第18次课程-用Cuda实现LLama模型中多头注意力算子

自制大模型推理框架

带你从零写一个支持LLama2/3推理
支持Cuda加速和int8量化的大模型框架

课程亮点

一、项目整体架构和设计

学习架构思维，我们将在这里学习算子、张量等关键数据结构的设计。

二、支持LLama2模型结构

对LLama模型的权重读取和构建，用KV Cache等机制支持加速推理。

三、模型的量化

为了减少显存的占用，我们会一起开发int8模型量化模块，包括量化模型导出和量化算子支持。

四、Cuda基础和算子实现

带你学Cuda并实战大模型算子的实现，每个算子都将讲解原理，从零手写。

五、用推理框架做点有趣的事情

文本生成，生成故事，多轮对话。

现价优惠　扫码马上报名

视频

正常版本.mp4

在前面的几次课程中，我们分别用Cuda高效的实现了RMSNorm、GEMV、kv-cache

“Hello，my”

32表示hello

[512维度]向量

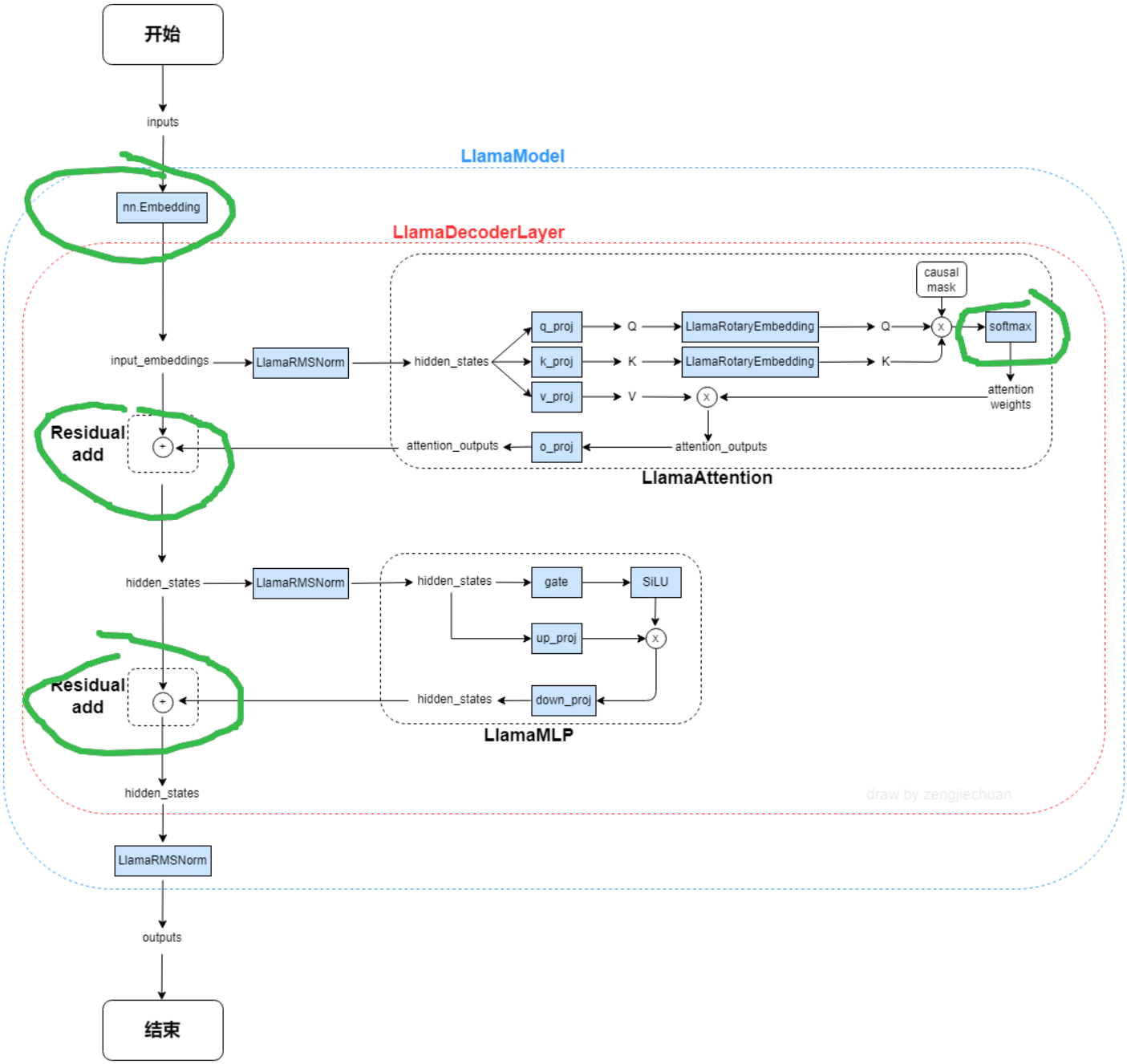
99表示，

[512维度] 向量

37表示my

Inputs = [32,99,37]

[3, 512]，维度为3×512输入向量



从以上LLama结构图中可以看出，除了这些算子之外，还要再实现下面的几个算子，我们将在这节课中当中进行逐一地分析和实现：

1. Softmax，在多头注意力机制的计算中使用，该算子的作用是将每个元素的值转换为介于0和1之间的概率值，并确保所有这些概率值加起来等于1；
2. Add算子，用于MLP块输入和MLP计算结果的求和，另外也用于将Attention块的计算结果和Attention块的输入进行求和；
3. Embedding算子，根据输入的id从词表中取出对应的映射，换句话说就是将每个单词或标记（token）映射到一个多维的向量空间中，这些向量是实数构成的，通常具有固定的长度。比如现在有一个大小为65200×512的词表，65200就表示对应输入的单词数量，512就表示映射后的向量维度。
4. 多头注意力算子

