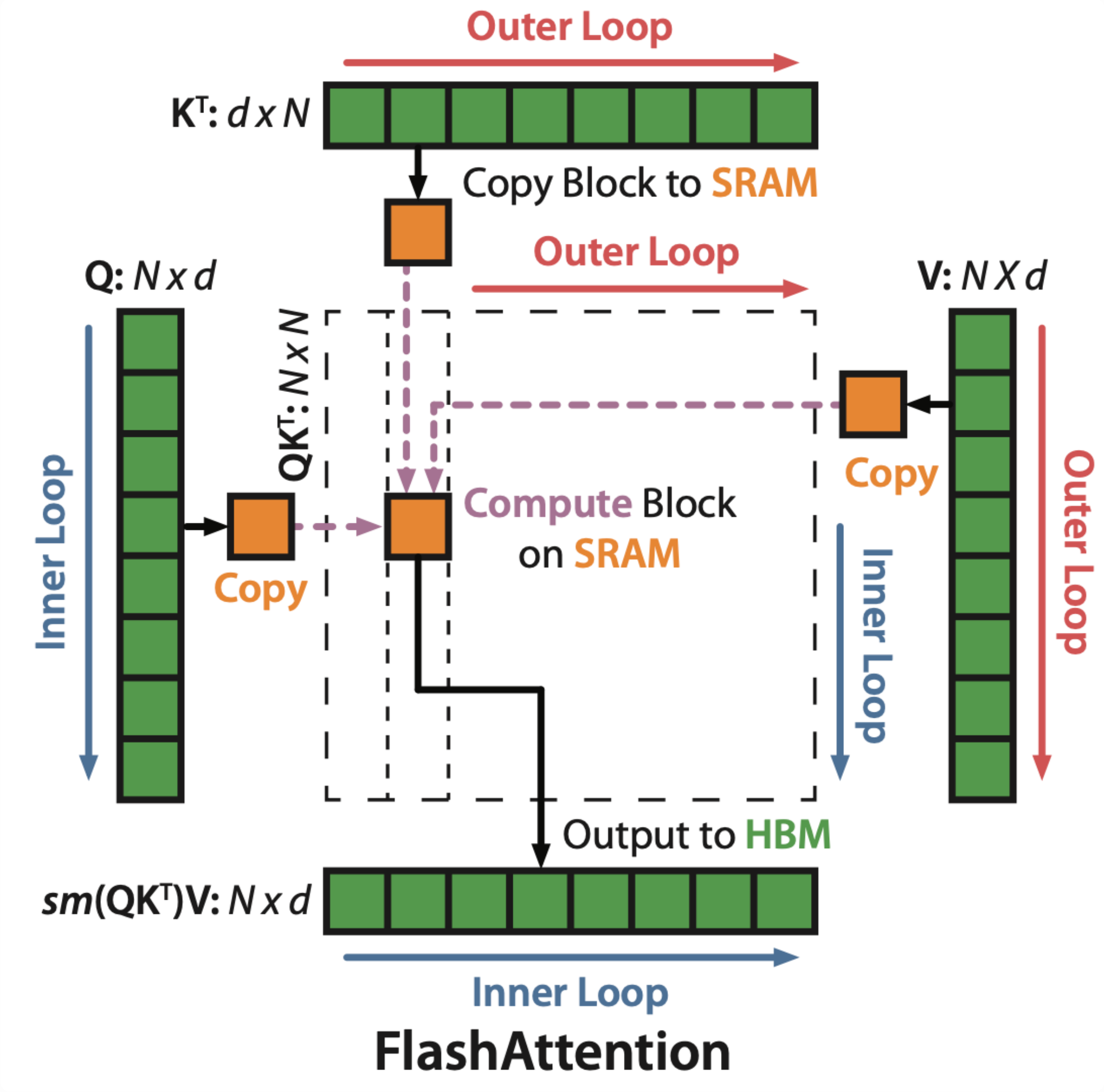
2024 Parallel Programming HW4 [112062610 劉得崙]

