Question 0:

不理解，请解释，用例子

```
#include <cuda_runtime.h>
#include <math.h>

// feature map phi(x) = ELU(x) + 1
__device__ __forceinline__ float phi_fn(float x) {
    // ELU(x) = x  (x > 0)
    //        = exp(x) - 1 (x <= 0)
    // phi(x) = ELU(x) + 1
    //        = x + 1 (x > 0)
    //        = exp(x) (x <= 0)
    return (x > 0.0f) ? (x + 1.0f) : expf(x);
}

// Apply phi to K -> Kp
__global__ void kernel_apply_phi_K(const float* __restrict__ K,
                                   float* __restrict__ Kp,
                                   int M, int d) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int n = M * d;
    if (idx >= n) return;
    float x = K[idx];
    Kp[idx] = phi_fn(x);
}

// k_sum[t] = sum_j Kp[j, t]
__global__ void kernel_k_sum(const float* __restrict__ Kp,
                             float* __restrict__ k_sum,
                             int M, int d) {
    int t = blockIdx.x * blockDim.x + threadIdx.x;
    if (t >= d) return;

    float acc = 0.0f;
    for (int j = 0; j < M; ++j) {
        acc += Kp[j * d + t];
    }
    k_sum[t] = acc;
}

// kv[r, c] = sum_j Kp[j, r] * V[j, c]
// kv has shape (d x d)
__global__ void kernel_kv(const float* __restrict__ Kp,
                          const float* __restrict__ V,
                          float* __restrict__ kv,
                          int M, int d) {
    int c = blockIdx.x * blockDim.x + threadIdx.x; // column in V / kv
    int r = blockIdx.y * blockDim.y + threadIdx.y; // row in Kp / kv
```

```
    if (c >= d || r >= d) return;

    float acc = 0.0f;
    for (int j = 0; j < M; ++j) {
        float k_val = Kp[j * d + r]; // Kp[j, r]
        float v_val = V[j * d + c];  // V[j, c]
        acc += k_val * v_val;
    }
    kv[r * d + c] = acc;
}

// Compute Qp = phi(Q) and denom[i] = Qp[i,:] · k_sum
__global__ void kernel_Qp_and_denom(const float* __restrict__ Q,
                                    const float* __restrict__ k_sum,
                                    float* __restrict__ Qp,
                                    float* __restrict__ denom,
                                    int M, int d) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // row index
    if (i >= M) return;

    float dacc = 0.0f;
    int base = i * d;
    for (int r = 0; r < d; ++r) {
        float x = Q[base + r];
        float qphi = phi_fn(x);
        Qp[base + r] = qphi;
        dacc += qphi * k_sum[r];
    }
    denom[i] = dacc;
}

// output[i, c] = (Qp[i,:] @ kv[:,c]) / denom[i]
__global__ void kernel_output(const float* __restrict__ Qp,
                              const float* __restrict__ kv,
                              const float* __restrict__ denom,
                              float* __restrict__ output,
                              int M, int d) {
    int c = blockIdx.x * blockDim.x + threadIdx.x; // column
    int i = blockIdx.y * blockDim.y + threadIdx.y; // row
    if (c >= d || i >= M) return;

    int base_row = i * d;
    float acc = 0.0f;
    for (int r = 0; r < d; ++r) {
        acc += Qp[base_row + r] * kv[r * d + c];
    }

    float den = denom[i];
    // den should be strictly > 0 with this feature map, but guard anyway
    if (den == 0.0f) den = 1e-6f;

    output[base_row + c] = acc / den;
}
```

```
// Q, K, V, output are device pointers
extern "C" void solve(const float* Q, const float* K, const float* V,
                      float* output, int M, int d) {
    const int Md = M * d;
    const int dd = d * d;

    float *Kp = nullptr, *Qp = nullptr;
    float *k_sum = nullptr, *kv = nullptr, *denom = nullptr;

    cudaMalloc(&Kp,    Md * sizeof(float));
    cudaMalloc(&Qp,    Md * sizeof(float));
    cudaMalloc(&k_sum, d  * sizeof(float));
    cudaMalloc(&kv,    dd * sizeof(float));
    cudaMalloc(&denom, M  * sizeof(float));

    // 1) phi(K) -> Kp
    {
        int block = 256;
        int grid  = (Md + block - 1) / block;
        kernel_apply_phi_K<<<grid, block>>>(K, Kp, M, d);
    }

    // 2) k_sum over rows of Kp
    {
        int block = 128;
        int grid  = (d + block - 1) / block;
        kernel_k_sum<<<grid, block>>>(Kp, k_sum, M, d);
    }

    // 3) kv = Kp^T V
    {
        dim3 block(16, 16);
        dim3 grid((d + block.x - 1) / block.x,
                  (d + block.y - 1) / block.y);
        kernel_kv<<<grid, block>>>(Kp, V, kv, M, d);
    }

    // 4) Qp = phi(Q), denom[i] = Qp[i,:] · k_sum
    {
        int block = 256;
        int grid  = (M + block - 1) / block;
        kernel_Qp_and_denom<<<grid, block>>>(Q, k_sum, Qp, denom, M, d);
    }

    // 5) final output
    {
        dim3 block(16, 16);
        dim3 grid((d + block.x - 1) / block.x,
                  (M + block.y - 1) / block.y);
        kernel_output<<<grid, block>>>(Qp, kv, denom, output, M, d);
    }

    cudaDeviceSynchronize();
```

```
    cudaFree(Kp);
    cudaFree(Qp);
    cudaFree(k_sum);
    cudaFree(kv);
    cudaFree(denom);
}
```

