Accessing Tensors (3 Points)

```
inline float fourDimRead(std::vector<float> &tensor, int &x, int &y,
                         int &z, int &b,
        const int &sizeX, const int &sizeY, const int &sizeZ) {
    // return 0.0;
    return tensor[(x * sizeX * sizeY * sizeZ) + (y * sizeY * sizeZ) +
                  (z * sizeZ) + b];
}

inline void fourDimWrite(std::vector<float> &tensor, int &x, int &y,
                         int &z, int &b,
        const int &sizeX, const int &sizeY, const int &sizeZ, float
        &val) {
    tensor[(x * sizeX * sizeY * sizeZ) + (y * sizeY * sizeZ) +
           (z * sizeZ) + b] = val;
}
```

A 4D tensor (e.g., with shape N×C×H×W) is typically stored in row-major order, meaning elements are laid out in memory such that the last dimension (W) changes fastest and the first (N) slowest. This convention was chosen to maximize performance by leveraging hardware memory access patterns and parallel computation capabilities.This layout ensures that data accessed sequentially iis stored contiguously in memory, which improves CPU cache efficiency and enables hardware-level optimizations like SIMD vectorization.

Part 1: A Simple (But Not So Efficient) Implementation of Attention (10 Points)

```
torch::Tensor myNaiveAttention(torch::Tensor QTensor, torch::Tensor
    KTensor, torch::Tensor VTensor, torch::Tensor QK_tTensor,
                int B, int H, int N, int d){

    // Q, K, V are passed in with Shape: (B, H, N, d)
    //QK^t Intermediate Tensor has Shape (N, N)

    //Make O Tensor with Shape (B, H, N, d)
    at::Tensor OTensor = at::zeros({B, H, N, d}, at::kFloat);

    //Format O, Q, K, and V tensors into 4D vectors
    std::vector<float> O = formatTensor(OTensor);
    std::vector<float> Q = formatTensor(QTensor);
    std::vector<float> K = formatTensor(KTensor);
    std::vector<float> V = formatTensor(VTensor);

    //Format QK_t Tensor into a 2D vector.
    std::vector<float> QK_t = formatTensor(QK_tTensor);
```

```
/* Here is an example of how to read/write 0's to  Q (B, H, N, d)
using the 4D accessors

//loop over Batch Size
for (int b = 0; b < B; b++) {

//loop over Heads
for (int h = 0; h < H; h++) {

//loop over Sequence Length
for (int i = 0; i < N; i++) {

//loop over Embedding Dimensionality
for (int j = 0; j < d; j++) {
float val = fourDimRead(Q, b, h, i, j, H, N,
d);
val = 0.0;
fourDimWrite(Q, b, h, i, j, H, N, d, val);
}
}
}
}
*/

/* Here is an example of how to read/write 0's to  QK_t (N, N)
using the 2D accessors

for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
float val = twoDimRead(QK_t, i, j, N);
val = 0.0;
twoDimWrite(QK_t, i, j, N, val);
}
}
*/

// -------- YOUR CODE HERE  -------- //

for (int b = 0; b < B; b++) {
for (int h = 0; h < H; h++) {

// Compute Q * K^T for this (b, h)
for (int i = 0; i < N; i++) {
```

```
for (int j = 0; j < N; j++) {
    float sum = 0.0f;
    for (int k = 0; k < d; k++) {
        float Qval = fourDimRead(Q, b, h, i, k, H, N,
d);
        float Kval = fourDimRead(K, b, h, j, k, H, N,
d); // note: K^T means index j here
        sum += Qval * Kval;
    }
    twoDimWrite(QK_t, i, j, N, sum);
}
}

// Softmax row-wise over QK_t
for (int i = 0; i < N; i++) {
    float rowSum = 0.0f;
    for (int j = 0; j < N; j++) {
        float val = twoDimRead(QK_t, i, j, N);
        val = expf(val);
        twoDimWrite(QK_t, i, j, N, val);
        rowSum += val;
    }
    for (int j = 0; j < N; j++) {
        float val = twoDimRead(QK_t, i, j, N);
        val /= rowSum;
        twoDimWrite(QK_t, i, j, N, val);
    }
}

// O = softmax(QK^T) * V
for (int i = 0; i < N; i++) {
    for (int k = 0; k < d; k++) {
        float sum = 0.0f;
        for (int j = 0; j < N; j++) {
            float QKval = twoDimRead(QK_t, i, j, N);
            float Vval = fourDimRead(V, b, h, j, k, H, N,
d);
            sum += QKval * Vval;
        }
        fourDimWrite(O, b, h, i, k, H, N, d, sum);
    }
}
}
```

```
}

// DO NOT EDIT THIS RETURN STATEMENT //
// It formats your C++ Vector O back into a Tensor of Shape (B, H,
N, d) and returns it //
return torch::from_blob(O.data(), {B, H, N, d},
torch::TensorOptions().dtype(torch::kFloat32)).clone();
}
```