> Flash attention illustration
>
> 
>

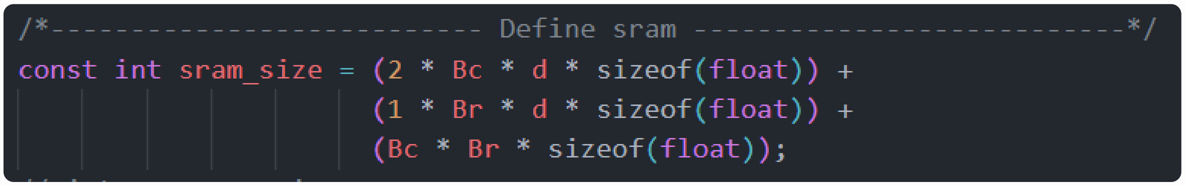
1. Implementation

a. Describe how you implemented the FlashAttention forward pass.

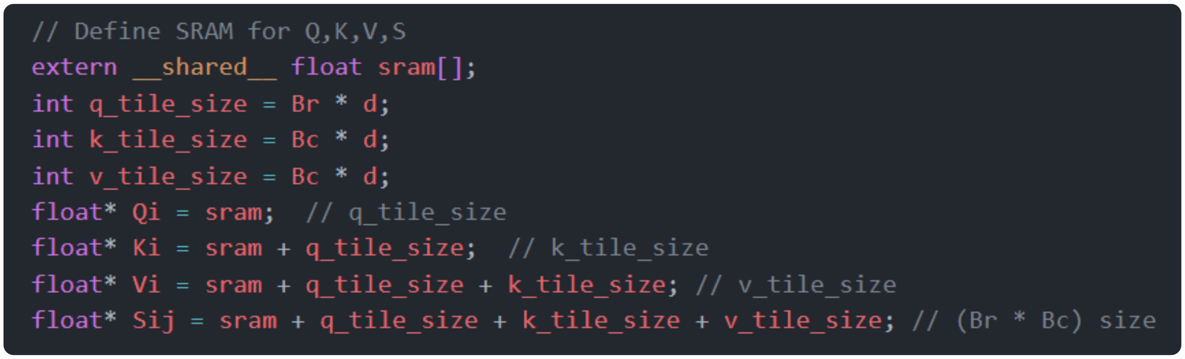
- Sram size
  Qi : Br * d * sizeof(float)
  Ki : Bc * d * sizeof(float)

Vi : Bc * d * sizeof(float)

Sij: Br * Bc * sizeof(float)

