SGEMM Optimize practice

> 参考blog: How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog
> SGEMM performs `C=αAB+βC` at single (=32b) precision.

How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog

该 blog 对应的代码在博主的 github 仓库：

SGEMM_CUDA

NVIDIA_SGEMM_PRACTICE

https://zhuanlan.zhihu.com/p/441146275

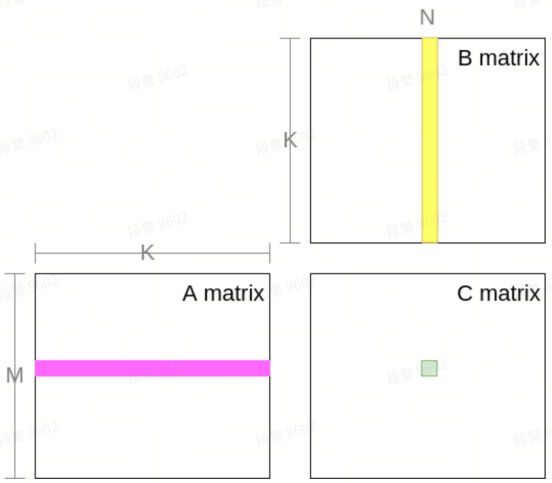
https://zhuanlan.zhihu.com/p/461060382

其他参考资料：

安培架构白皮书

> 下面内容相当于该blog的部分翻译和阅读笔记。

Kernel 1: Naive Implementation

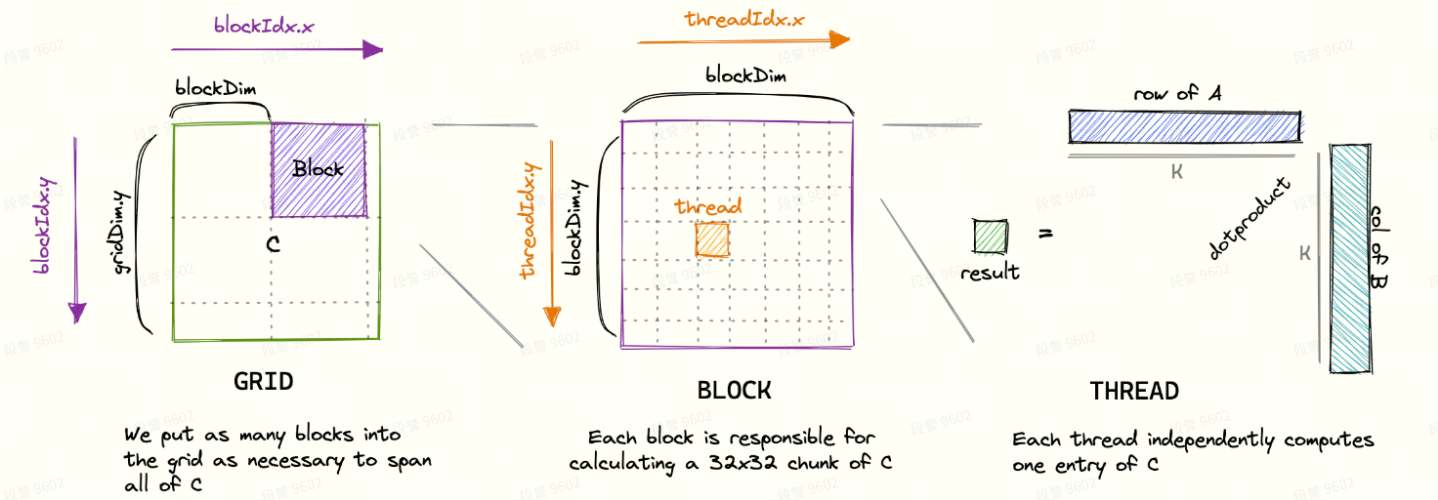


算法逻辑：

> 将每个 cuda 线程与矩阵C的每一个元素相对应，每个线程负责 C 中一个元素的计算；
>
> 所以 grid size 与结果矩阵 C 的 size 一致， block的最大线程数不能超过 1024，故 block size 和 gird size 设置如下：

