HW4 Flash Attention

student ID: 112062619 **name:** 陳航希

1. Implementation

(a) overall, (b)QKV division, (d)configurations

我在這份flash attention的作業中，遵循了原本的論文的核心概念。

將Q,K,V以tile(對K,V), block(對Q)的方式分成 cuda的 grid, block, 所以在每輪的運算當中都會把其中一個block load 進sram 以後做運算。grid, block 的分法由下面所示，

因為每個 Q,K,V的shape都是 (Batch, N, D), 所以我讓每個 kernel 處理 (1, BR, D) 的資料, 也就是處理一個 batch 跟大小為BR的 sequences。

而每個block 又會有BR個warp, 每個warp會有32個thread, 所以最後每個warp會處理 (1,1,D) 的 sequence, 這個sequence 會使用32個thread來處理, 以此最大程度的使用1080的性能。

```
#define BR 32
#define BC 32
dim3 blockDim(32, BR);
dim3 gridDim(ceil_div(N, BR), Batch);
```

接著根據dimension大小呼叫不同設定的kernel

```
if (d == 32) {
    flash_attention_forward_kernel_t<32><<<gridDim, blockDim>>>(dQ, dK, dV, dO, N, Batch);
} else if (d == 64) {
    flash_attention_forward_kernel_t<64><<<gridDim, blockDim>>>(dQ, dK, dV, dO, N, Batch);
}
```

在進入kernel以後我首先會根據前面提到的方法計算每個warp會看到的index, 如下面的code所示, row_in_batch 可以得到這個warp需要處理的sequence index, 而lane則是這個warp當中 thread的index, 在接下來的計算中會使用這個index讓每個thread高效的從global mem拿資料到shared mem, 並在shared mem當中做運算。

```
const int lane = threadIdx.x;
const int r_in_block = threadIdx.y;

const int b = blockIdx.y;  // batch index
const int row_in_batch = blockIdx.x * BR + r_in_block;
```

在每一個kernel當中我都會load進 Q,K,V到shared mem當中，因為在flash attention的實作中，同一個kernel使用的Q都是相同的, 所以shared mem的部分使用BR作為load進去的大小。而因為 K,V 在同一個kernel裡面會需要把整個sequence length大小的tensor laod進去一次，所以會使用另外一個大小BC來作為shared mem中的大小，以此來保持計算的彈性，遇到不同的gpu可以改變大小，不需要都跟BR相同。

```
__shared__ float sQ[BR * D];
__shared__ float sK[BC * D];
__shared__ float sV[BC * D];
```

接著因為Q都是相同的，我會在一開始就將Q從global mem當中拿到shared mem中。Q 的每一row 會由 threadIdx.y 對應, 並以 float4 vectorized 方式輸入 shared mem sQ[]。

```
__shared__ float sQ[BR * D];
float *sQ_row = sQ + r_in_block*D;
const int vec_d = D/4;
const float4 *gQ4 = reinterpret_cast<const float4*>(Q + (size_t)row * D);
float4 *sQ4 = reinterpret_cast<float4*>(sQ_row);
for (int vidx=lane; vidx<vec_d; vidx+=32) {
    sQ4[vidx] = gQ4[vidx];
}__syncthreads();
```

跟Q不同, 因為K,V需要iterate over整個sequence, 在每次計算 output O的時候都會需要從K, V當中load進對應index的K, V進去，在load的過程中也同樣會用 float4 vectorized 方式輸入到 sK[]、sV[]。

```
const int num_k_blocks = (N + BC - 1) / BC;
for (int kb=0; kb<num_k_blocks; ++kb) {
    const int k_start = kb * BC;
    int cols = N - k_start;
    if (cols > BC) cols = BC;
    // load K, V tile for this batch
    for (int kk=r_in_block; kk<cols; kk+=BR) {
        const int key_row = b*N + (k_start + kk);
        const float *gK_row = K + (size_t)key_row * D;
        const float *gV_row = V + (size_t)key_row * D;
        float *sK_row = sK + kk*D;
        float *sV_row = sV + kk*D;
        const float4 *gK4 = reinterpret_cast<const float4*>(gK_row);
        const float4 *gV4 = reinterpret_cast<const float4*>(gV_row);
        float4 *sK4 = reinterpret_cast<float4*>(sK_row);
```

```
    float4 *sV4 = reinterpret_cast<float4*>(sV_row);
    for (int vidx=lane; vidx<vec_d; vidx+=32) {
        sK4[vidx] = gK4[vidx];
        sV4[vidx] = gV4[vidx];}}
```