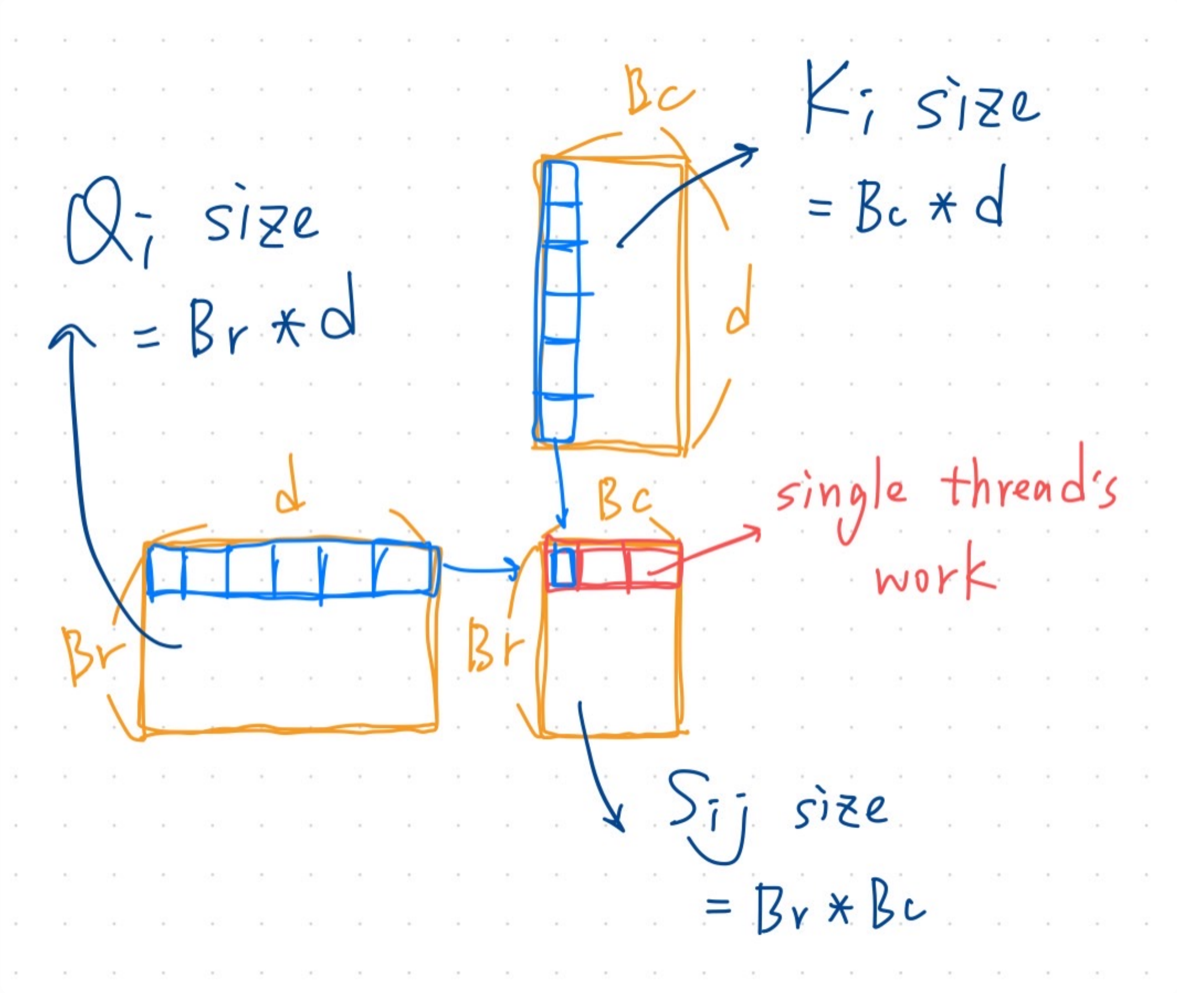


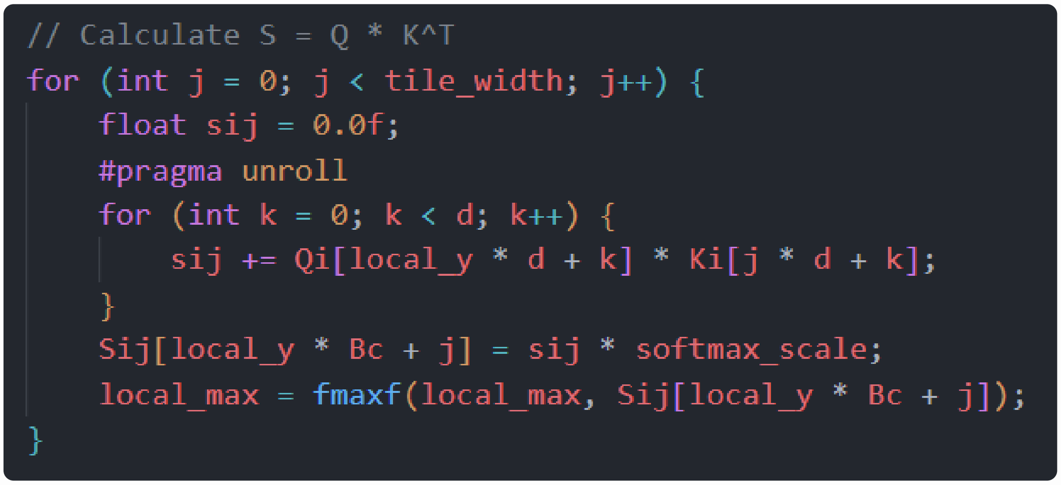
```
/*--------------------------- Define sram ---------------------------*/
const int sram_size = (2 * Bc * d * sizeof(float)) +
                      (1 * Br * d * sizeof(float)) +
                      (Bc * Br * sizeof(float));
```



```
// Define SRAM for Q,K,V,S
extern __shared__ float sram[];
int q_tile_size = Br * d;
int k_tile_size = Bc * d;
int v_tile_size = Bc * d;
float* Qi = sram;  // q_tile_size
float* Ki = sram + q_tile_size;  // k_tile_size
float* Vi = sram + q_tile_size + k_tile_size; // v_tile_size
float* Sij = sram + q_tile_size + k_tile_size + v_tile_size; // (Br * Bc) size
```

