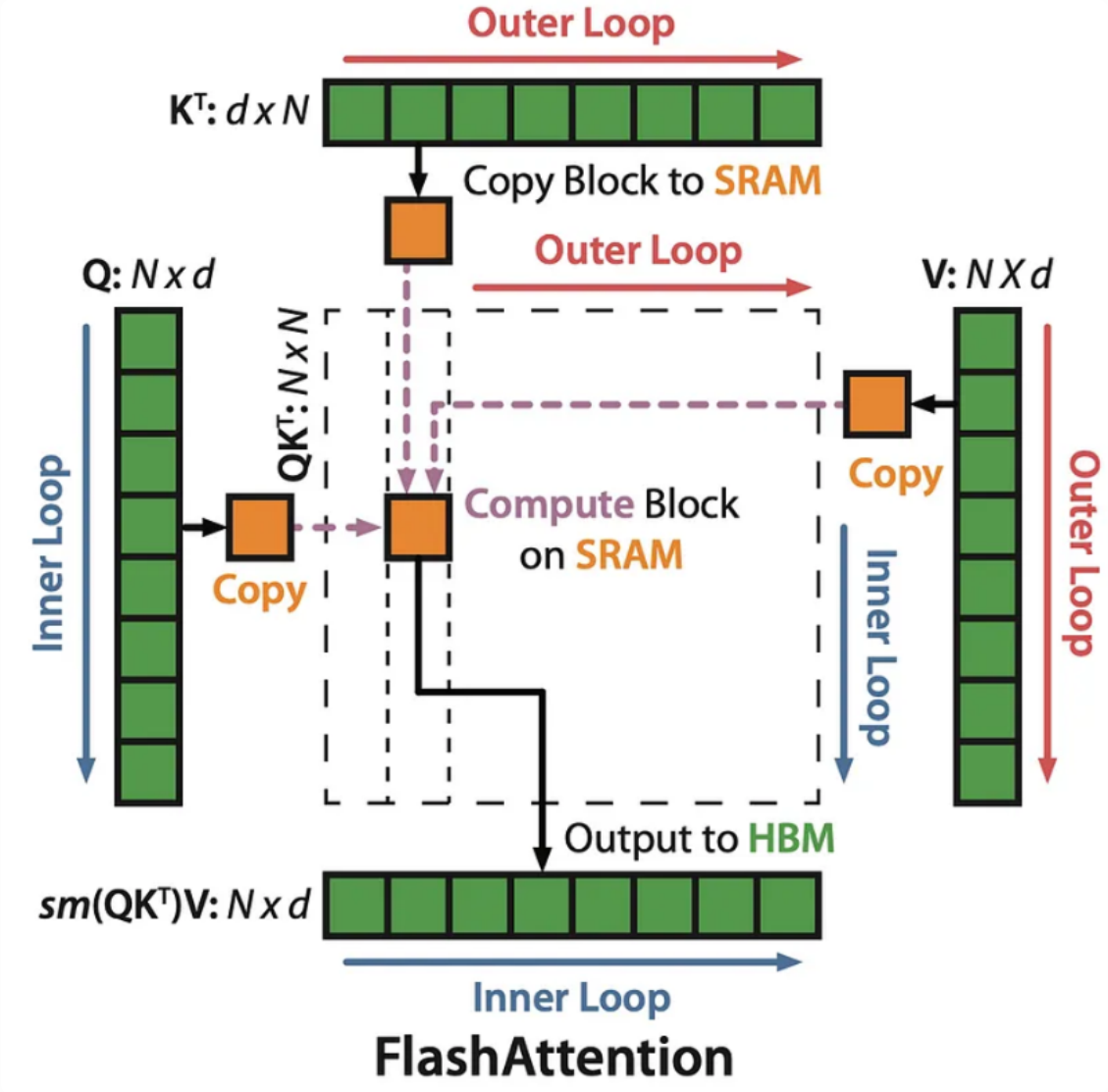
1.e Justify your choices and how they relate to the blocking factors and the SRAM size.