> dim3 blockDim(32, 32)
>
> dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32))

To visualize this simple kernel:[10]




```
// A(M, K) * B(K, N) = C(M, N)
// Naive Implementation
__global__ void mysgemm_v1(int M, int N, int K, float alpha, const float *A,
                           const float *B, float beta, float *C) {
//   const uint x = blockIdx.x * blockDim.x + threadIdx.x; // 网格中每个线程，都有单独的坐标 全局x
//   const uint y = blockIdx.y * blockDim.y + threadIdx.y; // 全局y

  int x = blockIdx.x * blockDim.x + threadIdx.x; // 全局x
  int y = blockIdx.y * blockDim.y + threadIdx.y; // 全局y
  // if statement is necessary to make things work under tile quantization
  if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
      tmp += A[x * K + i] * B[i * N + y]; // 两次全局内存访问和一次FMA（累加乘）
    }
    // C = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

接下来分析一下这个矩阵乘法计算的一些数值。

两个矩阵相乘，运算量为 `2*m*k*n`

所以：

> Total FLOPs: 2*4096^3 = 136 GFLOPs

最小需要读写的内存：`3 * 4096² * 4B + 4096² * 4B = 268 MB`

A100 的性能：

NVIDIA A100 TENSOR CORE GPU 规格
（SXM4 和 PCIE 外形规格）

	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS| 1248 TOPS*	
GPU 显存	80GB HBM2e	80GB HBM2e
GPU 显存带宽	1935GB/s	2039GB/s

fp32算力：19.5 TFLOPS

峰值显存带宽：1935 GB/s

因此，理论最小计算时间：`136 / 1000 / 19.5 = 7 ms`

理论最小内存读写时间：`268 / 1000 / 1935 = 0.14 ms`

计算时间超过内存读取时间的十倍，往往可以被认为是 **compute-bound**

我在 A100 上测试，实际数值为：

