# 体系结构Lab6实验报告

何跃强 PB22111649

2025-6-10

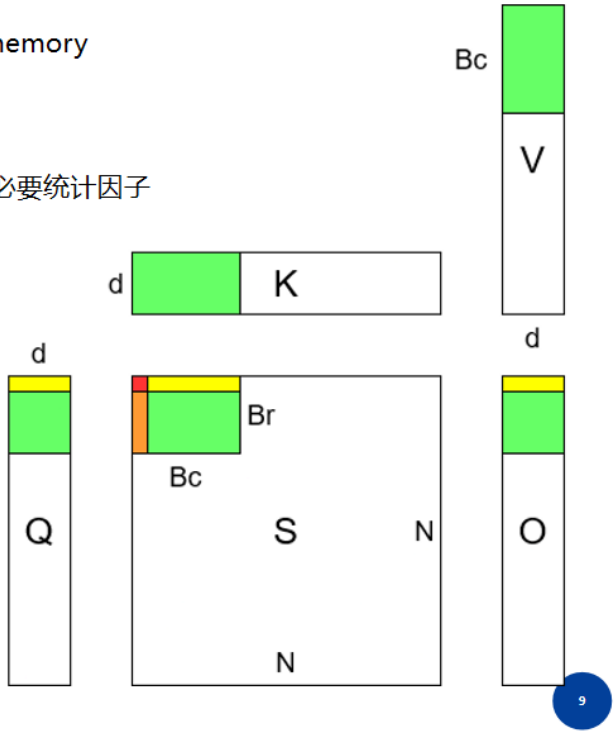
## 一、实验要求

改进Attention矩阵计算，即Flash attention算法和Single head Flash attention算法。

Flash Attention：

**Flash attention算法**

1. 如图所示，将所有有颜色部分放入shared memory
2. 计算方法：
    1. 先计算Q小块乘K小块得到S小块
    2. S小块做safe softmax，同时记录更新必要统计因子
    3. S小块乘V小块得到O小块
3. 内层对Q矩阵循环，外层对K、V循环。即算完O小块后K、V小块不动，搬入新的Q小块继续计算直到搬到最后一块Q小块。此时下一步才搬入新的K、V小块，Q小块则从头搬入
4. 算法对每个头启动Br个线程，这些线程如上所述完成计算



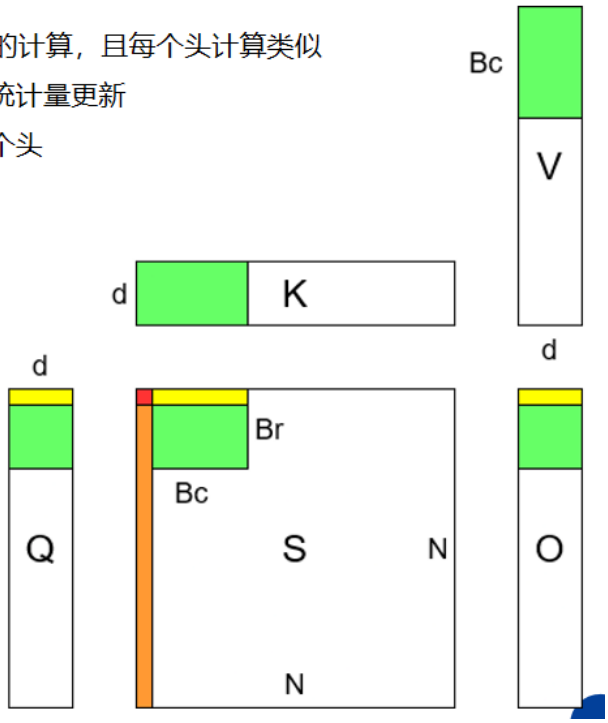

single head flash attention：

**Single head Flash attention**

1. 原始flash attention算法涉及多个注意力头的计算，且每个头计算类似
2. 原始算法涉及较为复杂的安全softmax及其统计量更新
3. 为此，使用不安全的softmax，且仅考虑一个头
4. 为加速计算，启动N个而非Br个线程
5. 分块仍保持原始flash attention算法的方案
6. 设定Bc=Br