接下來就進入到要計算attention的部分, 在flash attention計算的過程中會需要保存的就是算到目前為止score的最大值mi(用來做softmax數值穩定性)，, 與li(到目前為止sotmax attention score的平均值，用來計算value的貢獻值大小)。scale為一個常數，為計算attention的時候需要的normalization factor此外，這兩個變數（mi, li）對應到 flash attention 中的 online softmax 更新規則，可以讓整個演算法避免建立完整的 N×N attention matrix，只需要逐行更新即可，這也是他最主要的節省 SRAM/HBM 帶寬的來源之一。acc 則是對應到輸出向量 O 的部分累積值，以 warp-friendly 的 chunk(32 elements 為一組)切分，每次遇到新的 key/value row 都會被更新一次。

```
float mi = -1e30f;
float li = 0.0f;
float acc[(D + 31) / 32];
#pragma unroll
for (int u=0; u<(D+31)/32; ++u) acc[u]=0.0f;
const float scale = 1.0f/sqrtf((float)D);
```

開始在BC的範圍當中計算過去attension score的影響跟這一輪的attention score。在這裡, 程式會取出 shared memory 裡面第 kk 列的 key,value，並以 warp 為單位進行 dot product。所有 thread 只處理屬於自己 lane 的維度(例如 lane 0 處理維度 0, 32, 64…)，可以確保訪問是 coalesced 並且方便使用 warp-level reduction。

```
  for (int kk=0; kk<cols; ++kk) {
      float *k_vec = sK + kk*D;
      float *v_vec = sV + kk*D;

      float partial = 0.0f;
      #pragma unroll
      for (int t=lane; t<D; t+=32) {
          partial += sQ_row[t] * k_vec[t];
      }
      float dot = warp_reduce_sum(partial);
```

warp 減法完後由 lane 0 得到整體 dot product，並同步給所有 thread。
接著進行 numerically-stable softmax 的必要步驟。
這裡的 m_new、alpha、beta、l_new 是在線softmax的核心: m_new 是目前為止最大的 score, alpha 用來縮放舊的 softmax sum, beta 是新的 score 對 softmax 的貢獻, l_new = alpha*li + beta 則是新的 softmax normalization denominator.

```
      float score = 0.0f;
      if (lane == 0) score = dot * scale;
```

```
    score = __shfl_sync(0xffffffffu, score, 0);

    float m_new = fmaxf(mi, score);
    float alpha = __expf(mi - m_new);
    float beta  = __expf(score - m_new);
    float l_new = alpha*li + beta;
```

最後就是把前面運算的結果加起來。這裡的更新方式是flash attention中weighted sum of V的版本。coeff_prev 決定舊的累積值 acc 與新的基準對齊時應縮放多少, coeff_new 是這一次 score 對應的權重, acc[u] 則是代表輸出向量 O 的一段。

```
  float coeff_prev = (li>0.0f) ? (alpha*li/l_new) : 0.0f;
  float coeff_new  = beta / l_new;
  #pragma unroll
  for (int u=0; u<(D+31)/32; ++u) {
      const int t = lane + u*32;
      if (t < D) {
          float v_val = v_vec[t];
          acc[u] = coeff_prev*acc[u] + coeff_new*v_val;
      }
  }
  mi = m_new;
  li = l_new;
```

做完以後根據thread的lane index值將資料填回global mem中。

```
#pragma unroll
for (int u=0; u<(D+31)/32; ++u) {
  const int t = lane + u*32;
  if (t < D) {
      O[(size_t)row*D+t] = acc[u];
  }
}
```