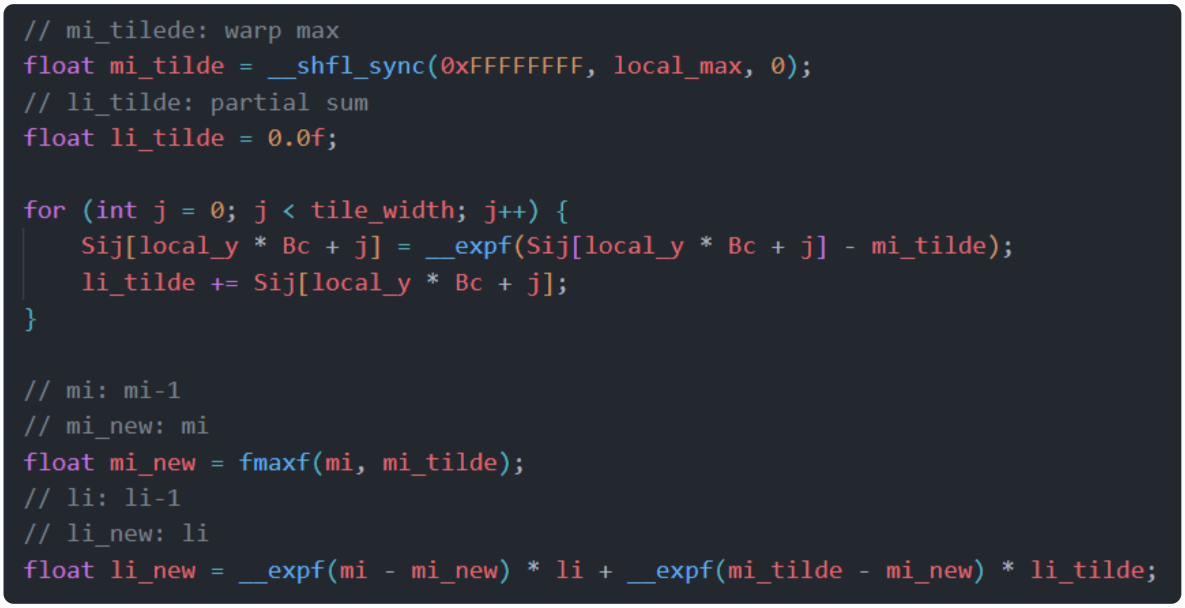
- S = Q * K^T





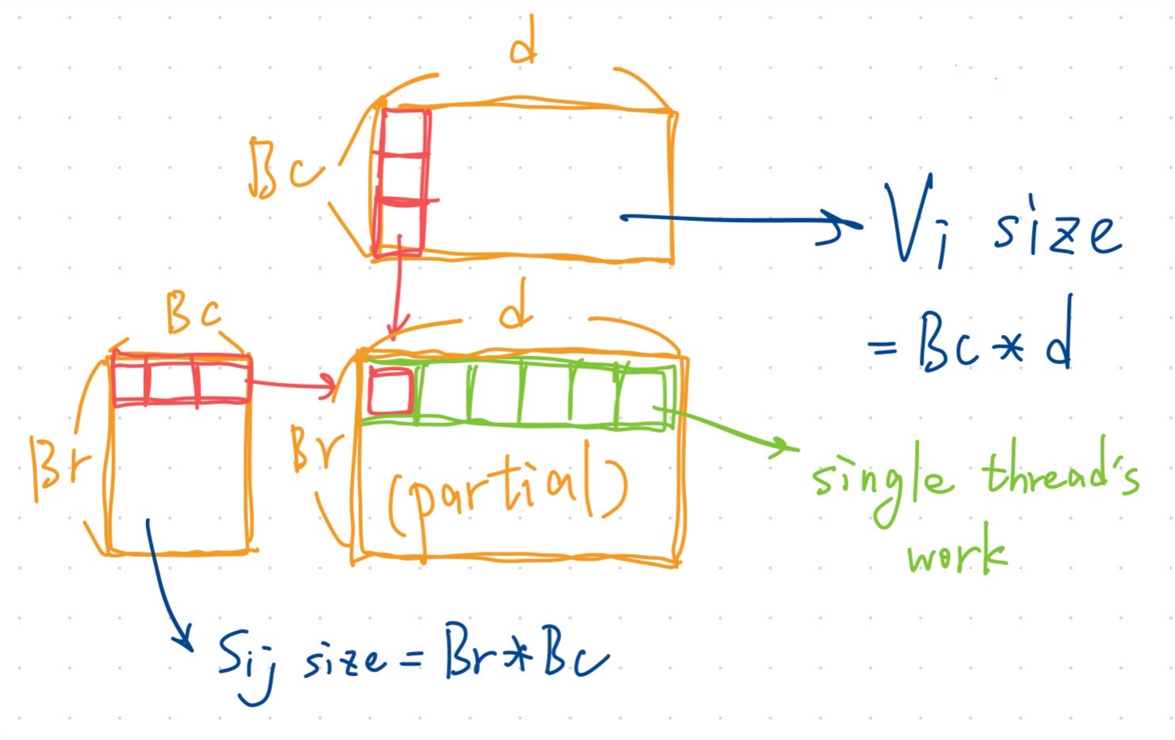
```
// Calculate S = Q * K^T
for (int j = 0; j < tile_width; j++) {
    float sij = 0.0f;
    #pragma unroll
    for (int k = 0; k < d; k++) {
        sij += Qi[local_y * d + k] * Ki[j * d + k];
    }
    Sij[local_y * Bc + j] = sij * softmax_scale;
    local_max = fmaxf(local_max, Sij[local_y * Bc + j]);
}
```

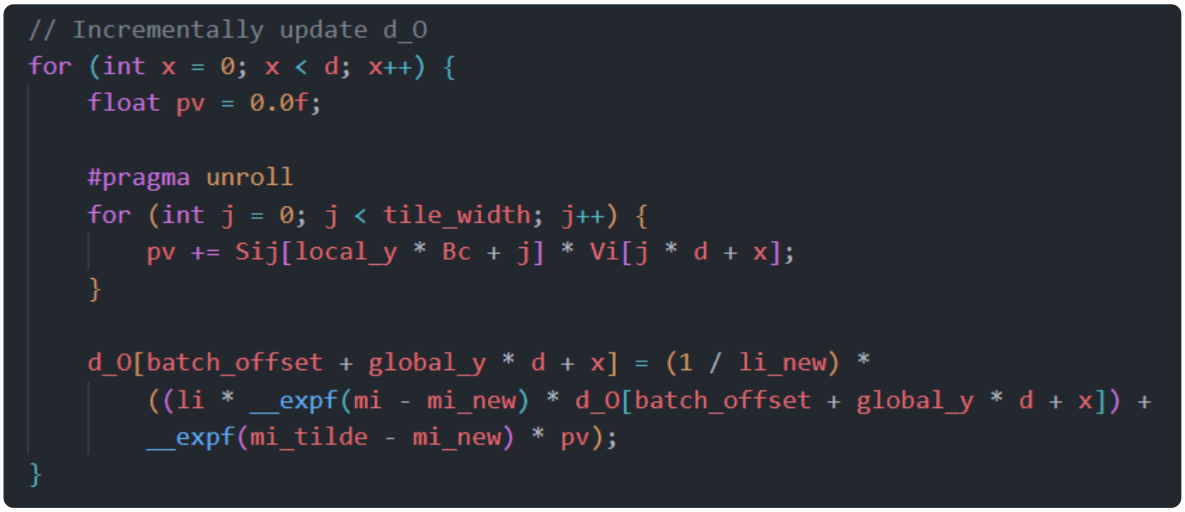
- Record warp max (mi_tilde) and partial sum (li_tilde)



```
// mi_tilede: warp max
float mi_tilde = __shfl_sync(0xFFFFFFFF, local_max, 0);
// li_tilde: partial sum
float li_tilde = 0.0f;

for (int j = 0; j < tile_width; j++) {
    Sij[local_y * Bc + j] = __expf(Sij[local_y * Bc + j] - mi_tilde);
    li_tilde += Sij[local_y * Bc + j];
}

// mi: mi-1
// mi_new: mi
float mi_new = fmaxf(mi, mi_tilde);
// li: li-1
// li_new: li
float li_new = __expf(mi - mi_new) * li + __expf(mi_tilde - mi_new) * li_tilde;
```