提示：

1. 先确保S矩阵计算正确，再确保O矩阵计算正确
2. 使用常规softmax函数，即不需要考虑-a操作而仅需取exp
3. 注意需要求softmax对应的归一化矩阵Z



## 二、代码解释

```
#include <cuda_profiler_api.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define smalloc(type, ptr, num)                                          \
  if (!(ptr = (type *)malloc(sizeof(type) * (num))))                     \
  exit(1)
#define Verifylen (1024)
#define Seqlen (262144)
#define AD (32)
#define AT (64)
#define BC (AT) // Bc = Br = AT

__global__ void Single_head_flash_atten(float *Q, float *K, float *V, float *O,
                                        unsigned l,
                                        float softmax_scale) { // unsigned d,
  // 线程索引
  const unsigned row = blockIdx.x * blockDim.x + threadIdx.x;
  if (row >= l)
    return;

  float sumexp = 0.0f;     // 当前行指数和
  float O_temp[AD] = {0}; // 中间结果，需要最后归一化

  __shared__ float Qs[BC][AD];
#pragma unroll
  for (int j = 0; j < AD; j++) {
    Qs[threadIdx.x][j] = Q[row * AD + j];
  }

  // 外层循环：遍历K/V的块
  for (unsigned j_block = 0; j_block < l; j_block += BC) {

    __shared__ float Ks[BC][AD];
    __shared__ float Vs[BC][AD];
    // __shared__ float Qs[BC][AD];

    // 协作加载K和V的分块到共享内存
    if (threadIdx.x < BC) {
      unsigned load_idx = j_block + threadIdx.x;
#pragma unroll
      for (int j = 0; j < AD; j++) {
        Ks[threadIdx.x][j] = K[load_idx * AD + j];
        Vs[threadIdx.x][j] = V[load_idx * AD + j];
        // Qs[threadIdx.x][j] = Q[row * AD + j];
      }
    }
    __syncthreads();

    // 计算当前块的S值
    float S_values[BC] = {0};
```

```
#pragma unroll
    for (int j = 0; j < BC; j++) {
#pragma unroll
      for (int k = 0; k < AD; k++) {
        S_values[j] += Qs[threadIdx.x][k] * Ks[j][k]; // Q * K^T
      }
      S_values[j] = expf(S_values[j] * softmax_scale);
      sumexp += S_values[j]; // 累加指数和
    }

// 计算O
#pragma unroll
    for (int j = 0; j < BC; j++) {
#pragma unroll
      for (int k = 0; k < AD; k++) {
        O_temp[k] += S_values[j] * Vs[j][k];
      }
    }

    __syncthreads();
  }

#pragma unroll
  for (int j = 0; j < AD; j++) {
    O[row * AD + j] = O_temp[j] / sumexp;
  }

  return;
}

__host__ void single_head_atten_base(float *Q, float *K, float *V, float *O,
                                     unsigned l, float softmax_scale) {
  unsigned i, j, k;
  float *S, *Ssum;
  smalloc(float, S, l *l);
  smalloc(float, Ssum, l);
  for (i = 0; i < l; i++) {
    Ssum[i] = 0;
    for (j = 0; j < l; j++) {
      S[i * l + j] = 0;
      for (k = 0; k < AD; k++) {
        S[i * l + j] += Q[i * AD + k] * K[k + j * AD]; // Q* KT
      }
      S[i * l + j] = exp(S[i * l + j] * softmax_scale); //
      Ssum[i] += S[i * l + j];
    }
  }

  for (i = 0; i < l; i++) {
    for (j = 0; j < AD; j++) {
      O[i * AD + j] = 0;
      for (k = 0; k < l; k++) {
        O[i * AD + j] += S[i * l + k] * V[k * AD + j] / Ssum[i];
      }
    }
  }
```

```
  free(S);
  free(Ssum);
}

__host__ void gen_QKV(float **phQ, float **phK, float **phV, unsigned l,
                      unsigned d) {
  float *hQ, *hK, *hV;
  smalloc(float, hQ, l *d);
  smalloc(float, hK, l *d);
  smalloc(float, hV, l *d);
  unsigned i;
  for (i = 0; i < l * d; i++) {
    hQ[i] = 1.0 * rand() / RAND_MAX;
    hK[i] = 1.0 * rand() / RAND_MAX;
    hV[i] = 1.0 * rand() / RAND_MAX;
  }