```
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.471403) s, performance: (  291.6) GFLOPS. size: (4096).
```

分析下这个 naive kernel 为啥如此的慢：

显然，这个kernel 实际计算过程中，并不仅仅只有 268 MB 这么多内存读写。同一行的 thread 会重复读取 A 中的每一行，同一列的 thread 会重复读取 B 中的每一列。所以实际内存的读写大小为：

`2 * 4092² * 4096 * 4B + 4092² * 2 * 4B = 549890 MB`

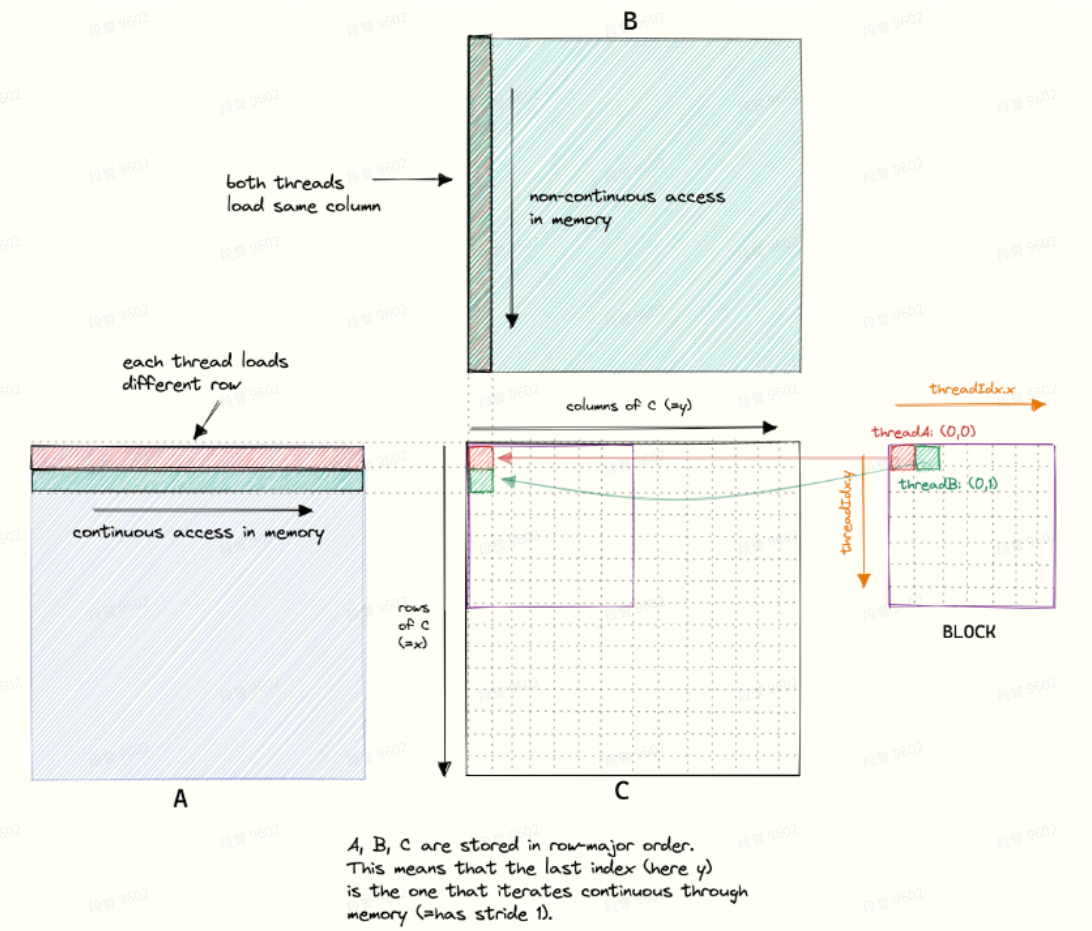
需要的时间为：`549890/ 1000 / 1935 = 0.28s`，实际内存加载时，内部会有一些优化，以及多个线程load 同一行或列时，存在广播机制，所以实际内存读取量没有这么大。

- 计算访存比：每次迭代需要进行一次FMA（乘累加）和两次全局内存读取，计算访存比1/2；
- 访存量：访问全局内存，C矩阵每个元素计算需要访问 `2K` 个单精度浮点数，完成全部计算需要 `2*K*M*N`；

全局内存访问延迟高（几百cycle），**同时相同位置元素被重复读取**（C中同一行元素计算共享A中同一行元素，C中同一列元素计算共享B中同一列元素），另一方面，较低的计算访存比无法有效隐藏访存延迟，因此，访存延迟和计算访存比是导致kernel 1效率低下的原因。

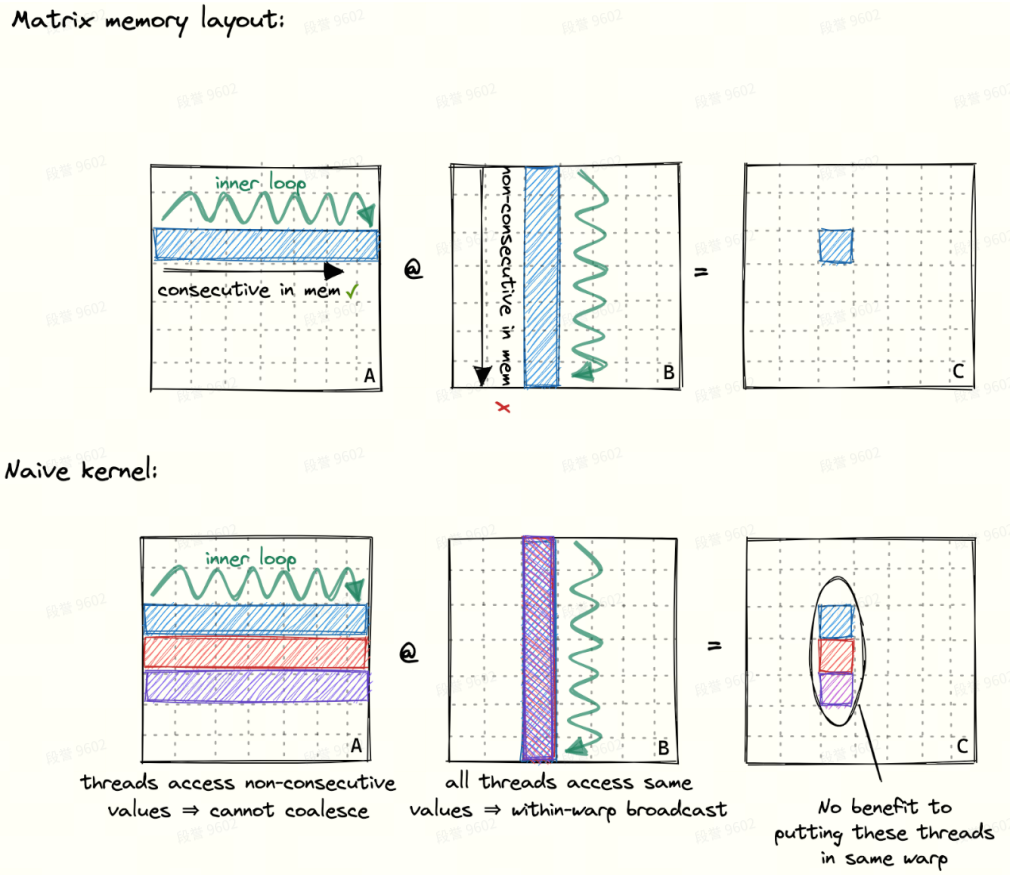
