Parallel Programming

Homework 4: FlashAttention

- 111065524 李懷-

Implementation

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. 先將整個 batch(B*N*d)的 QKV cudaMemcpy 到 GPU 的 global memory 上:

```
cudaMalloc(&d_K, B * N * d * sizeof(float));
cudaMalloc(&d_V, B * N * d * sizeof(float));
cudaMalloc(&d_Q, B * N * d * sizeof(float));
cudaMalloc(&d_O, B * N * d * sizeof(float));

cudaMemcpy(d_K, K, B * N * d * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_V, V, B * N * d * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_Q, Q, B * N * d * sizeof(float), cudaMemcpyHostToDevice);
// cudaMemcpy(d_O, O, B * N * d * sizeof(float), cudaMemcpyHostToDevice);
cudaMemset(d_O, 0, B * N * d * sizeof(float));
```

```
dim3 gridDim(N/Br, B); // N/Br query tiles per batch, B batches, each block each tile in o
int shared_mem_size = (Br *d + Bc *(d+1) + Bc * d) * sizeof(float); // q,o,k,v tiles size
FusedKernel<<<gridDim, dim3(Bc, Br), shared_mem_size>>>(d_Q, d_K, d_V, d_O, N, d, scale);
```

每個 block 是對應到一個 batch 裡的一個 Q tile(Br*d)，一開始先將其對應到的 global memory 的 Q 的資料 load 進 shared memory q_i:

```
int global_tile_start = batch_idx * N + tile_idx * Br;

int tid = ty * blockDim.x + tx; // coalesced access
int total_threads = blockDim.x * blockDim.y;

// 2D block
for (int idx = tid; idx < Br * dim; idx += total_threads) {
    q_i[idx] = q[(global_tile_start) * dim + idx];
}

float m_val = -FLT_MAX;
float l_val = 0.0f;

__syncthreads();
```

2. 由於每個 q_tile 需要與每個 k/v tile 去做計算，所以這邊每個 block 會 iterate through N/Bc，而 Block size 為 Bc*Br，因為一個 q tile 裡的每個 token 需與 k/v tile 裡的每個 token 去做 dot product，所以 Bc*Br 個 threads 可以讓每個 thread 對應計算其 Sij, mij 跟ℓij，首先，個別先將對應 global 的 k/v tile load 進 shared memory:

```
// loop over each k,v tile
for (int kv_tile_idx=0; kv_tile_idx < N/Bc; kv_tile_idx++) {
    // each thread load a k,v tile in a batch into shared mem
    int kv_offset = batch_idx * N + kv_tile_idx * Bc;

    for (int r = ty; r < Bc; r += blockDim.y) {
        for (int c = tx; c < dim; c += blockDim.x) {
            k_j[r * (dim + 1) + c] = k[(kv_offset + r) * dim + c];
            v_j[r * dim + c]       = v[(kv_offset + r) * dim + c];
        }
    }

    __syncthreads();
```

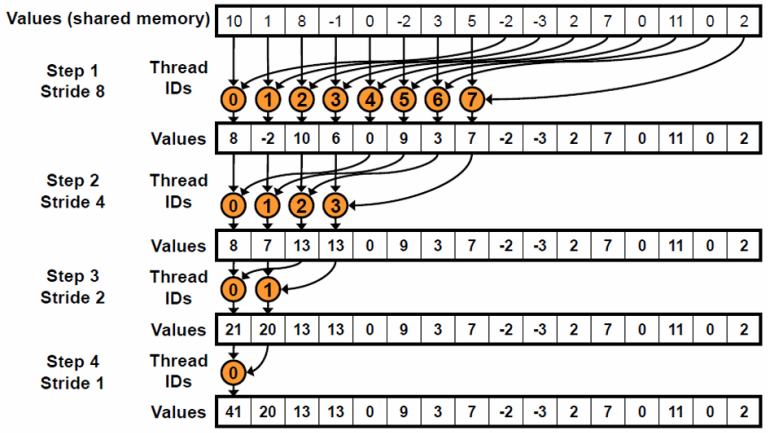
每個 thread(tx,ty)算出其對應的 scaled dot product，也就是S_{ij}:

```
// 2. Compute QK (sij)
float sij = 0.0f;
for (int d = 0; d < dim; d++) {
    sij += q_i[ty * dim + d] * k_j[tx * (dim+1) + d];
}
sij *= scale;
```