  *phQ = hQ;
  *phK = hK;
  *phV = hV;
}
__host__ unsigned compare(float *pred_, float *true_, unsigned n) {
  unsigned i;
  float relative_error;
  for (i = 0; i < n; i++) {
    relative_error = fabs((pred_[i] - true_[i]) / true_[i]);
    if (relative_error >= 1e-5) {
      printf("not equal! relative error: %12.9lf pred: %12.9f true: %12.9f\n",
             relative_error, pred_[i], true_[i]);
      return 1;
    } else {
      // printf("equal! relative error: %12.9lf pred: %12.9f true: %12.9f\n",
      //        relative_error, pred_[i], true_[i]);
    }
  }
  printf("equal!\n");
  return 0;
}

int prinMat(float *A, int m, int n, FILE *fp) {
  int i, j;
  for (i = 0; i < m; i++) {
    fprintf(fp, "%4d:", i);
    for (j = 0; j < n; j++) {
      fprintf(fp, "%12.9f ", A[i * n + j]);
    }
    fprintf(fp, "\n");
  }
  return 0;
}

int main(void) {
  float *dQ, *dK, *dV, *dO, *hQ, *hK, *hV, *hO, *Obase;
  const unsigned Vl = Verifylen, Pl = Seqlen;
  const float softmax_scale = 1 / sqrt(AD);
  unsigned i;
  gen_QKV(&hQ, &hK, &hV, Vl, AD);
```

```
  smalloc(float, hO, Vl *AD);
  smalloc(float, Obase, Vl *AD);
  cudaMalloc(&dQ, sizeof(float) * (Vl * AD));
  cudaMalloc(&dK, sizeof(float) * (Vl * AD));
  cudaMalloc(&dV, sizeof(float) * (Vl * AD));
  cudaMalloc(&dO, sizeof(float) * (Vl * AD));
  cudaMemcpy(dQ, hQ, sizeof(float) * (Vl * AD), cudaMemcpyHostToDevice);
  cudaMemcpy(dK, hK, sizeof(float) * (Vl * AD), cudaMemcpyHostToDevice);
  cudaMemcpy(dV, hV, sizeof(float) * (Vl * AD), cudaMemcpyHostToDevice);
  dim3 gridsize(Vl / AT), blocksize(AT);
  Single_head_flash_atten<<<gridsize, blocksize>>>(dQ, dK, dV, dO, Vl,
                                                   softmax_scale);
  cudaMemcpy(hO, dO, sizeof(float) * (Vl * AD), cudaMemcpyDeviceToHost);
  single_head_atten_base(hQ, hK, hV, Obase, Vl, softmax_scale);
  cudaDeviceSynchronize();
  unsigned flag = 0;
  flag |= compare(hO, Obase, Vl * AD);
  if (flag) {
    printf("test fail!\n");
    exit(0);
  }
  printf("test pass!\n");
  free(hQ);
  free(hK);
  free(hV);
  free(hO);
  free(Obase);
  cudaFree(dQ);
  cudaFree(dK);
  cudaFree(dV);
  cudaFree(dO);

  gen_QKV(&hQ, &hK, &hV, Pl, AD);
  cudaMalloc(&dQ, sizeof(float) * (Pl * AD));
  cudaMalloc(&dK, sizeof(float) * (Pl * AD));
  cudaMalloc(&dV, sizeof(float) * (Pl * AD));
  cudaMalloc(&dO, sizeof(float) * (Pl * AD));
  cudaMemcpy(dQ, hQ, sizeof(float) * (Pl * AD), cudaMemcpyHostToDevice);
  cudaMemcpy(dK, hK, sizeof(float) * (Pl * AD), cudaMemcpyHostToDevice);
  cudaMemcpy(dV, hV, sizeof(float) * (Pl * AD), cudaMemcpyHostToDevice);
  gridsize = {Pl / AT};
  blocksize = {AT};

  cudaEvent_t start, stop;
  float Time1 = 0.0, temp = 0;
  const unsigned loopnum = 10;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  for (i = 0; i < loopnum; i++) {
    cudaEventRecord(start, 0);
    Single_head_flash_atten<<<gridsize, blocksize>>>(dQ, dK, dV, dO, Pl,
                                                     softmax_scale);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&temp, start, stop);
    Time1 += temp;
```

```
    temp = 0;
    cudaDeviceSynchronize();
  }

  printf("l: %5.d   time: %12.9f\n", Pl, Time1 / loopnum);
  free(hQ);
  free(hK);
  free(hV);
  cudaFree(dQ);
  cudaFree(dK);
  cudaFree(dV);
  cudaFree(dO);
}
```