不理解，请解释，用例子

```
#include <cuda_runtime.h>
#include <math.h>
...
```

这是一个实现了 **线性化注意力机制 (Linear Attention)** 的 CUDA C++ 代码。这种机制旨在解决标准 Self-Attention 复杂度过高（$O(N^2)$）的问题，将其降为 **线性复杂度** $O(N)$，从而在处理长序列（如 Transformer）时显著提高速度。

代码通过 **核函数（Kernel）** 的流水线（Pipeline）来并行执行线性注意力公式的各个步骤。

---

1. 线性注意力 (Linear Attention) 原理

标准注意力公式：

$$\text{Attention}(Q, K, V) = \text{Softmax}(QK^T)V$$

线性注意力使用一个非负特征映射函数 $\phi(\cdot)$ 来近似 Softmax 核心，从而可以交换求和顺序。

我们定义 $\phi(X) = \text{ELU}(X) + 1$ (代码中简化为 $x + 1$ 或 e^x):

$$\text{Linear Attention} = \frac{\phi(Q)(\phi(K)^T V)}{\phi(Q)(\phi(K)^T \mathbf{1})}$$

其中，$\phi(Q)$ 和 $\phi(K)$ 是应用了特征映射的 Q 和 K 矩阵，$\mathbf{1}$ 是全 1 向量。

为了计算这个公式，代码采用了**分解计算**：

$$\text{Output} = \frac{\phi(Q) \cdot (\phi(K)^T V)}{\phi(Q) \cdot (\phi(K)^T \mathbf{1})} = \frac{Q_p \cdot \mathbf{KV}}{\mathbf{Denom}}$$

$$\text{Output} = \frac{\phi(Q) \cdot (\phi(K)^T V)}{\phi(Q) \cdot (\phi(K)^T \mathbf{1})} = \frac{Q_p \cdot \mathbf{KV}}{\mathbf{Denom}}$$

代码的五个 Kernel 步骤对应于计算：

1. $\phi(K) = K_p$
2. $K_{\text{sum}} = \phi(K)^T \mathbf{1}$ (即 K_p 按行求和)
3. $\mathbf{KV} = K_p^T V$
4. $\phi(Q) = Q_p$ 和 $\mathbf{Denom} = Q_p \cdot K_{\text{sum}}$ (即 Q_p 乘以 K_{sum})
5. $\text{Output} = Q_p \cdot \mathbf{KV} / \mathbf{Denom}$

---

2. Kernel 代码解释 (Pipeline 流程)

A. 通用工具函数 (`phi_fn`)

C++

```
__device__ __forceinline__ float phi_fn(float x) {
    // ... 实际计算 phi(x) = x + 1 (x > 0) 或 exp(x) (x <= 0)
    return (x > 0.0f) ? (x + 1.0f) : expf(x);
}
```

- **目的:** 实现特征映射函数 $\phi(\cdot)$。这是逐元素的。

B. Kernel 1: $\phi(K)$

C++

```
__global__ void kernel_apply_phi_K(...) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // ...
    Kp[idx] = phi_fn(K[idx]); // 逐元素应用 phi 函数
}
```

- **并行策略:** 每个线程负责 K 矩阵中的一个元素。
- **计算:** 将 ϕ 应用于 K，结果存储在 K_p (`Kp`) 中。

C. Kernel 2: K_{sum} (Denominator Component 1)

C++

```
__global__ void kernel_k_sum(const float* __restrict__ Kp, float* __restrict__ k_su
    int t = blockIdx.x * blockDim.x + threadIdx.x; // 索引 t 对应 Kp 的列
    if (t >= d) return;

    float acc = 0.0f;
    for (int j = 0; j < M; ++j) { // 串行循环 M 次，沿着列 t 求和
        acc += Kp[j * d + t]; // Kp[j, t]
    }
    k_sum[t] = acc; // k_sum[t] 存储 Kp 的第 t 列的总和
}
```

- **并行策略:** 每个线程负责计算 K_p 的**一整列**的总和。
- **计算:** K_{sum} 向量的长度为 d。线程 T_t 串行遍历 M 行，计算 K_p 第 t 列所有元素的和。

D. Kernel 3: $\mathbf{KV}$ ($K_p^T V$)

C++

```
__global__ void kernel_kv(...) {
    int c = blockIdx.x * blockDim.x + threadIdx.x; // kv 的列 (V 的列)
    int r = blockIdx.y * blockDim.y + threadIdx.y; // kv 的行 (Kp 的列)
    if (c >= d || r >= d) return;

    float acc = 0.0f;
    for (int j = 0; j < M; ++j) { // 串行循环 M 次
```