計算 row max m_{ij}:

```
// 3. Row Max
float mij = sij;
for (int offset = Bc/2; offset > 0; offset /= 2)
    mij = fmaxf(mij, __shfl_xor_sync(0xFFFFFFFF, mij, offset, Bc));
```

這裡用 warp shuffle reduction 的方式將一個 row(Bc 個)中的 max Sij 找出來(由於使用 warp 的 intrinsic，所以 Bc 需<=32)，每個 warp 裡的 threads 會將其 register 與+offset 的 thread 的 mij 去做比較，用__shfl_xor_sync()的方式可以達到 butterfly reduction 的效果，類似下面這種 parallel reduction 的方式:



把每個 sij 扣掉最大值後取 exp 算出 pij 後，用上面 row max 的 reduction 的方式來算 row sum:

```
// 4. MinusMaxAndExp
pij = expf(sij - mij);

// 5. RowSum
float lij = pij;
for (int offset = Bc/2; offset > 0; offset /= 2)
    lij += __shfl_xor_sync(0xFFFFFFFF, lij, offset, Bc);
```

接著得出每個 Bc tile 的 iteration 時的 max m_i(e.g. m=max(m1,m2))，進而求出當下 iteration 所有 exponential 的加總 ℓ_i:

```
// 6. UpdateMiLiOi
float m_prev = m_val;
float l_prev = l_val;

float m_new = fmaxf(m_prev, mij);
float l_new = expf(m_prev - m_new) * l_prev + expf(mij - m_new) * lij;

m_val = m_new;
l_val = l_new;

float scale_O = (l_prev * expf(m_prev - m_new));
float scale_PV = (expf(mij - m_new));
```

$$m_1 = max([1,2]) = 2 \qquad m_2 = max([3,4]) = 4 \qquad m = max(m_1, m_2) = 4$$

$$f_1 = [e^{1-2}, e^{2-2}] = [e^{-1}, e^{0}] \qquad f_2 = [e^{3-4}, e^{4-4}] = [e^{-1}, e^{0}] \qquad f = [e^{m_1-m} f_1, e^{m_2-m} f_2] = [e^{-3}, e^{-2}, e^{-1}, e^{0}]$$

$$l_1 = \sum f_1 = e^{-1} + e^{0} \qquad l_2 = \sum f_2 = e^{-1} + e^{0} \qquad l = e^{m_1-m} l_1 + e^{m_2-m} l_2 = e^{-3} + e^{-2} + e^{-1} + e^{0}$$

$$o_1 = \frac{f_1}{l_1} = \frac{[e^{-1}, e^{0}]}{e^{-1} + e^{0}} \qquad o_2 = \frac{f_2}{l_2} = \frac{[e^{-1}, e^{0}]}{e^{-1} + e^{0}} \qquad o = \frac{f}{l} = \frac{[e^{-3}, e^{-2}, e^{-1}, e^{0}]}{e^{-3} + e^{-2} + e^{-1} + e^{0}}$$

$$m = max(m_1, m_2) = 4$$

ℓ_{new}的計算方式如下:

$$l = e^{m_1-m} l_1 + e^{m_2-m} l_2 = e^{-3} + e^{-2} + e^{-1} + e^{0}$$

算出 oi 後寫回 global memory:

```
// 7. Compute Output (O=PV) and Write to Global Memory
for (int col = tx; col < dim; col += blockDim.x) {
    float pv_val = 0.0f;
    // Oi = PijVj, iterate over each of Bc
    for (int c = 0; c < Bc; c++) {
        float p_c = __shfl_sync(0xFFFFFFFF, pij, c, Bc);  // Pij of each tile
        pv_val += p_c * v_j[c * dim + col];
    }

    int o_idx = (global_tile_start + ty) * dim + col;
    float o_old = o[o_idx];
    o[o_idx] = (scale_O * o_old + scale_PV * pv_val) / l_new;
}

__syncthreads();
```