(c)how to choose Br,Bc, (e)Justify your choices

選擇BR=32, BC=32的理由主要是因為 gtx1080 的硬體限制剛好很適合這個大小。

1080 的 warp 大小是 32, 所以一列Q 直接讓一個 warp 處理非常自然,不需要額外同步,也能用 __shfl 做 reduction。另一方面，它每個 block 的 shared mem 上限約 48 KB，而 Q/K/V 三個 tile 加起來在d=64 時大約 24 KB，可以放得下，不會因為 shared mem 用太多讓 block 無法 launch。此外，32 這個大小在記憶體讀取上也比較整齊。因為d 是 32 或 64，float4 存取剛好對齊，KV tile 用 32 rows 也能保持 coalesced。

2. Profiling Results

```
pp25s051@apollo-login:~/hw4/kk4$ srun -p nvidia -N1 -n1 --gres=gpu:1 nvprof --metrics achieved_occupancy,sm_efficiency,shared_load_throughput,s
hared_store_throughput,gld_throughput,gst_throughput ./hw4 /home/pp25/share/hw4/testcases/t25 t25.out
==1824055== NVPROF is profiling process 1824055, command: ./hw4 /home/pp25/share/hw4/testcases/t25 t25.out
==1824055== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
(B, N, d): (30, 8192, 32)
Time (GPU): 5.303 seconds
==1824055== Profiling application: ./hw4 /home/pp25/share/hw4/testcases/t25 t25.out
==1824055== Profiling result:
==1824055== Metric result:
Invocations                               Metric Name                         Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: void flash_attention_forward_kernel_t<int=32>(float const *, float const *, float const *, float*, int, int)
          1                        achieved_occupancy                        Achieved Occupancy    0.499995    0.499995    0.499995
          1                                  sm_efficiency                   Multiprocessor Activity      99.95%      99.95%      99.95%
          1                          shared_load_throughput             Shared Memory Load Throughput  515.79GB/s  515.79GB/s  515.79GB/s
          1                         shared_store_throughput            Shared Memory Store Throughput  43.066GB/s  43.066GB/s  43.066GB/s
          1                                 gld_throughput                  Global Load Throughput  10.767GB/s  10.767GB/s  10.767GB/s
          1                                 gst_throughput                  Global Store Throughput  21.491MB/s  21.491MB/s  21.491MB/s
pp25s051@apollo-login:~/hw4/kk4$
```

3. Experiment & Analysis

a.System Spec: 實驗使用課程提供的 apollo-gpu，使用助教提供的測資 t25 做實驗紀錄，並使用nvprof or CUDA event 兩種方法紀錄程式執行時間。

b.Optimization:

優化的部分，我總共實作六個不同的版本，如下所示:

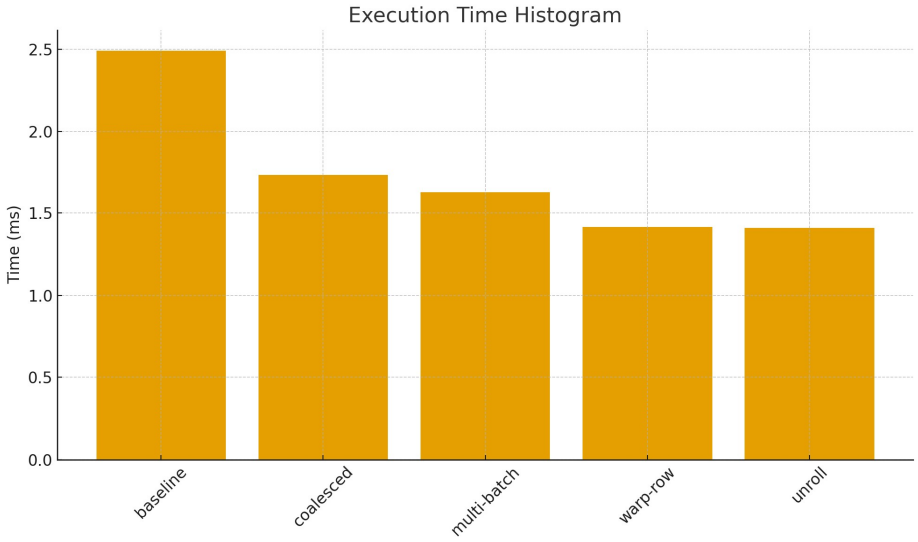
GPU baseline：僅依照 FlashAttention 基本流程實作、未加入任何 GPU 最佳化技巧的版本: -> 2.493