D. Kernel 3: $\mathbf{KV}$ ($K_p^T V$)

C++

```
__global__ void kernel_kv(...) {
    int c = blockIdx.x * blockDim.x + threadIdx.x; // kv 的列 (V 的列)
    int r = blockIdx.y * blockDim.y + threadIdx.y; // kv 的行 (Kp 的列)
    if (c >= d || r >= d) return;

    float acc = 0.0f;
    for (int j = 0; j < M; ++j) { // 串行循环 M 次
        float k_val = Kp[j * d + r]; // Kp[j, r]
        float v_val = V[j * d + c];  // V[j, c]
        acc += k_val * v_val;
    }
    kv[r * d + c] = acc; // 结果 kv[r, c]
}
```

- **并行策略:** 使用 2D 网格，每个线程负责计算 $d \times d$ 输出矩阵 kv 中的一个元素。
- **计算:** 这是矩阵乘法 $K_p^T \times V$。K_p 尺寸 $M \times d$，V 尺寸 $M \times d$。K_p^T 尺寸 $d \times M$。输出 kv 尺寸 $d \times d$。
  - **核心:** $kv_{r,c} = \sum_j K_{p_{j,r}} \cdot V_{j,c}$ (注意 K_p 的行索引和 V 的行索引 j 是共享的)。

E. Kernel 4: Q_p 和 $\mathbf{Denom}$ (Denominator Component 2)

C++

```
__global__ void kernel_Qp_and_denom(...) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // 行索引
    if (i >= M) return;

    float dacc = 0.0f;
    int base = i * d;
    for (int r = 0; r < d; ++r) { // 串行循环 d 次
        float x = Q[base + r];
```

C++

```
__global__ void kernel_Qp_and_denom(...) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // 行索引
    if (i >= M) return;

    float dacc = 0.0f;
    int base = i * d;
    for (int r = 0; r < d; ++r) { // 串行循环 d 次
        float x = Q[base + r];
        float qphi = phi_fn(x);
        Qp[base + r] = qphi; // 1. 计算 Qp = phi(Q)
        dacc += qphi * k_sum[r]; // 2. 计算分母的中间项
    }
    denom[i] = dacc; // 存储分母向量的第 i 个元素
}
```

- **并行策略:** 每个线程负责计算 Q 矩阵中的一整行。
- **计算:**
  1. 逐行计算 $Q_p = \phi(Q)$。
  2. 同时，计算分母向量 $\mathbf{Denom}$ 的每个元素 $\mathbf{Denom}_i = Q_{p_{i,:}} \cdot K_{\text{sum}}$ (点积)。

F. Kernel 5: 最终输出 (Output)

C++

```
__global__ void kernel_output(...) {
    int c = blockIdx.x * blockDim.x + threadIdx.x; // 列
    int i = blockIdx.y * blockDim.y + threadIdx.y; // 行
    if (c >= d || i >= M) return;

    float acc = 0.0f;
    for (int r = 0; r < d; ++r) { // 串行循环 d 次
        acc += Qp[base_row + r] * kv[r * d + c]; // 计算分子 Qp · KV
    }

    float den = denom[i];
```

F. Kernel 5: 最终输出 (Output)

C++

```
__global__ void kernel_output(...) {
    int c = blockIdx.x * blockDim.x + threadIdx.x; // 列
    int i = blockIdx.y * blockDim.y + threadIdx.y; // 行
    if (c >= d || i >= M) return;

    float acc = 0.0f;
    for (int r = 0; r < d; ++r) { // 串行循环 d 次
        acc += Qp[base_row + r] * kv[r * d + c]; // 计算分子 Qp · KV
    }

    float den = denom[i];
    // ... 边界检查 ...

    output[base_row + c] = acc / den; // 最终结果
}
```

- **并行策略:** 使用 2D 网格，每个线程计算最终输出矩阵 Output 中的一个元素。
- **计算:**
  1. 计算分子：$\text{Numerator}_{i,c} = Q_{p_{i,:}} \cdot \mathbf{KV}_{:,c}$。
  2. 除以分母：$\text{Output}_{i,c} = \text{Numerator}_{i,c} / \mathbf{Denom}_i$。

---

3. 示例追踪 (Kernel 3: $K_p^T V$)

假设 $M = 4$（序列长度），$d = 2$（特征维度）。K_p 和 V 都是 4×2 矩阵。

输出 kv 是 2×2 矩阵。我们追踪线程 $T_{0,0}$，它计算 $kv_{0,0}$。

步骤	变量/操作	结果	含义
线程 ID	$r = 0, c = 0$		目标：$kv_{0,0}$
循环 $j = 0$	k_{val}, v_{val}	$K_{p_{0,0}}, V_{0,0}$	K_p 第 0 行第 0 列; V 第 0 行第 0 列

2. 除以分母：$\text{Output}_{i,c} = \text{Numerator}_{i,c} / \mathbf{Denom}_i$。

---

3. 示例追踪 (Kernel 3: $K_p^T V$)