最後在進入下一個 Bc tile 前 syncthreads()。

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

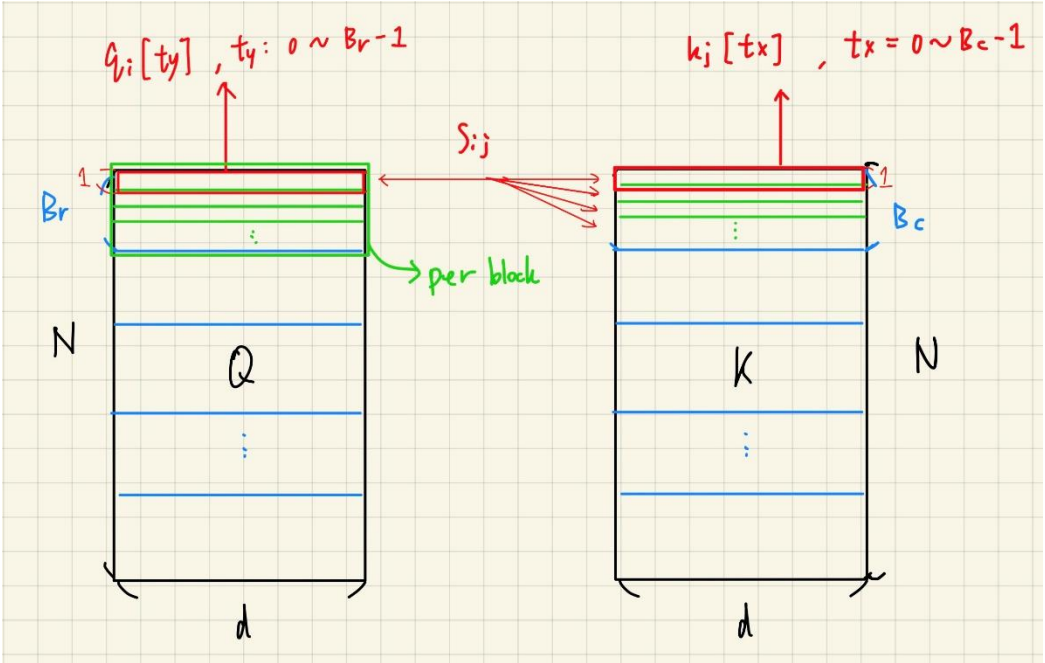
#blocks 設(N/Br, B)個，B 是讓每個 grid 的 row(blockIdx.y)對應到一個 batch 的 sequence(N*d)，由於將每個 sequence 切成 N/Br 等分(N mod Br = 0)，每個 blockIdx.x 對應到一個 Q tile (Br*d):

```
dim3 gridDim(N/Br, B); // N/Br query tiles per batch, B batches, each block each tile in o
int shared_mem_size = (Br *d + Bc *(d+1) + Bc * d) * sizeof(float); // q,o,k,v tiles size
FusedKernel<<<gridDim, dim3(Bc, Br), shared_mem_size>>>(d_Q, d_K, d_V, d_O, N, d, scale);
```

#threads 為 Bc*Br，因為一個 q tile 裡的每個 token 需與 k/v tile 裡的每個 token 去做運算，所以 Bc*Br 個 threads 可以讓每個 thread 負責計算其S_{ij}, m_{ij}跟ℓ_{ij}:

```
int tx = threadIdx.x;
int ty = threadIdx.y;
int batch_idx = blockIdx.y;
int tile_idx = blockIdx.x; // Q tile index: N/Br

int global_tile_start = batch_idx * N + tile_idx * Br;

int tid = ty * blockDim.x + tx; // coalesced access
int total_threads = blockDim.x * blockDim.y;
```

每個 thread 計算 dot product S_{ij}的示意圖:



global_tile_start 就是一個當下 block 對應到的 q tile 在 global 開始的 index，因為一個 tile 有 Br*d 的 data 需要 load，我讓 Br*Bc 個 threads 去分攤

```
// loop over each k,v tile
for (int kv_tile_idx=0; kv_tile_idx < N/Bc; kv_tile_idx++) {
    // each thread load a k,v tile in a batch into shared mem
    int kv_offset = batch_idx * N + kv_tile_idx * Bc;
    for (int r = ty; r < Bc; r += blockDim.y) {
        for (int c = tx; c < dim; c += blockDim.x) {
            k_j[r * (dim + 1) + c] = k[(kv_offset + r) * dim + c];
            v_j[r * dim + c]       = v[(kv_offset + r) * dim + c];
        }
    }

    __syncthreads();
```