Part 2: Blocked Matrix Multiply and Unfused Softmax (20 Points):

```
const int Tn = 32;
const int Td = std::min(32, d);


for (int b = 0; b < B; ++b) {
  for (int h = 0; h < H; ++h) {


    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        float zero = 0.0f;
        twoDimWrite(QK_t, i, j, N, zero);
      }
    }


    for (int i0 = 0; i0 < N; i0 += Tn) {
      int i_end = std::min(i0 + Tn, N);
      for (int j0 = 0; j0 < N; j0 += Tn) {
        int j_end = std::min(j0 + Tn, N);
        for (int k0 = 0; k0 < d; k0 += Td) {
          int k_end = std::min(k0 + Td, d);
          for (int i = i0; i < i_end; ++i) {
            for (int j = j0; j < j_end; ++j) {
              float sum = (k0 == 0)
                          ? 0.0f
                          : twoDimRead(QK_t, i, j, N);
              for (int k2 = k0; k2 < k_end; ++k2) {
                float Qv = fourDimRead(Q, b, h, i,
k2, H, N, d);
                float Kv = fourDimRead(K, b, h, j,
k2, H, N, d);
```

```
sum += Qv * Kv;
}
twoDimWrite(QK_t, i, j, N, sum);
}
}
}
}
}


for (int i = 0; i < N; ++i) {
float rowSum = 0.0f;
for (int j = 0; j < N; ++j) {
float v = twoDimRead(QK_t, i, j, N);
v = expf(v);
float expv = v;
twoDimWrite(QK_t, i, j, N, expv);
rowSum += v;
}
for (int j = 0; j < N; ++j) {
float v = twoDimRead(QK_t, i, j, N);
float norm = v / rowSum;
twoDimWrite(QK_t, i, j, N, norm);
}
}


for (int i0 = 0; i0 < N; i0 += Tn) {
int i_end = std::min(i0 + Tn, N);
for (int k0 = 0; k0 < d; k0 += Td) {
int k_end = std::min(k0 + Td, d);
for (int j0 = 0; j0 < N; j0 += Tn) {
int j_end = std::min(j0 + Tn, N);
for (int i = i0; i < i_end; ++i) {
for (int j = j0; j < j_end; ++j) {
float Pval = twoDimRead(QK_t, i, j, N);
for (int k2 = k0; k2 < k_end; ++k2) {
float Vv = fourDimRead(V, b, h, j,
k2, H, N, d);
float acc = fourDimRead(O, b, h, i,
k2, H, N, d);
acc += Pval * Vv;
float updated = acc;
```

```
                                    fourDimWrite(O, b, h, i, k2, H, N,
                              d, updated);
                                                }
                                              }
                                            }
                                          }
                                        }
                                      }
                                    }
                                  }
```

N-block size (T□)	Time (ms)
16	138
32	116.7
64	125
128	140

Optimal Tile size was 32 as at 32x32 blocks each Q block, K block and C Block is a 32x32 submatrix so the working per block is nearly 12KB, which fits easily in L1 cache.