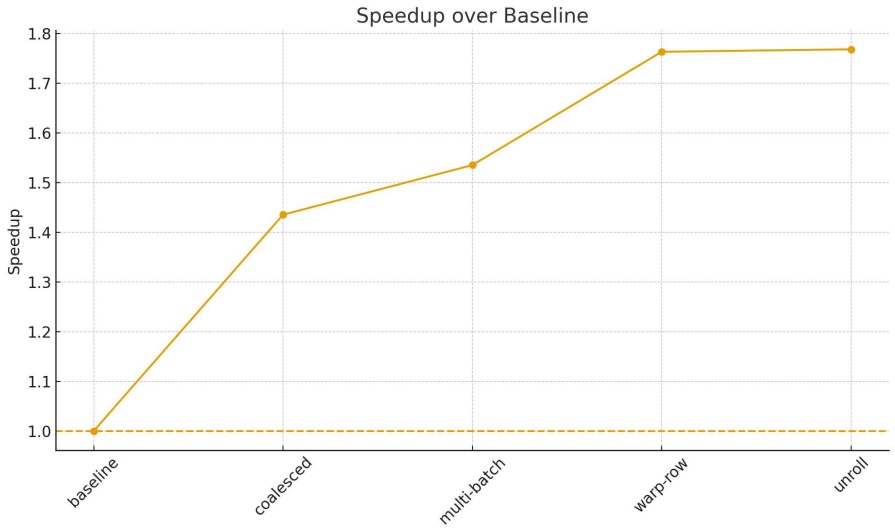
Coalesced global memory access: 在baseline的基礎上,加入fast math, 用 float4 加速 K/V 載入 -> 1.737