Memory Access Pattern of the Naive Kernel

Kernel 1 中，每一个 thread 都需要 load A 中的一行和 B 中的一列以及 C 中的一个元素，If we assume the worst case of zero caching, then each thread has to load `2*4092+1` floats from global memory. As we have 4092^2 threads total, this would result in 548GB of memory traffic.



从这幅图中，也可以直观的看出，该 kernel 有太多的重复内存读取。

Kernel 2: Global Memory Coalescing



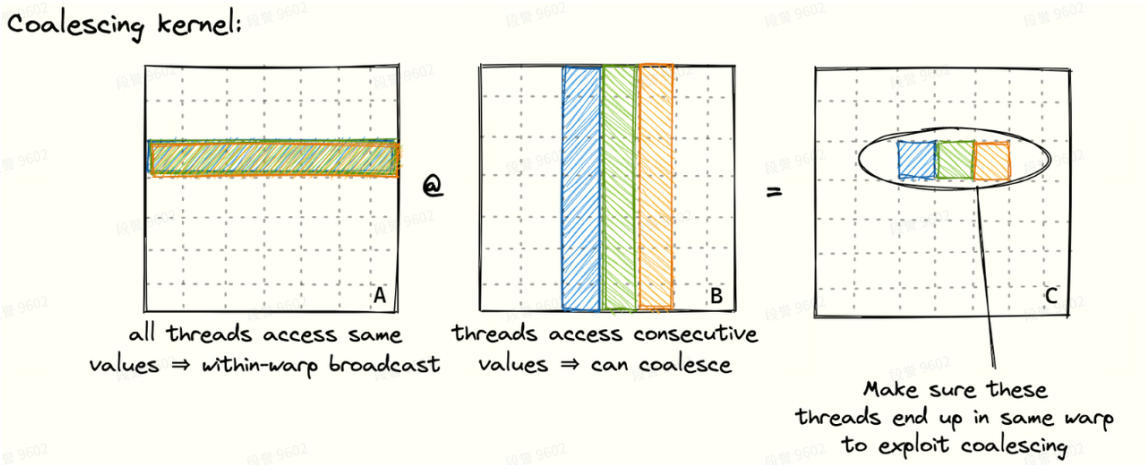
这里解释一下，为什么 Naive Kenel 是不能够合并内存访问的。

从图中可以看出，连续的三个线程，每次访问A时，都需要从A中的三个不同的行中访问一个元素，需要从B的同一列中访问同一个元素，故对A的访问是无法合并的。

是否是合并内存访问，需要站在一个warp的视角（32个线程），同时进行一次访问内存时（内层for循环的一次读取操作），是否可以合并为一次。

而不是一个线程，不同时间，多次访问的内存是否是连续内存。合并内存访问与被访问的内存是否连续有很大关系，但是要注意上述的理解点。

合并内存访问



与前面相反，图中3个线程，同时访问B中的不同列的某个元素，刚这三个元素在同一行中，处于连续内存中，故可以合并访问，对C中的访问也是类似的。代码如下：