- 將 blockDim 設成 (32×32) 是因為一個 block 最多能使用 (32×32) 個 threads，我希望能盡可能的使用 GPU 上的資源，不過 threads 的使用量也會影響到一個 SM 可以同時執行的 block 數量，這部分可能需要進一步做實驗來確認這樣的設置是不是最佳的。

- 將 gridDim 設為 $(B \times tr, 1)$ 是因為：

  - 在 sequential 版本的實作中，最外層的迴圈是遍歷每個 batch，再來是遍歷每個 column block，最後是遍歷每個 row block，我發現 dependency 主要是在每個 column block 之

間，如下圖所示，所有的 row block 對同一個 column block 做完運算會將整個 output 矩陣更新一次，因此在計算下一個 column block 之前，需要等待上一個 column block 計算完成，而每個 row block 對同一個 column block 的運算會更新 output 矩陣中不同的部分，所以這部分可以平行化，而每個 batch 之間的運算當然也是獨立的，也可以平行化，所以才選擇將gridDim 設為 $(\mathrm{B} \times \mathrm{tr}, 1)$，這樣能將整個運算最大程度的平行化。



- Shared Memory Allocation：

  以下為使用的 shared memory 空間計算。每個 `float` 佔用 4 bytes。

  我在程式中分配的 shared memory 為：

  - $\mathrm{Q_i}$: $(64, 64 + 1)$
  - $\mathrm{K_j}$: $(64, 64 + 1)$

  這邊因為在測資中 d 只會是 32 或 64，所以用靜態分配的方式直接將 d 分配為 64，比較好解決 bank conflict 的問題，不過這樣只是為了方便，我覺得還是以動態分配的方式會比較好。

  總計 float 數量：

$$64 * 65 * 2 = 8320 \text{ floats}$$

轉換成 bytes：

$$8320 \text{ floats} \times 4 \text{ bytes/ float} = 33280 \text{ bytes}$$

一個 block 最多能使用的 shared memory 空間是 48KB = 49152 bytes，而 33280 bytes < 49152 bytes，並未超過一個 block 最多能使用的空間。

不過就如 1.c 提到的，因為我先嘗試了將 (BR, BC) 擴大為 $(64, 64)$，才加入 shared memory 的優化，不然我覺得將 (BR, BC) 設為 $(32, 32)$，然後將 Q、K、V、O 全部載入 shared memory 可能是更好的做法。(若將 Q、K、V、O 全部載入 shared memory，總共會使用 32768 bytes，也夠用)

---

2. Profiling Results

這邊我使用 `t22` 測資作為 input，因為只有一個 kernel，所以就對唯一的 `FlashAttentionBlock` kernel 進行 profile，並利用 nvprof 收集以下效能指標（metrics），包含：

- **achieved_occupancy**
- **sm_efficiency**
- **gld_throughput** (Global Load Throughput)
- **gst_throughput** (Global Store Throughput)
- **shared_load_throughput** (Shared Memory Load Throughput)
- **shared_store_throughput** (Shared Memory Store Throughput)

實驗平台為 apollo-gpu-server (NVIDIA GeForce GTX 1080)

Profiling Metrics 結果

Metric	Min Value	Max Value	Avg Value
achieved_occupancy	0.499992	0.499992	0.499992
sm_efficiency	99.93%	99.93%	99.93%
gld_throughput	1602.3GB/s	1602.3GB/s	1602.3GB/s
gst_throughput	48.365GB/s	48.365GB/s	48.365GB/s
shared_load_throughput	362.74GB/s	362.74GB/s	362.74GB/s
shared_store_throughput	6.2346GB/s	6.2346GB/s	6.2346GB/s

---

3. Experiment & Analysis

3.a System Spec

- 實驗平台為 apollo-gpu-server (NVIDIA GeForce GTX 1080)