In a N×d × d×N matrix multiply you end up reading each Q[i,k] and each K[k,j] element N times, so you pull $2 \cdot N^2 \cdot d$ floats from DRAM. A blocked implementation reuses each Q[i,k] across all j in its block and each K[k,j] across all i in its block, so you load each Q and each K element oncel. As a result, the blocked version requires only 1/N as many DRAM reads as the naïve one, cutting memory traffic by a factor of N) and dramatically improving performance.

Part 3: Fused Attention (25 Points):

```
inline int index4D(int b, int h, int i, int k, int H, int N, int d) {
    return ((b * H + h) * N + i) * d + k;
}

torch::Tensor myFusedAttention(torch::Tensor QTensor, torch::Tensor
KTensor, torch::Tensor VTensor, torch::Tensor temp,
                int B, int H, int N, int d){

    // Q, K, V are passed in with Shape: (B, H, N, d)
```

```
    //Make O Tensor with Shape (B, H, N, d)
    //and O Row Tensor with Shape (N)
    at::Tensor OTensor = at::zeros({B, H, N, d}, at::kFloat);
    at::Tensor ORowTensor = at::zeros({N}, at::kFloat);

    //Format Y, Q, K, and V tensors into 4D vectors
    std::vector<float> O = formatTensor(OTensor);
    std::vector<float> Q = formatTensor(QTensor);
    std::vector<float> K = formatTensor(KTensor);
    std::vector<float> V = formatTensor(VTensor);

    //Format ORow Tensor into a 1D vector
    // You can simply access this as ORow[i]
    std::vector<float> ORow = formatTensor(ORowTensor);


    // -------- YOUR CODE HERE  -------- //
    // We give you a template of the first three loops for your
convenience
    #pragma omp parallel for collapse(3)
    //loop over batch
    for (int b = 0; b < B; b++){
        //loop over heads
        for (int h = 0; h < H; h++){
            for (int i = 0; i < N ; i++){

        // YRow is moved inside so each OpenMP thread gets a local
copy.
                at::Tensor ORowTensor =
temp.index({torch::indexing::Slice(omp_get_thread_num(),
torch::indexing::None)});
                std::vector<float> ORow = formatTensor(ORowTensor);
        //YOUR CODE HERE
        // Q[i] . K[j]
                for(int j=0;j<N;j++){
                    float dot=0.0;
                    for(int k=0;k<d;k++){

dot+=Q[index4D(b,h,i,k,H,N,d)]*K[index4D(b,h,j,k,H,N,d)];

                    }
                    ORow[j]=dot;
```

```
                }

                float
maxVal=*std::max_element(ORow.begin(),ORow.begin()+N);
                float sum=0.0;
                for(int j=0;j<N;j++){
                    ORow[j]=std::exp(ORow[j]-maxVal);
                    sum+=ORow[j];
                }

                for(int j=0;j<N;j++){
                    ORow[j]/=sum;
                }

                //softmax
                for(int k=0;k<d;k++){
                    float val=0.0;
                    for(int j=0;j<N;j++){
                        val+=ORow[j]*V[index4D(b,h,j,k,H,N,d)];
                    }
                    O[index4D(b,h,i,k,H,N,d)]=val;
                }

            }
        }
    }
```

We use much less memory in Part 3 because we don't store the full attention matrix. Instead, we compute and apply attention scores on-the-fly for each output position, reducing memory from quadratic to linear in sequence length `N`.