```
const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);

if (x < M && y < N) {
    float tmp = 0.0;
    for (int i = 0; i < K; ++i) {
        tmp += A[x * K + i] * B[i * N + y];
    }
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
}
```

And we call it like so.

```
// gridDim stays the same
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
// make blockDim 1-dimensional, but don't change number of threads
dim3 blockDim(32 * 32);
sgemm_coalescing<<<gridDim, blockDim(M, N, K, alpha, A, B, beta, C);
```

实际效果如下：

```
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.044722) s, performance: ( 3073.2) GFLOPS. size: (4096).
```

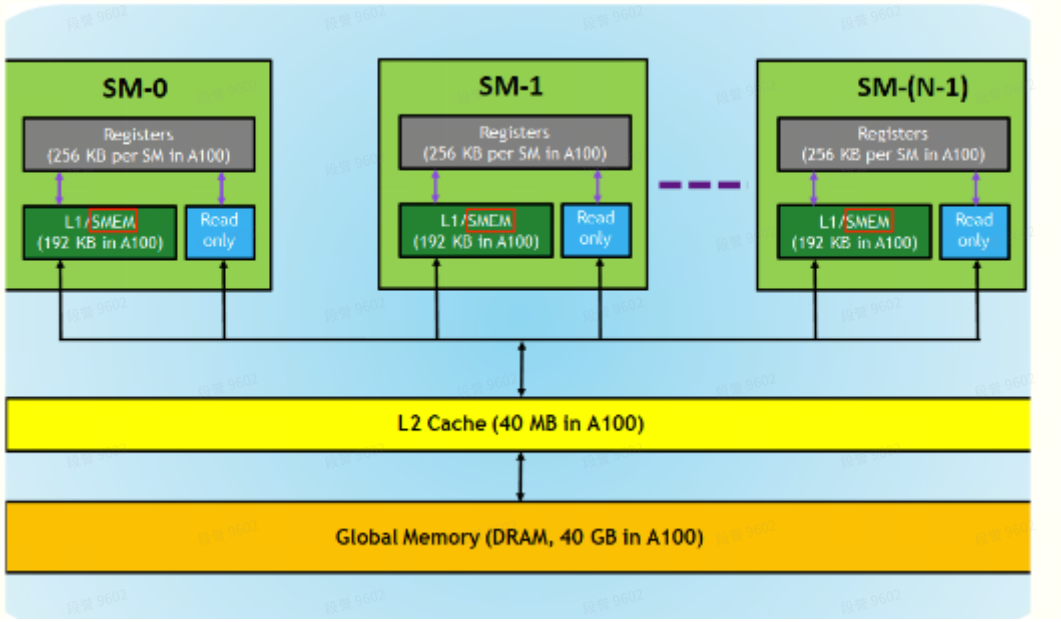
与 kernel1 相比，合并内存访问能大幅提高内存加载效率，减少内存加载次数，因此性能上获得了巨大提升。