- `O = S * V`. Incrementally update d_O.



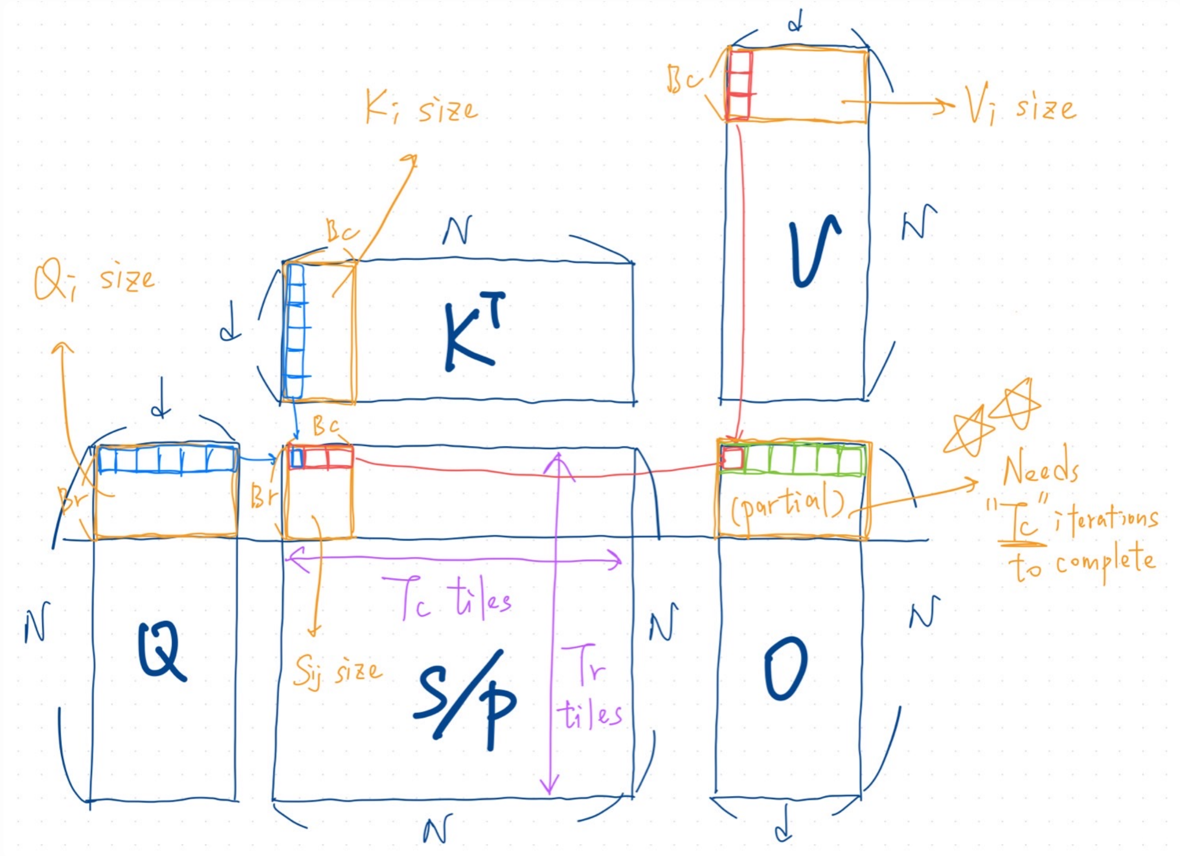


```
// Incrementally update d_O
for (int x = 0; x < d; x++) {
    float pv = 0.0f;

    #pragma unroll
    for (int j = 0; j < tile_width; j++) {
        pv += Sij[local_y * Bc + j] * Vi[j * d + x];
    }

    d_O[batch_offset + global_y * d + x] = (1 / li_new) *
        ((li * __expf(mi - mi_new) * d_O[batch_offset + global_y * d + x]) +
        __expf(mi_tilde - mi_new) * pv);
}
```