假设 $M = 4$（序列长度），$d = 2$（特征维度）。K_p 和 V 都是 4×2 矩阵。

输出 kv 是 2×2 矩阵。我们追踪线程 $T_{\mathbf{0,0}}$，它计算 $kv_{\mathbf{0,0}}$。

步骤	变量/操作	结果	含义
线程 ID	$r = 0, c = 0$		目标：$kv_{0,0}$
循环 $j = 0$	k_{val}, v_{val}	$K_{p_{0,0}}, V_{0,0}$	K_p 第 0 行第 0 列; V 第 0 行第 0 列
循环 $j = 1$	k_{val}, v_{val}	$K_{p_{1,0}}, V_{1,0}$	K_p 第 1 行第 0 列; V 第 1 行第 0 列
循环 $j = 2$	k_{val}, v_{val}	$K_{p_{2,0}}, V_{2,0}$	
循环 $j = 3$	k_{val}, v_{val}	$K_{p_{3,0}}, V_{3,0}$	

匯出到試算表

计算公式展开：

$$kv_{0,0} = (K_{p_{0,0}} V_{0,0}) + (K_{p_{1,0}} V_{1,0}) + (K_{p_{2,0}} V_{2,0}) + (K_{p_{3,0}} V_{3,0})$$

这正是 K_p 的第 0 列与 V 的第 0 列的点积，实现了 $K_p^T V$ 乘法的 $(0, 0)$ 元素。

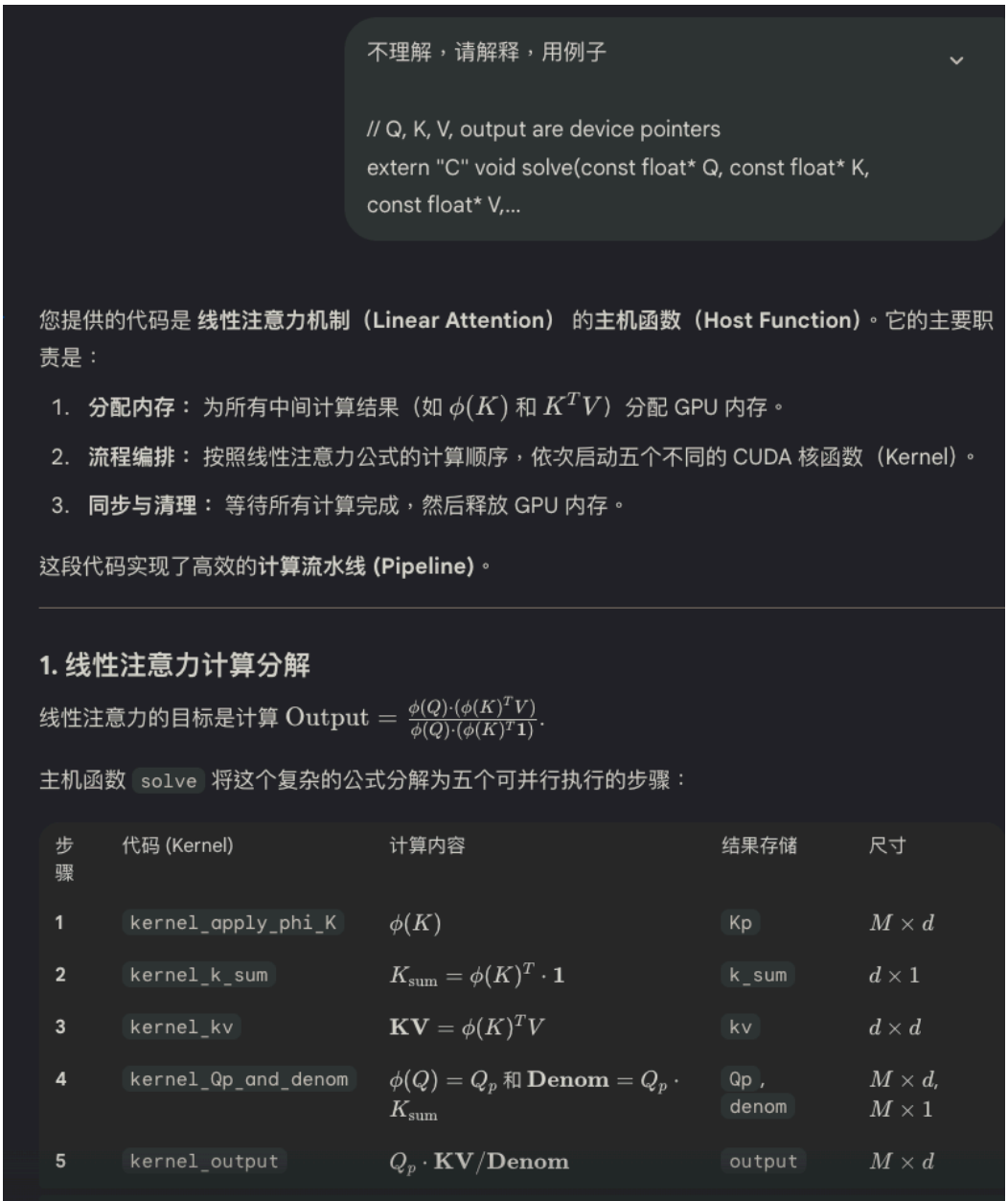
结论: 这五个 Kernel 构成了高效的线性注意力计算流水线。通过将大型矩阵乘法分解为多个 $O(M \cdot d)$ 或 $O(d^3)$ 的中间步骤，它将整体复杂度保持在 $O(N \cdot d)$（其中 $N = M$ 是序列长度），实现了线性扩展。