kernel支援多batch運算: 移除了外層 batch 迴圈, 一次把全部 Batch, N, d 丟上 GPU -> 1.624

加入blockDim(32, BR): 讓一整個warp算同一列的dot -> 1.414

加入unroll: 算for loop時加入 #pragma unroll -> 1.410



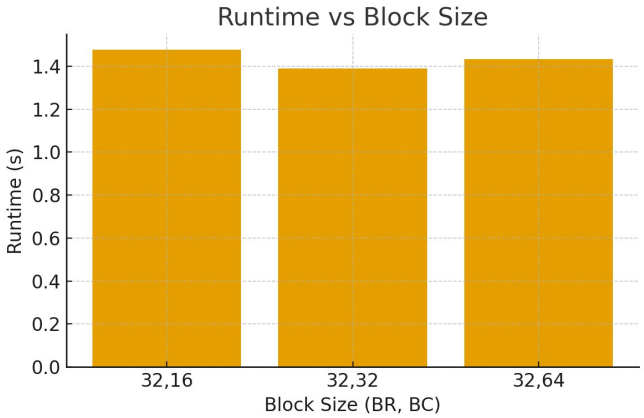


Others

1. BC大小對執行速度影響

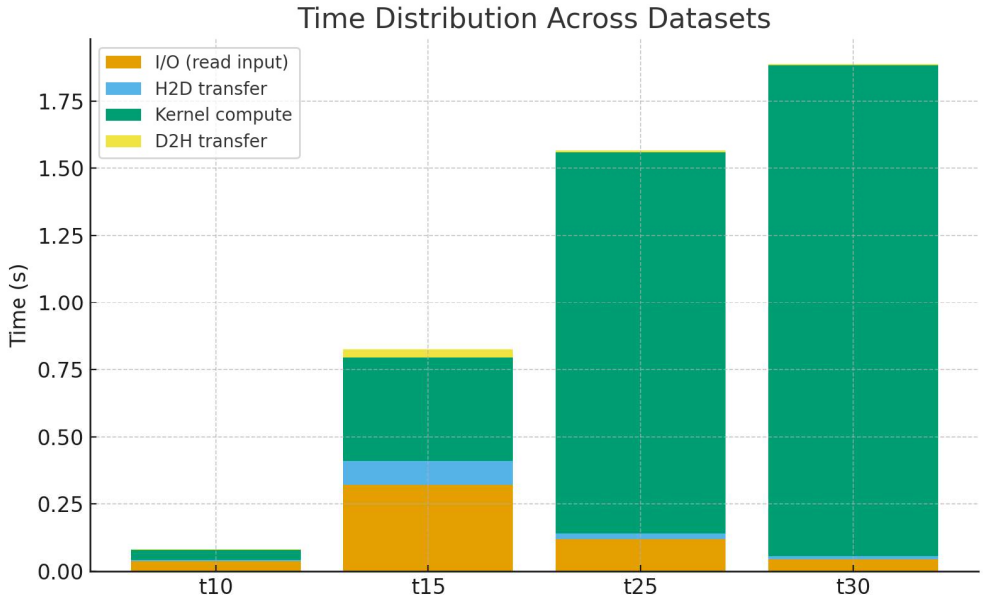
下面根據不同的BC大小看執行速度的差異, BR都固定為32。實驗結果顯示(32, 32) 的執行時間最低,而 (32, 16) 最慢。這三組的差異主要來自記憶體重複使用程度的不同:當 BC 過小,例如 16 時,每次載入的 K/V tile 容量不足，需要進行更多輪 global memory 存取，因此整體時間上升；而當 BC 過大（如 64），雖然減少了迭代次數，但 shared memory 壓力提高，使得效能依然無法比上平衡的 (32, 32) 配置。

Block Size (BR, BC)	Runtime (s)
32, 16	1.476
32, 32	1.389
32, 64	1.432



2. time distribution分析

這裡做了不同測資大小下會看到的time distribution差別，結果可以看到：當測資變大時，整體執行時間由 kernel compute 所主導，尤其在 t25 與 t30 中，kernel 已佔到九成左右。H2D 與 D2H 的成本相當固定，而 IO 則根據不同測資會有不同結果，相當特別。



4. AMD GPU Porting and Analysis

a. Implementation

我將原本的 CUDA 程式改成 HIP，以便在 AMD MI210 上執行。移植過程跟hw3相同，主要是把 CUDA 的 API 換成 ROCm 的 HIP API，其餘程式結構幾乎不變。

1.API 對應替換

cudaMalloc → hipMalloc

cudaFree → hipFree

kernel 呼叫語法 <<< >>> 保持相同

2. 標頭檔替換

#include <cuda_runtime.h> → #include <hip/hip_runtime.h>

3. Kernel 內容維持不變

thread/block 索引、__shared__、__syncthreads() 都能直接在 hip 上使用

b. Use profiling tools to see differences

在 AMD 實驗中, 我使用 nvprof 與 rocprof 取得 kernel time, 結果顯示 MI210 約為 GTX1080 的 3.2 倍速度。推測原因是 FlashAttention 在本次實作中屬於 computation-bound，MI210 具有更多計算單元、更高向量運算吞吐量、以及 CDNA2 架構提供的 MFMA/VALU 優勢, 所以在計算密集型負載中呈現明顯效能提升。

GPU	Kernel Time	Profiling 工具
NVIDIA GTX1080	1.378 s	nvprof
AMD MI210	0.433 s	rocprof –stats

one optimization (hw3-2)