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

- Q is divided to `Tr = ceil(N / Br)`
- K, V is diviced to `Tc = ceil(N/Bc)`

After loading the shared memory, each thread can access Ki and Vi in parallel.



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

- Br: As large as possible to fully utilize the hardware.
- Bc: Not to exceed the size of sram.

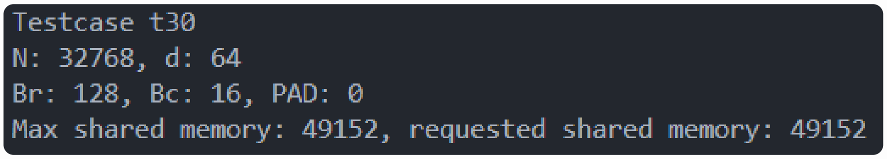
d. Specify the configurations for CUDA kernel launches, such as the number of threads per block, shared memory allocation, and grid dimensions.

- threads per block : 128
- grid dimensions : Tr * batch_size

```
/*--------------------------- Kernel launch ---------------------------*/
// grid.X: Tr, how many blocks to do
// grid.Y: B, for batch size
dim3 grid_dim(Tr, B);
// block.X: Br, how many rows in a block
dim3 block_dim(Br);

flash_attention_kernel<<<grid_dim, block_dim, sram_size>>>
    (d_Q, d_K, d_V, d_O, Bc, Br, Tc, Tr, softmax_scale);
```

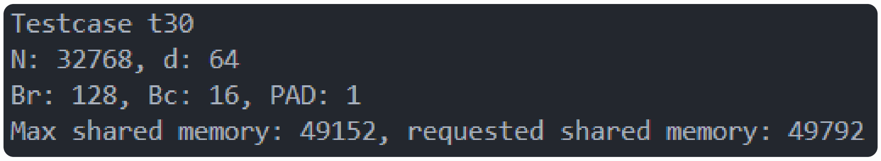
e. Justify your choices and how they relate to the blocking factors and the SRAM size.

- 128 * 16 -> pass



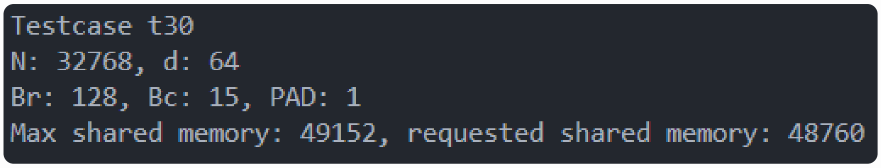
```
Testcase t30
N: 32768, d: 64
Br: 128, Bc: 16, PAD: 0
Max shared memory: 49152, requested shared memory: 49152
```

- 128 * 16 + PAD -> fail



```
Testcase t30
N: 32768, d: 64
Br: 128, Bc: 16, PAD: 1
Max shared memory: 49152, requested shared memory: 49792
```

- 128 * 15 + PAD -> pass



```
Testcase t30
N: 32768, d: 64
Br: 128, Bc: 15, PAD: 1
Max shared memory: 49152, requested shared memory: 48760
```