Question 1:

不理解，请解释，用例子

```
// Q, K, V, output are device pointers
extern "C" void solve(const float* Q, const float* K, const float* V,
                      float* output, int M, int d) {
    const int Md = M * d;
    const int dd = d * d;

    float *Kp = nullptr, *Qp = nullptr;
    float *k_sum = nullptr, *kv = nullptr, *denom = nullptr;

    cudaMalloc(&Kp,    Md * sizeof(float));
    cudaMalloc(&Qp,    Md * sizeof(float));
    cudaMalloc(&k_sum, d  * sizeof(float));
    cudaMalloc(&kv,    dd * sizeof(float));
    cudaMalloc(&denom, M  * sizeof(float));

    // 1) phi(K) -> Kp
    {
        int block = 256;
        int grid  = (Md + block - 1) / block;
        kernel_apply_phi_K<<<grid, block>>>(K, Kp, M, d);
    }

    // 2) k_sum over rows of Kp
    {
        int block = 128;
        int grid  = (d + block - 1) / block;
        kernel_k_sum<<<grid, block>>>(Kp, k_sum, M, d);
    }

    // 3) kv = Kp^T V
    {
        dim3 block(16, 16);
        dim3 grid((d + block.x - 1) / block.x,
                  (d + block.y - 1) / block.y);
        kernel_kv<<<grid, block>>>(Kp, V, kv, M, d);
    }

    // 4) Qp = phi(Q), denom[i] = Qp[i,:] · k_sum
    {
        int block = 256;
        int grid  = (M + block - 1) / block;
```

```
        kernel_Qp_and_denom<<<grid, block>>>(Q, k_sum, Qp, denom, M, d);
    }

    // 5) final output
    {
        dim3 block(16, 16);
        dim3 grid((d + block.x - 1) / block.x,
                  (M + block.y - 1) / block.y);
        kernel_output<<<grid, block>>>(Qp, kv, denom, output, M, d);
    }

    cudaDeviceSynchronize();

    cudaFree(Kp);
    cudaFree(Qp);
    cudaFree(k_sum);
    cudaFree(kv);
    cudaFree(denom);
}
```



不理解，请解释，用例子

// Q, K, V, output are device pointers
extern "C" void solve(const float* Q, const float* K,
const float* V,...

您提供的代码是 **线性注意力机制（Linear Attention）** 的**主机函数（Host Function）**。它的主要职责是：

1. **分配内存：** 为所有中间计算结果（如 $\phi(K)$ 和 K^TV）分配 GPU 内存。
2. **流程编排：** 按照线性注意力公式的计算顺序，依次启动五个不同的 CUDA 核函数（Kernel）。
3. **同步与清理：** 等待所有计算完成，然后释放 GPU 内存。

这段代码实现了高效的**计算流水线 (Pipeline)**。

---

1. 线性注意力计算分解

线性注意力的目标是计算 $\text{Output} = \frac{\phi(Q) \cdot (\phi(K)^T V)}{\phi(Q) \cdot (\phi(K)^T \mathbf{1})}$.

主机函数 `solve` 将这个复杂的公式分解为五个可并行执行的步骤：

步骤	代码 (Kernel)	计算内容	结果存储	尺寸
1	`kernel_apply_phi_K`	$\phi(K)$	`Kp`	$M \times d$
2	`kernel_k_sum`	$K_{\text{sum}} = \phi(K)^T \cdot \mathbf{1}$	`k_sum`	$d \times 1$
3	`kernel_kv`	$\mathbf{KV} = \phi(K)^T V$	`kv`	$d \times d$
4	`kernel_Qp_and_denom`	$\phi(Q) = Q_p$ 和 $\mathbf{Denom} = Q_p \cdot K_{\text{sum}}$	`Qp`, `denom`	$M \times d$, $M \times 1$
5	`kernel_output`	$Q_p \cdot \mathbf{KV} / \mathbf{Denom}$	`output`	$M \times d$

2. 代码逐行解释：流程与内存管理

A. 内存分配 (Memory Allocation)

C++

```
const int Md = M * d; // M x d 矩阵的大小
const int dd = d * d; // d x d 矩阵的大小 (kv)

// 分配 GPU 内存 (指针是 device pointers)
cudaMalloc(&Kp,    Md * sizeof(float)); // phi(K)
cudaMalloc(&Qp,    Md * sizeof(float)); // phi(Q)
cudaMalloc(&k_sum, d  * sizeof(float)); // 列总和向量
cudaMalloc(&kv,    dd * sizeof(float)); // Kp^T V 结果
cudaMalloc(&denom, M  * sizeof(float)); // 分母向量
```