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

我 Bc 是設 32，Br 設 16，由於在 row max/sum 時使用到了 warp 的 intrinsic 的關係，所以 Bc 需<=32，Br 之所以設 16，主要是當 Br 設 32 時，occupancy 會過低(50%)，但照其 allocate 的 shared memory size 來說，每個 sm 應該能有 2 個 blocks，所以 occupancy 應該要接近 100%，我猜測可能的原因是當 32*32 個 threads 時，一個 block 會用掉太多 registers，以至於 registers per sm 被一個 block 用滿，導致一個 sm 上一次只能有一個 block，進而拉低了 occupancy，將 Br 降為 16 能改善此問題，後來也實測了 Br=16 的確較快。

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

我 grid 是設(N/Br, B)，block 設(Bc*br)個 threads，Bc 設 32，Br 設 16；

shared memory 的部分由於 QKV tile 的的 size 分別為 Br*d, Bc*d 及 Bc*d，而 d 是根據 input 決定的，所以這邊我們需要 dynamically allocate shared memory，以下為 kernel launch:

```
dim3 gridDim(N/Br, B); // N/Br query tiles per batch, B batches, each block each tile in o
int shared_mem_size = (Br *d + Bc *(d+1) + Bc * d) * sizeof(float); // q,o,k,v tiles size
FusedKernel<<<gridDim, dim3(Bc, Br), shared_mem_size>>>(d_Q, d_K, d_V, d_O, N, d, scale);
```

```
extern __shared__ float smem[]; // (Br*d + Bc*d + Bc*d)*

float *q_i = smem; // Br * d
float *k_j = &smem[Br * dim]; // Bc * d
float *v_j = &smem[Br * dim + Bc * (dim+1)]; // Bc * d
```

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

由於 GTX 1080 的 shared memory per block 上限是 49152 bytes，且 testcase 中 input dimension 最大為 64，而論文中有提到 Br 及 Bc 的 size:

$$\text{Set block sizes } B_c = \lceil \frac{M}{4d} \rceil, B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$$

但我們有 q,k 及 v 三個 tiles 需要 allocate，所以 Br 及 Bc 的上限為 49152/(4*64*3)=64，但考量到在 row max/sum 的部分有使用到 warp shuffle reduction 的關係，所以 Bc 需<=32，且 Br*Bc<=1024。

Profiling Results

Bc=32, Br=16, Testcase: t30:

```
Invocations                               Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: FusedKernel(float*, float*, float*, float*, int, int, float)
          1                    shared_ld_bank_conflict           0           0           0           0
          1                    shared_st_bank_conflict           0           0           0           0

======== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: FusedKernel(float*, float*, float*, float*, int, int, float)
          1                         branch_efficiency                             Branch Efficiency     100.00%     100.00%     100.00%
          1                           gld_throughput                       Global Load Throughput  138.71GB/s  138.71GB/s  138.71GB/s
          1                           gst_throughput                      Global Store Throughput  27.737GB/s  27.737GB/s  27.737GB/s
          1                   shared_load_throughput                  Shared Memory Load Throughput  2662.7GB/s  2662.7GB/s  2662.7GB/s
          1                  shared_store_throughput                 Shared Memory Store Throughput  110.97GB/s  110.97GB/s  110.97GB/s
          1                        shared_efficiency                       Shared Memory Efficiency      69.00%      69.00%      69.00%
          1                            inst_integer                          Integer Instructions  4.2753e+11  4.2753e+11  4.2753e+11
          1                          sm_efficiency                        Multiprocessor Activity      99.56%      99.56%      99.56%
          1                     achieved_occupancy                             Achieved Occupancy    0.748216    0.748216    0.748216
```