2. Profiling Results (hw3-2)

- Command

```
make hw4
srun -pnvidia -N1 -n1 --gres=gpu:1 \
nvprof --metrics achieved_occupancy,sm_efficiency,\
gld_throughput,gst_throughput,shared_load_throughput,shared_store_throughput \
./hw4 /share/testcases/hw4/t20 t20.out
```

- Output (flash_attention_kernel)

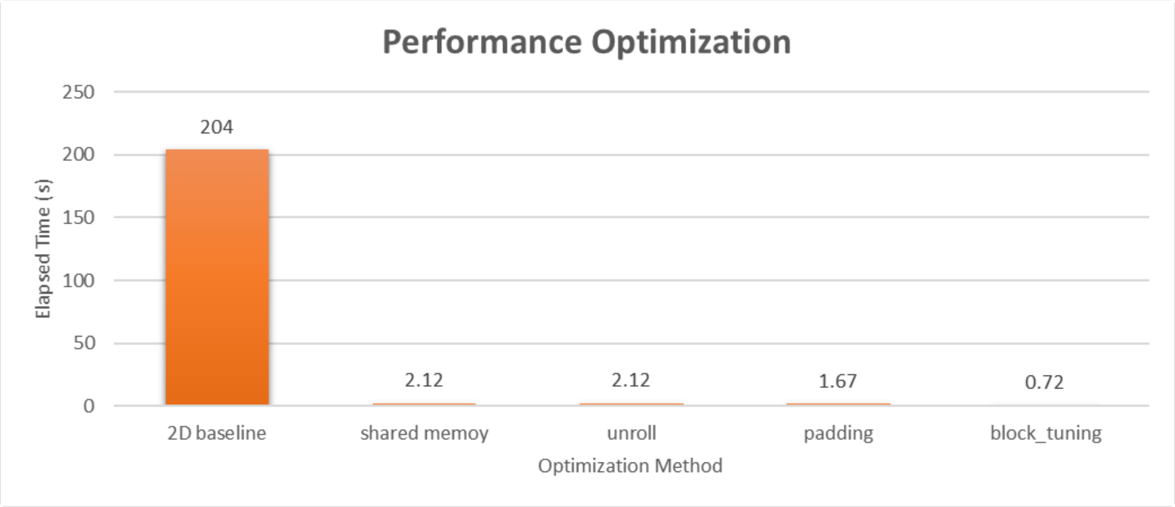
Metric	Min	Max	Average
Achieved Occupancy	0.046761	0.046761	0.046761
Multiprocessor Activity	99.68%	99.68%	99.68%
Global Load Throughput	42.192GB/s	42.192GB/s	42.192GB/s
Global Store Throughput	13.980GB/s	13.980GB/s	13.980GB/s
Shared Memory Load Throughput	3718.6GB/s	3718.6GB/s	3718.6GB/s
Shared Memory Store Throughput	168.63GB/s	168.63GB/s	168.63GB/s

3. Experiment & Analysis

a. System Spec

> Apollo-gpu workstation

b. Optimization

> - Command
>
> ```
> hw4-judge --debug -i t20
> ```
>
> 
>

4. Experience & Conclusion

Through this homework, I gained deep insights into modern GPU programming and attention mechanisms.

The implementation of FlashAttention revealed the critical importance of memory hierarchy optimization - particularly the interplay between HBM and SRAM that significantly impacts performance.Working with CUDA for matrix operations taught me practical skills in tiling, blocking, and kernel fusion techniques essential for efficient parallel computing.

The assignment demonstrated how theoretical concepts in AI algorithms translate to hardware-level optimizations, especially in handling large-scale matrix operations.The experience of tuning thread organizations, managing shared memory, and optimizing memory access patterns provided valuable lessons in balancing computational efficiency with hardware constraints.

Perhaps most importantly, this homework bridged the gap between theoretical understanding and practical implementation of state-of-the-art attention mechanisms, showing how algorithmic innovations can address hardware limitations.