3.b Optimization

優化一

將 (B_r, B_c) 從 $(32, 32)$ 擴大為 $(64, 64)$，讓一個 thread 從原本處理 $S_{ij} 中的一個元素變成處理 S_{ij} 中一個 2x2 的區塊 (4 個元素)，這部分在 1.c 中有詳細說明了，這邊就不再贅述。

在測試不同版本間的程式時，我將不同版本的程式跑過每個測資並把每個測茲的執行時間記錄下來，然後用一個 python 程式來比較兩個版本的程式在不同測資上的執行時間，以下是在做優化一前後的比較：

testcases	(B, N, d)	優化一前(s)	優化一後(s)	diff (後-前)
1	(320, 128, 32)	0.105	0.100	-0.005
2	(160, 128, 64)	0.117	0.124	+0.007
3	(160, 256, 32)	0.110	0.104	-0.006
4	(80, 256, 64)	0.114	0.109	-0.005
5	(80, 512, 32)	0.120	0.113	-0.007
6	(40, 512, 64)	0.119	0.122	+0.003
7	(40, 1024, 32)	0.147	0.133	-0.014
8	(20, 1024, 64)	0.150	0.138	-0.012
9	(20, 2048, 32)	0.207	0.175	-0.032
10	(10, 2048, 64)	0.194	0.185	-0.009
11	(13600, 128, 32)	0.642	0.560	-0.082
12	(9600, 128, 64)	0.826	0.742	-0.084
13	(6400, 256, 32)	0.857	0.732	-0.125
14	(2000, 256, 64)	0.550	0.521	-0.029
15	(2000, 512, 32)	0.907	0.755	-0.152
16	(800, 512, 64)	0.709	0.619	-0.090
17	(800, 1024, 32)	1.338	1.032	-0.306
18	(300, 1024, 64)	0.926	0.789	-0.137

19	(300, 2048, 32)	1.802	1.391	-0.411
20	(100, 2048, 64)	1.147	0.950	-0.197
21	(500, 2048, 32)	2.940	2.257	-0.683
22	(500, 2048, 64)	5.622	4.473	-1.149
23	(100, 4096, 32)	2.311	1.787	-0.524
24	(100, 4096, 64)	4.179	3.441	-0.738
25	(30, 8192, 32)	2.638	2.055	-0.583
26	(10, 8192, 64)	1.790	1.400	-0.390
27	(10, 16384, 32)	3.369	2.696	-0.673
28	(4, 16384, 64)	2.686	2.185	-0.501
29	(4, 32768, 32)	5.399	4.243	-1.156
30	(2, 32768, 64)	5.527	4.291	-1.236

可以看到幾乎在所有測資上都變快了。

優化二

我在 kernel 中有很多像是 `if (q_row_global_0 < N && k_col_global_0 < N)` 的判斷式，但在我仔細觀察測資特性後，我發現在這個作業的 Input specification 中：

$$N \in \{128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768\}$$

$N / 64$ 一定會整除，所以其實沒有必要有這些判斷式，拿掉之後可以減緩 warp divergence 的情況。

以下是在做優化二前後的比較：

testcases	(B, N, d)	優化二前(s)	優化二後(s)	diff (後-前)
1	(320, 128, 32)	0.100	0.110	+0.010
2	(160, 128, 64)	0.124	0.101	-0.023
3	(160, 256, 32)	0.104	0.100	-0.004
4	(80, 256, 64)	0.109	0.106	-0.003
5	(80, 512, 32)	0.113	0.113	+0.000
6	(40, 512, 64)	0.122	0.104	-0.018
7	(40, 1024, 32)	0.133	0.121	-0.012