- **目的：** 在 GPU 的全局内存中，为所有中间计算结果分配空间。这是 GPU 计算的第一步。
- **参数：** M 是序列长度（行数），d 是特征维度（列数）。

B. 阶段调度与核函数启动 (Kernel Launch)

主机代码主体由五个独立的 Block 组成，每个 Block 启动一个 Kernel，并设置其 Grid 和 Block 尺寸。

1) 步骤 1：计算 $\phi(K)$

C++

```
// 1) phi(K) -> Kp
{
    int block = 256;
    int grid  = (Md + block - 1) / block; // 计算 Md 个元素的 Grid
    kernel_apply_phi_K<<<grid, block>>>(K, Kp, M, d);
}
```

- **计算内容：** 将 ϕ 函数逐元素应用于输入 K 矩阵。
- **并行策略：** 采用一维 Grid/Block，确保每个线程处理 $M \times d$ 矩阵中的一个元素。

2) 步骤 2：计算 K_{sum}

C++

```
// 2) k_sum over rows of Kp
{
    int block = 128;
    int grid  = (d + block - 1) / block; // 计算 d 个元素的 Grid
    kernel_k_sum<<<grid, block>>>(Kp, k_sum, M, d);
}
```

- **计算内容：** 计算 K_p 矩阵的每一列的总和，得到长度为 d 的向量 `k_sum`。
- **并行策略：** 每个线程负责计算向量 `k_sum` 中的一个元素。

3) 步骤 3：计算 $\mathbf{KV}$ ($K_p^T V$)

C++

```
// 3) kv = Kp^T V
{
    dim3 block(16, 16);
    dim3 grid((d + block.x - 1) / block.x, (d + block.y - 1) / block.y); // 2D
    kernel_kv<<<grid, block>>>(Kp, V, kv, M, d);
}
```

- **计算内容：** 执行矩阵乘法 $K_p^T V$。
- **并行策略：** 使用 2D Grid，确保每个线程处理 $d \times d$ 输出矩阵 `kv` 中的一个元素。

- **并行策略：** 使用 2D Grid，确保每个线程处理 $d \times d$ 输出矩阵 `kv` 中的一个元素。

4) 步骤 4：计算 Q_p 和 $\mathbf{Denom}$

C++

```
// 4) Qp = phi(Q), denom[i] = Qp[i,:] · k_sum
{
    int block = 256;
    int grid  = (M + block - 1) / block; // 计算 M 行的 Grid
    kernel_Qp_and_denom<<<grid, block>>>(Q, k_sum, Qp, denom, M, d);
}
```

- **计算内容：** 同时完成两项任务：将 ϕ 应用于 Q，并计算分母项 $\mathbf{Denom}$。
- **并行策略：** 每个线程负责处理 Q 矩阵中的一整行。

5) 步骤 5：最终输出

C++

```
// 5) final output
{
    dim3 block(16, 16);
    dim3 grid((d + block.x - 1) / block.x, (M + block.y - 1) / block.y); // M
    kernel_output<<<grid, block>>>(Qp, kv, denom, output, M, d);
}
```

- **计算内容：** 将所有中间结果组合起来，计算最终的线性注意力输出。
- **并行策略：** 使用 2D Grid，确保每个线程处理最终 $M \times d$ 输出矩阵 `output` 中的一个元素。

C. 清理 (Cleanup)

C. 清理 (Cleanup)

C++

```
    cudaDeviceSynchronize(); // 关键：等待所有 Kernel 完成
    cudaFree(Kp);
    // ... cudaFree 释放所有中间指针
```

- `cudaDeviceSynchronize()`：这是必要的同步点。它确保主机程序在释放 GPU 内存或继续执行 CPU 任务之前，所有的 Kernel 运算都已在 GPU 上彻底完成。
- `cudaFree(...)`：释放所有在 GPU 上分配的临时内存，防止内存泄漏。

---

3. 示例追踪 (假设 M=2, d=3)

假设序列长度 $M = 2$，特征维度 $d = 3$。输入 Q, K, V 都是 2×3 矩阵。

步骤	计算目标	尺寸	Kernel 启动 Grid (块数)	线程任务
内存	`Md` (Kp, Qp)	$2 \times 3 = 6$		
	`dd` (kv)	$3 \times 3 = 9$		
1. $\phi(K)$	K_p	2×3	`grid` ≈ 1	6 个线程，每个线程处理一个 K 元素。
2. K_{sum}	k_{sum}	3×1	`grid` ≈ 1	3 个线程，每个线程计算 K_p 的一列总和。
3. $\mathbf{KV}$	kv	3×3	`grid` $\approx (1 \times 1)$	9 个线程，每个线程计算 kv 的一个元素。
4. $\mathbf{Denom}$	Q_p, `denom`	$2 \times 3, 2 \times 1$	`grid` ≈ 1	2 个线程，每个线程计算 Q_p 的一行，并计算 $\mathbf{Denom}$ 的一个元素。
5. Output	`output`	2×3	`grid` $\approx (1 \times 1)$	6 个线程，每个线程计算 Output 的一个元素。