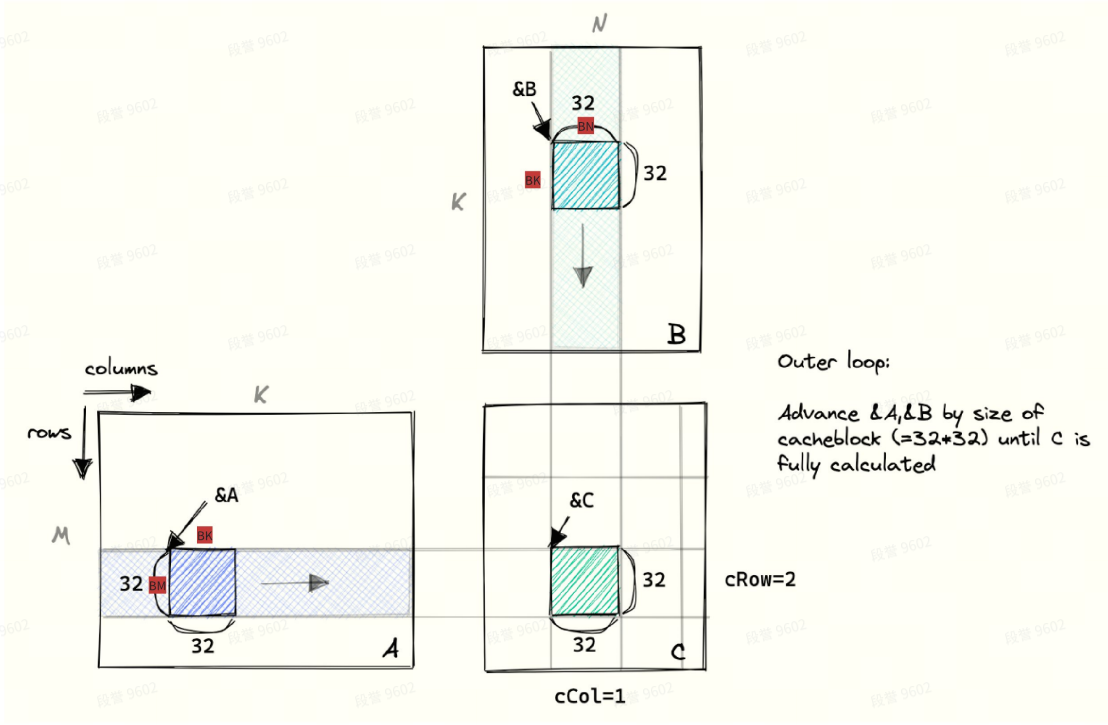
Kernel 3: Shared Memory Cache-Blocking

上面两个 kernel 中，依然存在大量的 **全局内存的重复访问**。

一个优化思路自然是，把需要重复访问的数据放到 **shared memory**， 提高读取的速度。

共享内存是片上内存，具有较低的访存延迟（几十cycle）,使用共享内存进行缓存可降低访存延迟；





该kernel的核心思想是， 每一个 block，只负责计算result 矩阵 C 中的一部分。

不断滑动BM*BK，BK*BN 两个区域，将该区域里面的数据加载到共享内存中，然后做 FMA。

该 kernel 的代码如下：

```
template <const int BLOCKSIZE>
__global__ void sgemm_shared_mem_block(int M, int N, int K, float alpha,
                                       const float *A, const float *B,
                                       float beta, float *C) {
  // the output block that we want to compute in this threadblock
```

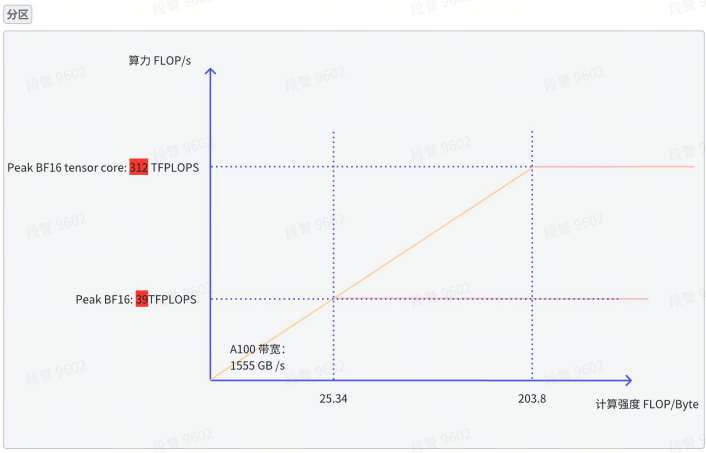
```
    const uint cRow = blockIdx.x;
    const uint cCol = blockIdx.y;