Bc=32, Br=32, Testcase: t30:

```
Invocations                               Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: FusedKernel(float*, float*, float*, float*, int, int, float)
          1                    shared_ld_bank_conflict           0           0           0           0
          1                    shared_st_bank_conflict           0           0           0           0

======== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: FusedKernel(float*, float*, float*, float*, int, int, float)
          1                         branch_efficiency                             Branch Efficiency     100.00%     100.00%     100.00%
          1                           gld_throughput                    Global Load Throughput  70.756GB/s  70.756GB/s  70.756GB/s
          1                           gst_throughput                   Global Store Throughput  23.578GB/s  23.578GB/s  23.578GB/s
          1                    shared_load_throughput             Shared Memory Load Throughput  2263.4GB/s  2263.4GB/s  2263.4GB/s
          1                   shared_store_throughput            Shared Memory Store Throughput  47.178GB/s  47.178GB/s  47.178GB/s
          1                         shared_efficiency                   Shared Memory Efficiency      68.37%      68.37%      68.37%
          1                             inst_integer                      Integer Instructions  3.4592e+11  3.4592e+11  3.4592e+11
          1                           sm_efficiency                  Multiprocessor Activity      99.42%      99.42%      99.42%
          1                      achieved_occupancy                        Achieved Occupancy    0.499998    0.499998    0.499998
```

這裡我用 nvprof 去 profile occupancy, sm efficiency, shared/global memory throughput 等 metrics，可以看到，當 Br 設 32 時，occupancy 會較低，將 Br 降為 16 能提升 occupancy；shared 及 global memory 的 throughput 也是有顯著提升。

Experiment & Analysis (Nvidia GPU)

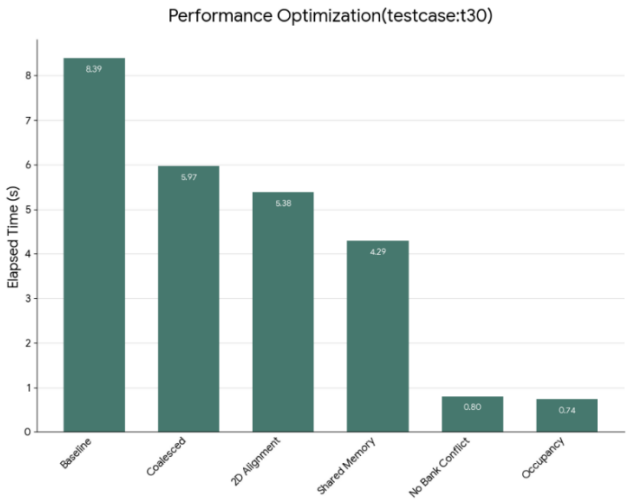
a. System Spec:

使用的機器為這門課提供的 Apollo Server。

b. Optimization

我依序試了以下幾種 optimizations，optimizations 是 cumulative 的，所以每一步都是在前一項優化的基礎上繼續優化，使用的 testcase 為 t30:

1. Coalesced memory access: 讀取 global memory 時確保 warp 讀取連續的地址；
2. 2D alignment: 由原本 baseline 的 1D block 改成 2D block；
3. Shared memory(w/ bank conflicts): 將 q,k,v tile load 進 shared memory 去 reuse，但因在計算 dot product 時 warp access k tile 的方式是 access by column 的，導致嚴重的 bank conflicts；
4. Shared memory(w/o bank conflicts):用 memory padding 的方式將 k tile 的 shared memory tile 的 column 加 1 來移除 bank conflict；
5. Occupancy optimization: 將 block size 由 32*32 改為 32*16，提高 occupancy；



w/ bank conflicts:

```cpp
// 2. Compute QK (sij)
float sij = 0.0f;
for (int d = 0; d < dim; d++) {
    sij += q_i[ty * dim + d] * k_j[tx * dim + d];
}
sij *= scale;
```

可以看到 shared memory load 有大量的 bank conflicts，主要發生在存取 k tile 時:

```
Invocations                               Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: FusedKernel(float*, float*, float*, float*, int, int, float)
          1                  shared_ld_bank_conflict  1.3314e+11  1.3314e+11  1.3314e+11  1.3314e+11
          1                  shared_st_bank_conflict           0           0           0           0

======== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: FusedKernel(float*, float*, float*, float*, int, int, float)
          1                          gld_throughput                    Global Load Throughput  11.661GB/s  11.661GB/s  11.661GB/s
          1                          gst_throughput                   Global Store Throughput  3.8858GB/s  3.8858GB/s  3.8858GB/s
          1                   shared_load_throughput             Shared Memory Load Throughput  4227.7GB/s  4227.7GB/s  4227.7GB/s
          1                  shared_store_throughput            Shared Memory Store Throughput  7.7754GB/s  7.7754GB/s  7.7754GB/s
          1                        shared_efficiency                  Shared Memory Efficiency      6.15%       6.15%       6.15%
          1                       achieved_occupancy                       Achieved Occupancy   0.500000    0.500000    0.500000
```