3. 示例追踪 (假设 M=2, d=3)

假设序列长度 $M = 2$，特征维度 $d = 3$。输入 Q, K, V 都是 2×3 矩阵。

步骤	计算目标	尺寸	Kernel 启动 Grid (块数)	线程任务
内存	`Md` (Kp, Qp)	$2 \times 3 = 6$		
	`dd` (kv)	$3 \times 3 = 9$		
1. $\phi(K)$	K_p	2×3	`grid` ≈ 1	6 个线程，每个线程处理一个 K 元素。
2. K_{sum}	k_{sum}	3×1	`grid` ≈ 1	3 个线程，每个线程计算 K_p 的一列总和。
3. $\mathbf{KV}$	kv	3×3	`grid` $\approx (1 \times 1)$	9 个线程，每个线程计算 kv 的一个元素。
4. $\mathbf{Denom}$	Q_p, `denom`	$2 \times 3, 2 \times 1$	`grid` ≈ 1	2 个线程，每个线程计算 Q_p 的一行，并计算 $\mathbf{Denom}$ 的一个元素。
5. Output	`output`	2×3	`grid` $\approx (1 \times 1)$	6 个线程，每个线程计算 Output 的一个元素。

匯出到試算表

结论: `solve` 函数展示了如何将一个复杂的数学公式（线性注意力）高效地分解为一系列并行、相互依赖的 CUDA Kernel，并在主机端精确地管理它们的执行顺序、内存和线程配置。

Question 2:

不理解，请解释，用例子

不理解，为啥这里

```
    int i = blockIdx.x * blockDim.x + threadIdx.x; // row index
```

blockIdx.x * blockDim.x + threadIdx.x是row index

```
// Compute Qp = phi(Q) and denom[i] = Qp[i,:] · k_sum
__global__ void kernel_Qp_and_denom(const float* __restrict__ Q,
                                    const float* __restrict__ k_sum,
                                    float* __restrict__ Qp,
                                    float* __restrict__ denom,
                                    int M, int d) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // row index
    if (i >= M) return;

}
```

这里，blockIdx.x * blockDim.x + threadIdx.x是column index

```
// output[i, c] = (Qp[i,:] @ kv[:,c]) / denom[i]
__global__ void kernel_output(const float* __restrict__ Qp,
                              const float* __restrict__ kv,
                              const float* __restrict__ denom,
                              float* __restrict__ output,
                              int M, int d) {
    int c = blockIdx.x * blockDim.x + threadIdx.x; // column
    int i = blockIdx.y * blockDim.y + threadIdx.y; // row
    if (c >= d || i >= M) return;

}
```

不理解，请解释，用例子

不理解，为啥这里
int i = blockIdx.x * blockDim.x + threadIdx.x; // row index
blockIdx.x * blockDim.x + threadIdx.x是row index...

这是一个关于 CUDA 编程中 **并行策略（Parallel Strategy）** 和 **线程索引映射** 的核心问题。您观察到的现象是完全正确的，也是 CUDA 编程中设计高效 Kernel 的关键所在。

简而言之：CUDA 线程的 ID (`threadIdx.x`, `blockIdx.x` 等) 本身没有固定的含义（行或列），它的含义完全取决于您在主机端启动 Kernel 时如何配置 Grid/Block 尺寸，以及在 Kernel 内部如何解释这个 ID。

在您提供的两个 Kernel 中，这种映射方式是不同的，因为它们要解决的计算任务性质不同。

---

1. 原理：线程 ID 的含义由 Kernel 决定

在 CUDA 中：

- **全局线程 ID:** `threadIdx.x + blockIdx.x * blockDim.x` 只是一个从 0 开始的、一维的、唯一的数字。
- **映射:** 我们必须在 Kernel 内部将这个数字映射到我们想要处理的 $M \times N$ 矩阵的 **(行, 列)** 坐标上。

映射公式（以 X 轴为例）

场景	Grid 尺寸 (Host)	Kernel 内部映射	为什么？
一维任务	Grid 只有 X 维度	ID → 数组索引	简单数组操作

映射公式（以 X 轴为例）

场景	Grid 尺寸 (Host)	Kernel 内部映射	为什么？
一维任务	Grid 只有 X 维度	ID → 数组索引	简单数组操作
二维任务 1	Grid 是 $X \times 1$	ID → **行**索引	矩阵的行数 M 远小于列数 N 或 M 适合一维处理。
二维任务 2	Grid 是 $X \times Y$	ID.x → **列**索引，ID.y → **行**索引	矩阵尺寸平衡，使用 2D 并行提高效率。