Reference Implementation (Manual Execution Time):

- With `#pragma omp`: ~97.3 ms
- Without `#pragma omp`: ~281.9 ms

Student Implementation (Manual Execution Time):

- With `#pragma omp`: ~71.6 ms
- Without `#pragma omp`: ~266.7 ms

Fused attention combines multiple operations into a single GPU kernel, reducing the need for intermediate memory reads/writes and synchronization between separate threads. In Part

1, each operation was handled independently, causing thread stalls and inefficient memory usage. Fused attention allows for better thread cooperation within the kernel, maximizing parallel execution and minimizing overhead, thus making it much easier to fully utilize multithreading capabilities on modern GPUs.

Part 4 : Putting it all Together - Flash Attention (35 Points):

```
// -------------------------------------------------------------- //
//                PART 4: FLASH ATTENTION              //
// -------------------------------------------------------------- //

torch::Tensor myFlashAttention(
    // Q, K, V are passed in with Shape: (B, H, N, d)
    // Sij, Pij are passed in with Shape: (Br, Bc)
    // Kj, Vj are passed in with Shape: (Bc, d)
    // Qi, Oi, and PV  are passed in with Shape: (Br, d)
    // L in passed in with Shape: (N)
    // Li, Lij, and Lnew are passed in with shape (Br)

    //Make O Tensor with Shape (B, H, N, d)
    torch::Tensor QTensor, torch::Tensor KTensor, torch::Tensor
VTensor,
    torch::Tensor /*QiTensor*/, torch::Tensor /*KjTensor*/,
torch::Tensor /*VjTensor*/,
    torch::Tensor /*SijTensor*/, torch::Tensor /*PijTensor*/,
torch::Tensor /*PVTensor*/,
    torch::Tensor /*OiTensor*/, torch::Tensor /*LTensor*/,
torch::Tensor /*LiTensor*/,
    torch::Tensor /*LijTensor*/, torch::Tensor /*LnewTensor*/,
    int Bc, int Br, int B, int H, int N, int d) {

    //Format All Tensors into Vectors
    at::Tensor OTensor = at::zeros({B, H, N, d}, at::kFloat);
    std::vector<float> O = formatTensor(OTensor);
    std::vector<float> Q = formatTensor(QTensor);
    std::vector<float> K = formatTensor(KTensor);
    std::vector<float> V = formatTensor(VTensor);


    std::vector<float> Qi(Br * d), Kj(Bc * d), Vj(Bc * d);
    std::vector<float> Sij(Br * Bc), Pij(Br * Bc);
    std::vector<float> PV(Br * d), Oi(Br * d);
```

```
    std::vector<float> li(Br), Lij(Br), lnew(Br);
    std::vector<float> PBsum(Br);

    // -------- YOUR CODE HERE  -------- //
    for (int b = 0; b < B; ++b) {
        for (int h = 0; h < H; ++h) {
            for (int i = 0; i < N; i += Br) {

                for (int bi = 0; bi < Br; ++bi) {
                    li[bi]   = -INFINITY;
                    lnew[bi] = 0.0f;
                    for (int k = 0; k < d; ++k) {
                        Oi[bi * d + k] = 0.0f;
                    }
                }

                for (int j = 0; j < N; j += Bc) {

                    for (int bi = 0; bi < Br; ++bi) {
                        int row = i + bi;
                        if (row >= N) break;
                        for (int k = 0; k < d; ++k) {
                            Qi[bi * d + k] = Q[index4D(b, h, row, k, H,
N, d)];
                        }
                    }

                    for (int bj = 0; bj < Bc; ++bj) {
                        int col = j + bj;
                        if (col >= N) break;
                        for (int k = 0; k < d; ++k) {
                            Kj[bj * d + k] = K[index4D(b, h, col, k, H,
N, d)];
                            Vj[bj * d + k] = V[index4D(b, h, col, k, H,
N, d)];
                        }
                    }


                    for (int bi = 0; bi < Br; ++bi) {
                        int row = i + bi;
                        if (row >= N) break;
                        for (int bj = 0; bj < Bc; ++bj) {
```

```
                int col = j + bj;
                if (col >= N) break;
                float sum = 0.0f;
                for (int k = 0; k < d; ++k) {
                    sum += Qi[bi * d + k] * Kj[bj * d + k];