    // allocate buffer for current block in fast shared mem
    // shared mem is shared between all threads in a block
    __shared__ float As[BLOCKSIZE * BLOCKSIZE];
    __shared__ float Bs[BLOCKSIZE * BLOCKSIZE];

    // the inner row & col that we're accessing in this thread
    const uint threadCol = threadIdx.x % BLOCKSIZE;
    const uint threadRow = threadIdx.x / BLOCKSIZE;

    // advance pointers to the starting positions
    A += cRow * BLOCKSIZE * K;                    // row=cRow, col=0
    B += cCol * BLOCKSIZE;                        // row=0, col=cCol
    C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol

    float tmp = 0.0;
    for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
      // Have each thread load one of the elements in A & B
      // Make the threadCol (=threadIdx.x) the consecutive index
      // to allow global memory access coalescing
      As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
      Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];

      // block threads in this block until cache is fully populated
      __syncthreads();
      A += BLOCKSIZE;
      B += BLOCKSIZE * N;

      // execute the dotproduct on the currently cached block
      for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
        tmp += As[threadRow * BLOCKSIZE + dotIdx] *
               Bs[dotIdx * BLOCKSIZE + threadCol];
      }
      // need to sync again at the end, to avoid faster threads
      // fetching the next block into the cache before slower threads are done
      __syncthreads();
    }
    C[threadRow * N + threadCol] =
        alpha * tmp + beta * C[threadRow * N + threadCol];
}
```

该算子实际性能如下：

```
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.025287) s, performance: ( 5435.2) GFLOPS. size: (4096).
```

前面 kernel2 的性能如下：

```
dimensions(m=n=k) 4096, alpha: 0.5, beta: 3
Average elapsed time: (0.044722) s, performance: ( 3073.2) GFLOPS. size: (4096).
```



附件

generate阶段，seq_len = 1

attention计算量：

两个乘法

`q =(batch, num_heads, 1,  head_size)`

`k,v= (batch, num_head, context_len, head_size)`

`attn_weight = q*K^T = (batch, num_heads, 1, seq_len)`

`attn_out = attn_weight*v = (batch, num_heads, seq_len, head_size)`

计算量 **4* batch*num_heads*context_len*head_size**

= **4* batch*context_len*hidden_size**

Softmax

对于一个维度为n的输入向量，计算Softmax函数需要进行n次**指数**运算和n次加法/乘法运算。

f(context_len)

mlp 计算量：

也是两个乘法：

(batch, 1, hidden_size) *(hidden_size, middle_size)

(batch, 1, middle_size)*(middle_size, hidden_size)

计算量：**4*batch*hidden_size*middle_size**

比值：

$$\frac{4*batch*hidden\_size*context\_len+f(context\_len)}{4*batch*hidden\_size*middle\_size}=\frac{context\_len}{middle\_size}+\frac{f(context\_len)}{4*batch*hidden\_size*middle\_size}$$

后一项忽略不计

Table 1. NVIDIA A100 Tensor Core GPU Performance Specs

Peak FP64[1]	9.7 TFLOPS
Peak FP64 Tensor Core[1]	19.5 TFLOPS
Peak FP32[1]	19.5 TFLOPS
Peak FP16[1]	78 TFLOPS
Peak BF16[1]	39 TFLOPS
Peak TF32 Tensor Core[1]	156 TFLOPS | 312 TFLOPS[2]
Peak FP16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak BF16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak INT8 Tensor Core[1]	624 TOPS | 1,248 TOPS[2]
Peak INT4 Tensor Core[1]	1,248 TOPS | 2,496 TOPS[2]