匯出到試算表

---

2. Kernel 1: `kernel_Qp_and_denom` (一维映射到 Row Index)

这个 Kernel 的任务是**逐行**计算 Q_p 和 denom。这意味着每个线程需要负责处理 Q 矩阵中的一整行。

A. 主机端启动配置（Host Launch Configuration）

在主机函数 `solve` 中，该 Kernel 是这样启动的：

C++

```
// Host side launch
// M 是行数
int grid  = (M + block - 1) / block;
kernel_Qp_and_denom<<<grid, block>>>(..., M, d);
```

- **Grid 维度：** 只有 X 维度（`grid` > 1），而 Y 维度是 1。
- **Grid 大小：** Grid 覆盖了矩阵 Q 的**行数** M。

- **Grid 大小：** Grid 覆盖了矩阵 Q 的**行数** M。

B. 设备端 Kernel 映射

C++

```
__global__ void kernel_Qp_and_denom(...) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // row index
    if (i >= M) return;
    // ... 线程 i 串行循环 d 次，计算 Q 矩阵的第 i 行 ...
}
```

- **解释:** 由于 Grid 的大小是根据**行数** M 计算的，所以将线程的全局 X 坐标（`i`）直接解释为**行索引**（`row index`）是最自然、最简单的方式。
- **效率考量:** 在 Kernel 内部，线程 i 随后**串行**循环 d 次（d 是列数），完成了该行的所有计算。这种策略被称为 **Map-to-Row**。

示例追踪 (M=10, 追踪 Row Index)

假设 $M = 10$ 行，`blockDim.x`=5，`blockIdx.x`=0。

线程 ID（`threadIdx.x`）	全局 ID i	映射含义	线程任务
0	$0 \times 5 + 0 = \mathbf{0}$	**Row 0**	线程 T_0 处理 Q 的第 0 行
1	$0 \times 5 + 1 = \mathbf{1}$	**Row 1**	线程 T_1 处理 Q 的第 1 行
...	...	...	...
5	$1 \times 5 + 0 = \mathbf{5}$	**Row 5**	线程 T_5 处理 Q 的第 5 行

3. Kernel 2: `kernel_output` (二维映射到 Row & Column Index)

这个 Kernel 的任务是计算最终输出矩阵 output 的每一个元素。

A. 主机端启动配置（Host Launch Configuration）

在主机函数 `solve` 中，该 Kernel 是这样启动的：

C++

```
// Host side launch
// M 是行数, d 是列数
dim3 grid((d + block.x - 1) / block.x,  // Grid.x 覆盖 d 列
          (M + block.y - 1) / block.y); // Grid.y 覆盖 M 行
kernel_output<<<grid, block>>>(..., M, d);
```

- **Grid 维度：** Grid 是 $X \times Y$ 的二维结构。
- **Grid 大小：** Grid.x 覆盖**列数** d；Grid.y 覆盖**行数** M。

B. 设备端 Kernel 映射

C++

```
__global__ void kernel_output(...) {
    int c = blockIdx.x * blockDim.x + threadIdx.x; // column
    int i = blockIdx.y * blockDim.y + threadIdx.y; // row
    if (c >= d || i >= M) return;
    // ... 线程 (i, c) 串行循环 d 次，计算 output[i, c] ...
}
```

- **解释:** 由于 Grid 的尺寸与输出矩阵 output 的 $M \times d$ 尺寸完全匹配，所以：
  - X **轴 (col):** 线程的全局 X 坐标 (`c`) 被映射为**列索引** (`column index`)。
  - Y **轴 (row):** 线程的全局 Y 坐标 (`i`) 被映射为**行索引** (`row index`)。

- **解释:** 由于 Grid 的尺寸与输出矩阵 output 的 $M \times d$ 尺寸完全匹配，所以：
  - X **轴 (col):** 线程的全局 X 坐标 (`c`) 被映射为**列索引 (** `column index` **)**。
  - Y **轴 (row):** 线程的全局 Y 坐标 (`i`) 被映射为**行索引 (** `row index` **)**。
- **效率考量:** 这种 **Map-to-Element** 策略是矩阵操作中最常见的，因为它让每个线程负责一个输出元素，最大限度地实现了并行化。

示例追踪 (M=4, d=4, Block=2x2)

假设 $M = 4, d = 4$。我们追踪 Block (0, 1) 中的线程 $T_{(1,0)}$。

坐标	线程 ID	Block ID	Block Dim	映射含义
X (列 c)	`threadIdx.x` = 1	`blockIdx.x` = 0	`blockDim.x` = 2	$c = 0 \times 2 + 1 = \mathbf{1}$
Y (行 i)	`threadIdx.y` = 0	`blockIdx.y` = 1	`blockDim.y` = 2	$i = 1 \times 2 + 0 = \mathbf{2}$

匯出到試算表

结论: 线程 $T_{(1,0)}$ 负责计算 output 矩阵中 $(\mathbf{2}, \mathbf{1})$ 处的元素。

总结：核心区别

Kernel	映射方式	目的
`kernel_Qp_and_denom`	**1D (Map-to-Row)**	线程 T_i 处理第 i 行的所有 d 个元素。
`kernel_output`	**2D (Map-to-Element)**	线程 $T_{(i,c)}$ 只处理第 i 行第 c 列的一个元素。

匯出到試算表