                }
                Sij[bi * Bc + bj] = sum;
            }
        }


        for (int bi = 0; bi < Br; ++bi) {
            int row = i + bi;
            if (row >= N) break;
            float m = -INFINITY;
            for (int bj = 0; bj < Bc; ++bj) {
                int col = j + bj;
                if (col >= N) break;
                m = std::max(m, Sij[bi * Bc + bj]);
            }
            Lij[bi] = m;
        }


        for (int bi = 0; bi < Br; ++bi) {
            int row = i + bi;
            if (row >= N) break;
            float sumExp = 0.0f;
            for (int bj = 0; bj < Bc; ++bj) {
                int col = j + bj;
                if (col >= N) break;
                float v = std::exp(Sij[bi * Bc + bj] -
Lij[bi]);

                Pij[bi * Bc + bj] = v;
                sumExp += v;
            }
            PBsum[bi] = sumExp;
        }


        for (int bi = 0; bi < Br; ++bi) {
            int row = i + bi;
            if (row >= N) break;
```

```
                        for (int k = 0; k < d; ++k) {
                            float acc = 0.0f;
                            for (int bj = 0; bj < Bc; ++bj) {
                                int col = j + bj;
                                if (col >= N) break;
                                acc += Pij[bi * Bc + bj] * Vj[bj * d +
k];

                            }
                            PV[bi * d + k] = acc;
                        }
                    }


                    for (int bi = 0; bi < Br; ++bi) {
                        int row = i + bi;
                        if (row >= N) break;
                        float old_li = li[bi];
                        float new_li = Lij[bi];
                        float alpha = std::exp(old_li - new_li);


                        lnew[bi] = alpha * lnew[bi] + PBsum[bi];


                        for (int k = 0; k < d; ++k) {
                            Oi[bi * d + k] = alpha * Oi[bi * d + k] +
PV[bi * d + k];
                        }


                        li[bi] = new_li;
                    }
                }


                for (int bi = 0; bi < Br; ++bi) {
                    int row = i + bi;
                    if (row >= N) break;
                    for (int k = 0; k < d; ++k) {
                        int idx = index4D(b, h, row, k, H, N, d);
                        O[idx] = Oi[bi * d + k] / lnew[bi];
                    }
                }
```

```
            }
        }
    }



    // DO NOT EDIT THIS RETURN STATEMENT //
    // It formats your C++ Vector O back into a Tensor of Shape (B, H,
N, d) and returns it //
    return torch::from_blob(O.data(), {B, H, N, d},
torch::TensorOptions().dtype(torch::kFloat32)).clone();
}
```

In Parts 1 and 2, we build and store the full attention score matrix QK^T of size N×N for each batch and head, which incurs $O(N^2)$ extra memory just to hold those intermediate scores. In Part 3, by fusing computation one row at a time, we only materialize a single length-N buffer, reducing the scratch space to O(N). Flash Attention in Part 4, streams Q, K, and V through small Br×Bc tiles, holding at most one tile's worth of scores and related accumulators in SRAM at any moment. As a result, aside from the output tensor and O(N) running softmax state, Flash Attention never needs more than O(1) extra per tile, so its auxiliary memory scales linearly in N rather than quadratically.

No,our straight C++ tiling is functionally correct but not optimised. A few key optimizations:

- **Parallelize with OpenMP (or TBB):** distribute row- and head-blocks across threads to utilize all cores.
- **SIMD-vectorize inner loops:** use AVX/AVX-512 intrinsics (or compiler pragmas) for the dot-products and the exp() calls to process multiple floats per instruction.
- **Software-prefetching and alignment:** prefetch the next Q/K/V tiles into L1/L2 and align your block buffers on 64-byte boundaries to avoid cache misses.