8	(20, 1024, 64)	0.138	0.120	-0.018
9	(20, 2048, 32)	0.175	0.152	-0.023
10	(10, 2048, 64)	0.185	0.150	-0.035
11	(13600, 128, 32)	0.560	0.484	-0.076
12	(9600, 128, 64)	0.742	0.615	-0.127
13	(6400, 256, 32)	0.732	0.573	-0.159
14	(2000, 256, 64)	0.521	0.377	-0.144
15	(2000, 512, 32)	0.755	0.541	-0.214
16	(800, 512, 64)	0.619	0.439	-0.180
17	(800, 1024, 32)	1.032	0.738	-0.294
18	(300, 1024, 64)	0.789	0.535	-0.254
19	(300, 2048, 32)	1.391	0.871	-0.520
20	(100, 2048, 64)	0.950	0.593	-0.357
21	(500, 2048, 32)	2.257	1.454	-0.803
22	(500, 2048, 64)	4.473	2.824	-1.649
23	(100, 4096, 32)	1.787	1.076	-0.711
24	(100, 4096, 64)	3.441	2.021	-1.420
25	(30, 8192, 32)	2.055	1.257	-0.798
26	(10, 8192, 64)	1.400	0.896	-0.504
27	(10, 16384, 32)	2.696	1.623	-1.073
28	(4, 16384, 64)	2.185	1.275	-0.910
29	(4, 32768, 32)	4.243	2.532	-1.711
30	(2, 32768, 64)	4.291	2.491	-1.800

可以看到幾乎在所有測資上都變快了。

優化三

在以上版本中，我是把沿著 column block 的 outer loop 放在 kernel launch 外面的：

```
dim3 blockDim(32, 32);
dim3 gridDim(B*tr, 1);

for (int j = 0; j < tc; j++) {
    FlashAttentionBlock<<<gridDim, blockDim>>>(d_Q, d_K, d_V, d_O, d_m, d_l,
}
```

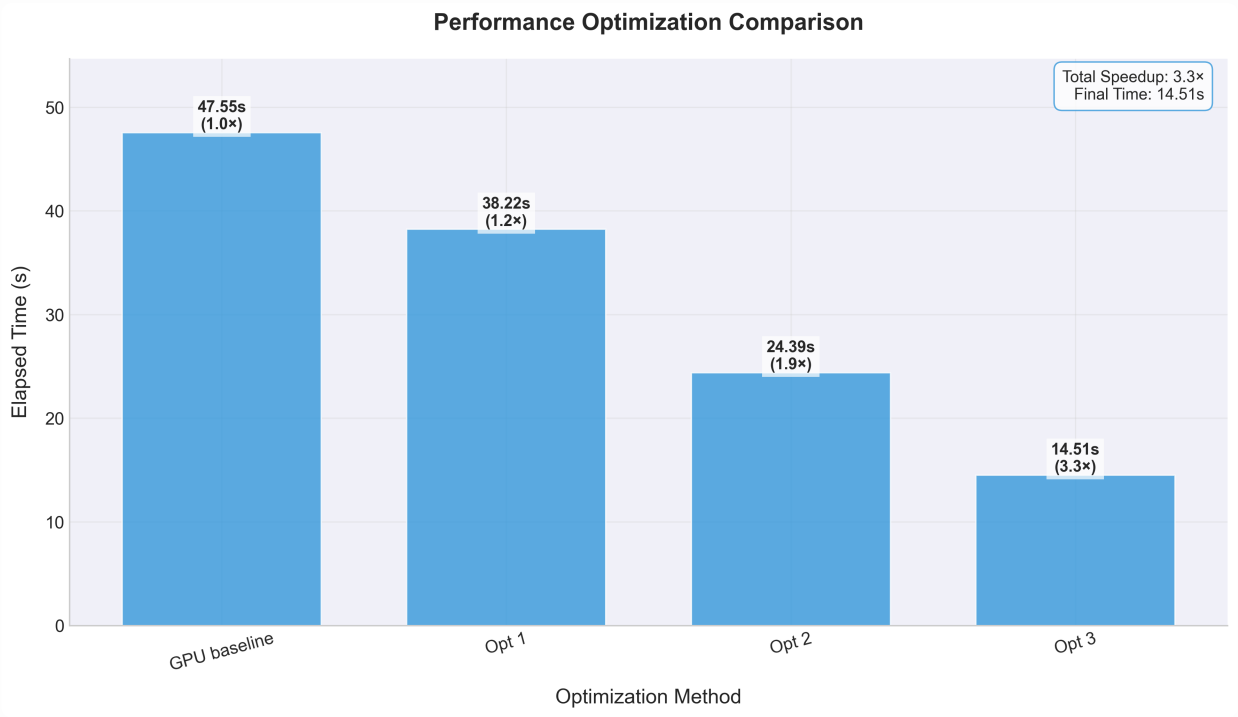
在這種架構下，每個 block 處理一個 batch 中的一個 row block 以及一個 column block，使用 shared memory 並沒有很大的效益，因為同一個 $\mathrm{Q_i}$ block 對 每一個 $\mathrm{K_j}$ block 的計算無法重用，所以我改成將沿著 column block 的 outer loop 寫進 kernel funtion 中，然後用 `__syncthreads()` 來同步每個 iterations，然後我將 $\mathrm{Q_i}$ block 跟 $\mathrm{K_j}$ block load 到 shared memory 中，並用 $(64, 64 + 1)$ 的方式解決 bank conflict 的問題。