# 三、运行结果


可见测试通过

# 四、问题回答

## 4.1 原始FlashAttention 算法中能否将内层循环改为对K矩阵操作?

将原始Flash Attention 算法的内层循环改为对 K 矩阵操作需要调整计算顺序。 在修改后的方法中，Q和V矩阵被加载到共享内存中，而K矩阵按块处理。

过程：

Q和V矩阵最初加载到每个线程块的共享内存中。

K矩阵按块 （大小$Bc * d$）加载到共享内存中，每个迭代一次。

计算中间注意力分数（S）并累积。

O 矩阵通过将S与V相乘并存储结果进行更新。

## 4.2 Q、K、V、O小块大小能否为(Br,d/2)?

使用(Br, d/2) 作为 Q、K、V 和O矩阵的小块大小在理论上是可行的，但存在问题。

可能遇到的问题：

与硬件warp大小（例如NVIDIAGPU 上的32）不对齐可 能导致线程利用率不足。

每个线程处理特征维度的一半，可能增加延迟。

在较小的 d/2 块上进行 Softmax 归一化可能因平均不足而引入精度误差。

内存合并可能受干扰，增加全局内存访问时间。

### 4.3 Single Head Flash Attention启动N个线程相比原始Flash Attention 算法对矩阵分块复用的影响

相比原始算法基于块的分块复用，Single Head Flash Attention 启动 N 个线程更能影响矩阵复用。原始方法在块内复用Q、K和V块，减少全局内存获取。

分析：

N个线程增加并行性，但如果每个线程访问不同的全局内存位置，复用减少。

每个块的共享内存使用减少，可能导致更频繁的全局内存加载。

线程可能不共享相同的Q,K,V子集，增加内存带宽需求。

N增加会导致同步开销上升，若未优化，会影响性能。

### 4.4 实现的Single Head Flash Attention 运行速度较慢的分析

原因如下：

过度使用共享内存（Qs、Ks、Vs）且块大小未优化，导致冲突。

针对较大的Seqlen（例如测试的n:262144）缺乏平铺优化，导致内存访问模式效率低下。

每次块加载后的syncthreads()会影响性能。

改进方案：

优化块大小（BC）以匹配GPU warp大小（例如32），减少冲突。

实施平铺处理Q、K、V，按较小块处理，提高缓存命中率。

尽可能并行处理K 和V，减少syncthreads()带来的时间影响。

困难：

调优块大小需要大量性能分析，且可能因GPU架构而异。

代码复杂性和调试工作量增加。

对于syncthreads()需要十分谨慎。