在嘗試做 AMD 版本的優化時,我原本的想法很直接: 既然 MI210 的執行單位是 64-thread 的 wavefront,而我之前的 cuda 寫法是完全依照 32-thread warp 設計的,那把 kernel 的 block 維度調整成符合 64-thread 會不會更快?帶著這個假設,我把 blockDim.x 從32改成 64,並把相關的 warp intrinsic 也換成 64-thread 的版本,希望讓每個 wavefront 在 AMD 上能被完全用滿。

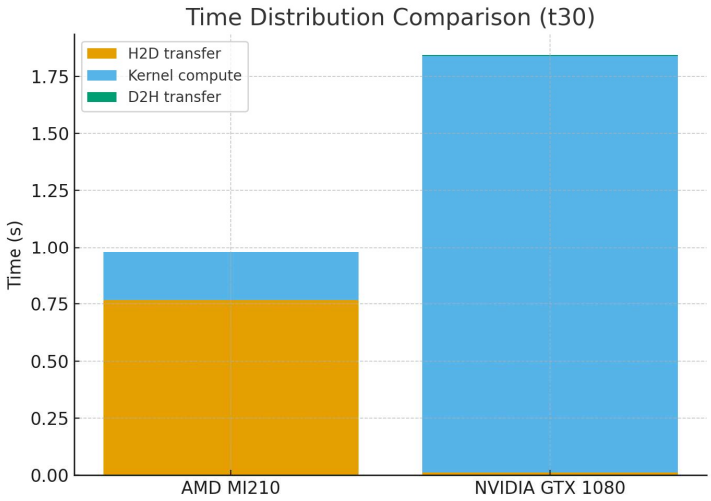
但實際跑起來卻完全不是這回事。這個版本不但沒有加速,反而變得明顯更慢。猜測是因為flash attention 的效能不是靠”一次更多 thread” 達成,而是靠 “同一個 tile 能被多條 row 重複利用” 。我把 mapping 改成每個 wavefront 只算一條 row 之後,原本的 tile reuse 完全被破壞,每條 row 都得重新載入 K/V,記憶體流量大量增加。另外,原始程式裡所有 thread 分工, stride,shared memory 讀寫模式都是為 32-thread warp 最佳化的, 硬改成 64-thread 反而讓資料對齊,coalescing 和 memory access pattern 全部變差。

版本	MI210 Kernel Time
baseline HIP	1.230 s
wavefront-64 (AMD-only opt)	5.433 s

所以這次 AMD 專屬的 64-thread wavefront 優化最後證明是無效的。它雖然形式上更符合 MI210 的硬體規格,但卻不符合 FlashAttention 本身的演算法特性,導致變慢。

c. Others - time distribution

在這個實驗中,我使用t30這個測資測試amd MI210 與 gtx1080 之間的差別, 可以發現到相當明確的差別。測試中, AMD MI210 的 kernel time 大約是 0.211 秒, 而 NVIDIA GTX 1080 的 kernel time 是 1.827 秒, 速度差了大約 8.6 倍。這組結果可以看到 flash attention 這個 computation-bound 的工作在 MI210 上有天然優勢, 有就是運算單元比1080 多很多。但mi210在h2d花的時間卻更多，這個部分還需要再額外研究才會清楚。



5. Experience & conclusion

這次實作 FlashAttention 的過程讓我對 GPU 程式效能有更深的理解。最大的體會是記憶體存取模式的重要性: 善用 shared memory, 減少不必要的 global memory 存取，就能帶來明顯的加速，我的最終版本大約提升了 3.5 倍。此外，在調整 batch size 時也觀察到效能並非越大越好，過大的 BS 反而會造成記憶體壓力與衝突，必須在效能與資源之間找到平衡。

整個寫作業的過程充滿挑戰，包括 indexing 的複雜度, thread 用量的安排, tile reuse 的策略，以及 debug 時的反覆驗證。然而透過一次次調整與測試，我更清楚理解到 GPU 最

佳化其實是硬體特性、演算法設計與資料流動三者共同作用的結果。這次作業不只讓我把 FlashAttention 跑起來，也讓我對平行程式的設計與思考方式有更扎實的體會。