以下是在做優化三前後的比較：

testcases	(B, N, d)	優化三前(s)	優化三後(s)	diff (後-前)
1	(320, 128, 32)	0.110	0.135	+0.025
2	(160, 128, 64)	0.101	0.108	+0.007
3	(160, 256, 32)	0.100	0.103	+0.003
4	(80, 256, 64)	0.106	0.102	-0.004
5	(80, 512, 32)	0.113	0.098	-0.015
6	(40, 512, 64)	0.104	0.098	-0.006
7	(40, 1024, 32)	0.121	0.104	-0.017
8	(20, 1024, 64)	0.120	0.103	-0.017
9	(20, 2048, 32)	0.152	0.118	-0.034
10	(10, 2048, 64)	0.150	0.118	-0.032
11	(13600, 128, 32)	0.484	0.407	-0.077
12	(9600, 128, 64)	0.615	0.542	-0.073
13	(6400, 256, 32)	0.573	0.448	-0.125
14	(2000, 256, 64)	0.377	0.316	-0.061
15	(2000, 512, 32)	0.541	0.386	-0.155
16	(800, 512, 64)	0.439	0.315	-0.124
17	(800, 1024, 32)	0.738	0.431	-0.307
18	(300, 1024, 64)	0.535	0.338	-0.197

19	(300, 2048, 32)	0.871	0.520	-0.351
20	(100, 2048, 64)	0.593	0.365	-0.228
21	(500, 2048, 32)	1.454	0.803	-0.651
22	(500, 2048, 64)	2.824	1.615	-1.209
23	(100, 4096, 32)	1.076	0.612	-0.464
24	(100, 4096, 64)	2.021	1.083	-0.938
25	(30, 8192, 32)	1.257	0.685	-0.572
26	(10, 8192, 64)	0.896	0.459	-0.437
27	(10, 16384, 32)	1.623	0.855	-0.768
28	(4, 16384, 64)	1.275	0.681	-0.594
29	(4, 32768, 32)	2.532	1.311	-1.221
30	(2, 32768, 64)	2.491	1.248	-1.243

可以看到幾乎在所有測資上都變快了。



這張圖片是我用所有測資測試不同版本的 GPU 程式所花的總時間以及 speedup 比較。

---

4. Experience & Conclusion

透過這個作業我學到了 flash attention 是如何加速 attention 的計算，不過在實際實作時，如果要更進一步優化，還是有蠻多點需要考量的，我覺得的確是個練習 GPU programming 的好題目。