Memory padding(w/o bank conflicts):

```cpp
int shared_mem_size = (Br *d + Bc *(d+1) + Bc * d) * sizeof(float); // q,o,k,v tiles size
// FusedKernel<<<gridDim, Bc, shared_mem_size>>>(d_Q, d_K, d_V, d_O, N, d, scale); // each t
FusedKernel<<<gridDim, dim3(Bc, Br), shared_mem_size>>>(d_Q, d_K, d_V, d_O, N, d, scale);
```

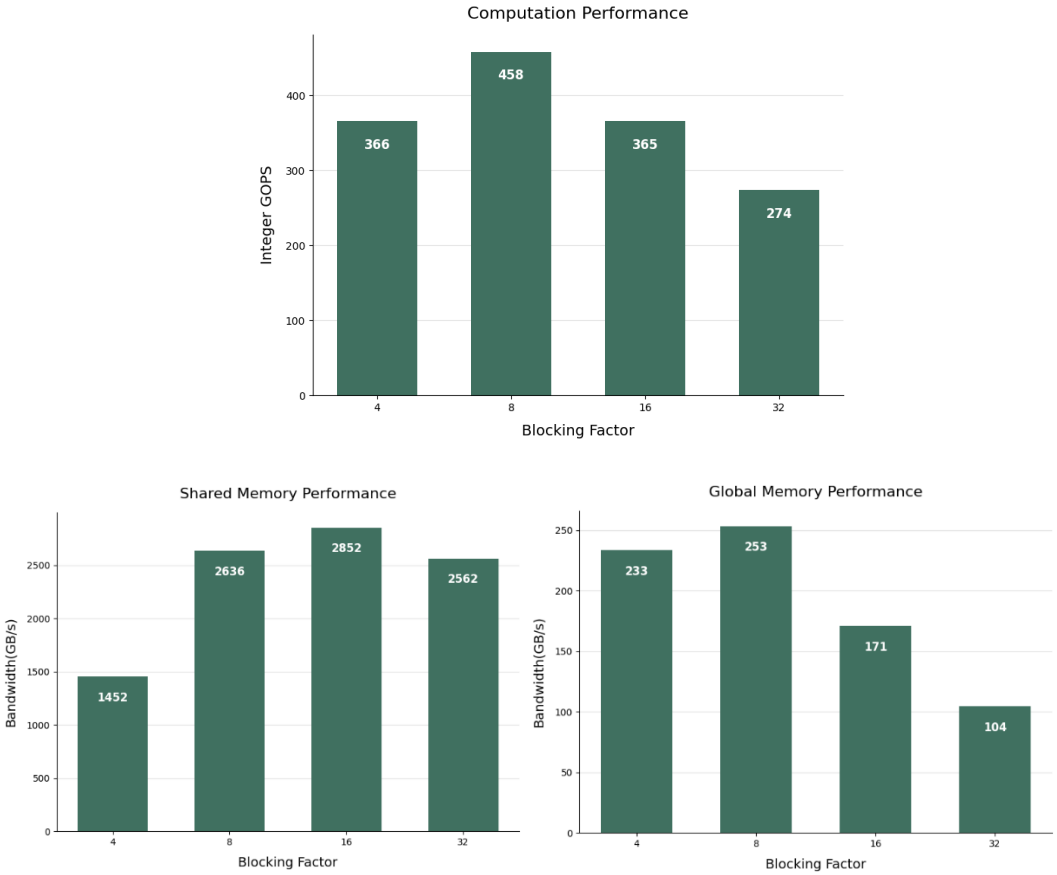
```cpp
// 2. Compute QK (sij)
float sij = 0.0f;
for (int d = 0; d < dim; d++) {
    sij += q_i[ty * dim + d] * k_j[tx * (dim+1) + d];
}
sij *= scale;
```

可以看到移除 bank conflicts 後，shared memory efficiency 大幅地上升:

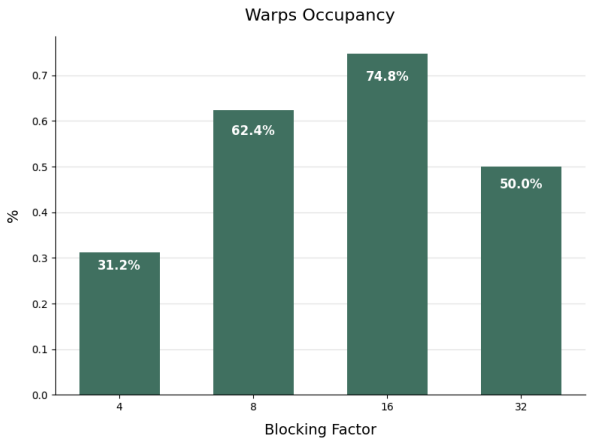
```
Invocations                               Event Name         Min         Max         Avg       Total
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: FusedKernel(float*, float*, float*, float*, int, int, float)
          1                  shared_ld_bank_conflict           0           0           0           0
          1                  shared_st_bank_conflict           0           0           0           0

======== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "NVIDIA GeForce GTX 1080 (0)"
    Kernel: FusedKernel(float*, float*, float*, float*, int, int, float)
          1                          gld_throughput                    Global Load Throughput  65.114GB/s  65.114GB/s  65.114GB/s
          1                          gst_throughput                   Global Store Throughput  21.698GB/s  21.698GB/s  21.698GB/s
          1                   shared_load_throughput             Shared Memory Load Throughput  2083.0GB/s  2083.0GB/s  2083.0GB/s
          1                  shared_store_throughput            Shared Memory Store Throughput  43.416GB/s  43.416GB/s  43.416GB/s
          1                        shared_efficiency                  Shared Memory Efficiency      68.37%      68.37%      68.37%
          1                       achieved_occupancy                       Achieved Occupancy   0.499998    0.499998    0.499998
```

c. Different Br:

可以看到當 Br 設 8 時 Int GOPS 來到最高，之後隨著 Br 上升逐漸下降:



而 occupancy 則是在 Br=16 時達到最高:



這與 Hw3 的 blocked Floyd Warshall 的結果略有不同，Flash attention 由於一次會從 global memory 讀取更多 data，以至於需要更高的 occupancy 去掩蓋 access global memory 時 stall 的 latency，可以看到當 Br=32 時，global memory bandwidth 變低，但 occupancy 又下降。

AMD GPU Porting and Analysis

a. Implementation:

Explain how you ported the CUDA code to HIP code:

基本上 implementation 跟 cuda 版本一樣，只是將 cuda 的 function 替換成 hip，例 cudaMalloc -> hipMalloc。

b. Experiment & Analyze:

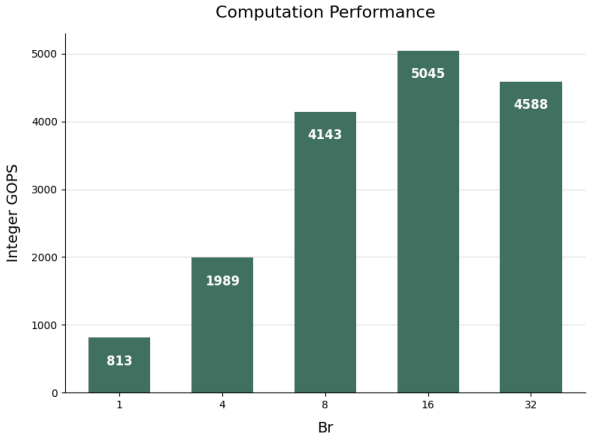
我是用 rocprof 去做 profiling，metrics 有 SQ_INSTS_VMEM_WR, SQ_INSTS_VMEM_RD , SQ_INSTS_VALU, SQ_LDS_BANK_CONFLICT, SQ_INSTS_LDS，由於 rocprof 沒有直接提供 shared/global memory throughput，所以只能透過 SQ_INSTS_LDS 跟 SQ_INSTS_VMEM_RD/WR 及 durations 去算，各算法如下:

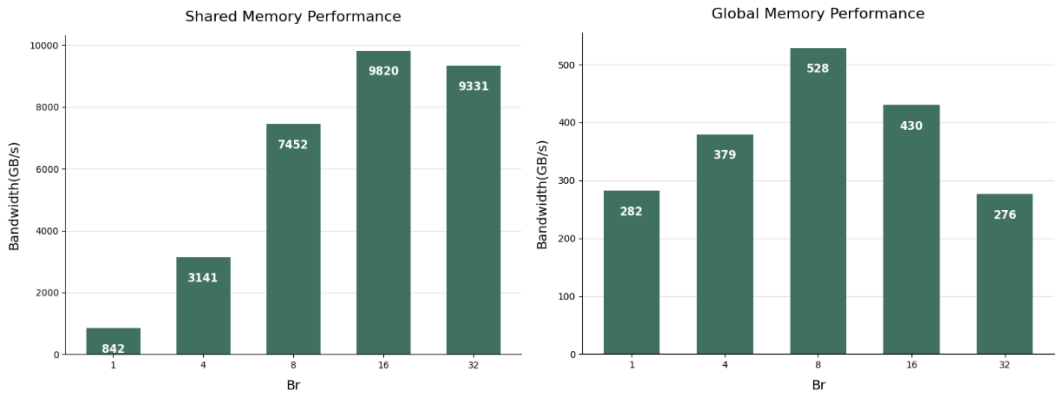
$$Int\ GOPS = \frac{I_{avg}}{t_{avg}(ns) * 10^{-9}} * 10^{-9}$$

$$Bandwidth = \frac{R_{avg}}{t_{avg}(ns) * 10^{-9}} * 10^{-9}$$

1. $I_{phase\_i_{avg}}$: Kernel 一個 invocation 的平均 integer instructions，取自 SQ_INSTS_VALU*64(wavefront size)；
2. $R_{phase\_i_{avg}}$: Kernel 一個 invocation 的平均 shared mem/global mem 的 instructions，shared mem : SQ_INSTS_LDS*64*4 , global mem: SQ_INSTS_VMEM_RD/WR *64*4bytes；
3. $t_{phase\_i_{avg}}$: Kernel 一個 invocation 的平均執行時間，單位為 ns；

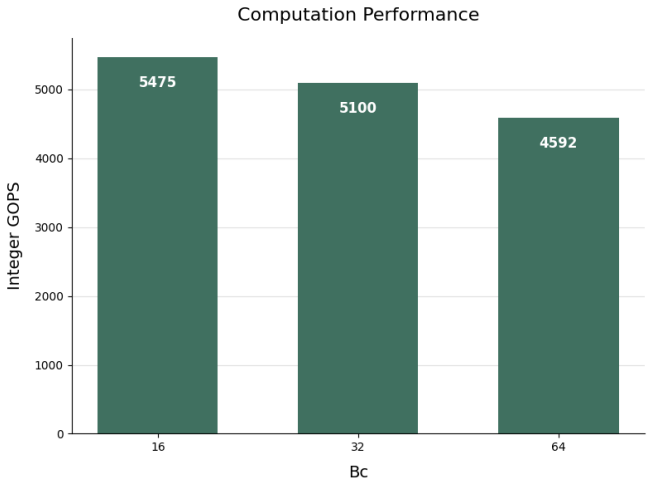
1. Different Br:





可以看到跟 Gtx 1080 比起來，AMD 的 Integer GOPS 隨著 Br 上升地較明顯，即便 occupancy 一樣下降，這可能是因為 MI210 的 VRAM 為 HBM2e，其 bandwidth 高出 1080 的 gddr5x 許多，表示一次可以有更多 threads 進行 access，也就導致其 stall latency 較小，不需要過高的 occupancy 去掩蓋 latency。

另外，相較於 NV 的 warp size 為 32，AMD CDNA2 架構的 warp size(Wavefront)為 64，所以我們在做 warp shuffle reduction 時可以一次做 64 個 threads，可將 Bc 提高至 64，但可以看到，提高 Bc 對於整體的 computation performance 並沒有提升:



Experience & conclusion

a. What have you learned from this homework?

透過這次 Flash attention 的實作及論文的閱讀後，我深刻體會到其是如何透過 tiling 及 shared memory 來改善原先 attention 機制的不足的，且相較於 Hw3 的 Blocked Floyd Warshall 演算法，這次作業對 shared memory 的讀取頻率更高，因此更能體會優化 shared memory 讀取前後（例如移除 bank conflicts）的效能差異。