多头注意力算子的实现

Q = (3, dim)

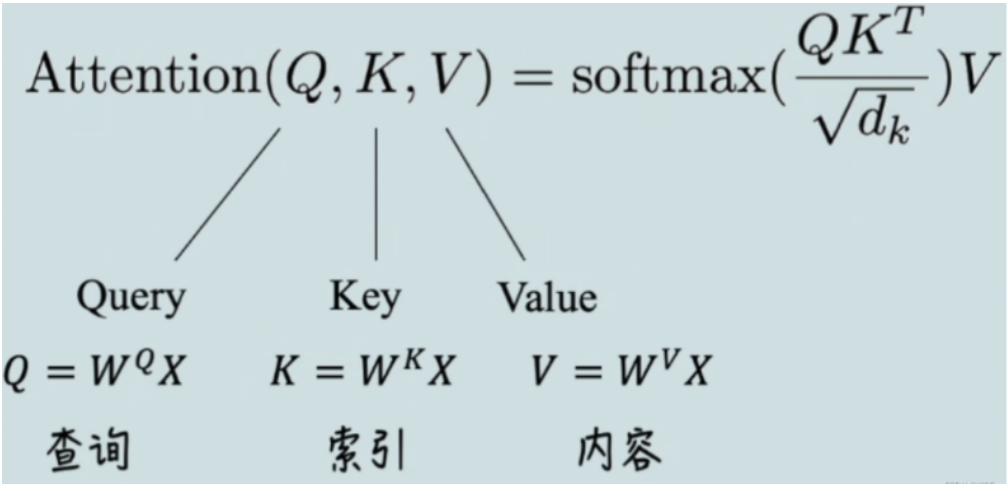
K = (3, dim)

V = (3,dim)

score = Q matmul K.T = **(3, 3)**

score/= sqrt(dim)

score @V = (3,dim)



在自注意力机制的实现中，有如下的Python实现则有以下的步骤：

多头将wq = nn.Linear(dim,dim) 变为wq = nn.Linear(dim, head_dim * num_head, bias=False)

```
wq = nn.Linear(dim, head_dim * num_head, bias=False) # dim = head_dim * num_head
wk = nn.Linear(dim, head_dim * num_head, bias=False)
wv = nn.Linear(dim, head_dim * num_head, bias=False)
wo = nn.Linear(head_dim * num_head, dim, bias=False)
inputs = torch.randn(bsz, seq_len, dim) # (1, 3, dim)

q = wq(inputs).view(bsz, seq_len, num_head, head_dim)
k = wk(inputs).view(bsz, seq_len, num_head, head_dim)
v = wv(inputs).view(bsz, seq_len, num_head, head_dim)

# q  =  bsz, seq_len, num_head, head_dim
# kt =  bsz, seq_len, head_dim, num_head
score = q @ k.transpose(2, 3) #
# score = bsz, seq_len, num_head, num_heads
# v =     bsz, seq_len, num_head, head_dim
output = score @ v # bsz, seq_len, num_head, head_dim ==> bsz,seqlen,dim
```

```
17  output = output.transpose(1, 2).reshape(bsz, seq_len, num_head * head_dim) #
    bsz, seq_len, dim
18  # 因为num_heads*head_dim等于dim
```

计算过程在第13行中，首先是将query矩阵和key.T矩阵进行矩阵相乘，对于这里的情况来说，query的维度为(bs, seqlen, heads, head_dim)，key的维度为(bs,seqlen，heads, head_dim)，随后再将query和keys矩阵进行矩阵相乘。

```
1   # q = bsz, seq_len, num_head, head_dim
2   xq = xq.transpose(1, 2)  # (bs, n_heads, seqlen, head_dim)
3
4   keys = keys.transpose(1, 2)  # (bs, n_heads, cache_len + seqlen, head_dim)
5   values = values.transpose( 1, 2)  # (bs, n_heads, cache_len + seqlen,
    head_dim)
6   scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
7
8
9   q = bsz, seq_len, num_head, head_dim ==> 1,1,num_head,head_dim
10  keys =  bsz, cache_len + seq_len, num_head, head_dim =
    1,cache_len+1,num_head,head_dim
11  每个block（线程块）处理head
12
13  q_head = head_dim
14  keys_head = (cache_len+1,head_dim)
15
16  q_head matmul key_head
```

cache_len历史步上所有的输入长度，seqlen = 1

由于在我们的项目中bs和seqlen均为1，所以key的维度就有(1, heads, cache_len+1, head_dim)，另外有dim = num_heads × head_dim。另外scores矩阵值的维度为(bs, heads, seqlen, head_dim)，根据刚才的预设信息，scores = (1,heads,1,head_dim)，同样地dim = heads × heads_dim。

随后我们来看我们的实现，由于我们每个线程块处理一个head，所以对于当前线程块key的维度为（cache_len+1,head_dim），所以我们在当前块中就是要将(cache_len+1,head_dim）的key矩阵和(1,head_dim）的query矩阵进行矩阵相乘。

query的维度为(1, 1, heads, head_dim)，key的维度为(1,1+cache_len，heads, head_dim)

对于每个线程块，一个线程块处理一个head，query_head = (1，head_dim)； key_head = (1+cache_len, head_dim)，依次取出(head_dim)维度的key值，**q_head** matmul **key_head = scores = (1,head_dim)@(head_dim, 1+cache_len) = (1,cache_len+1)。**

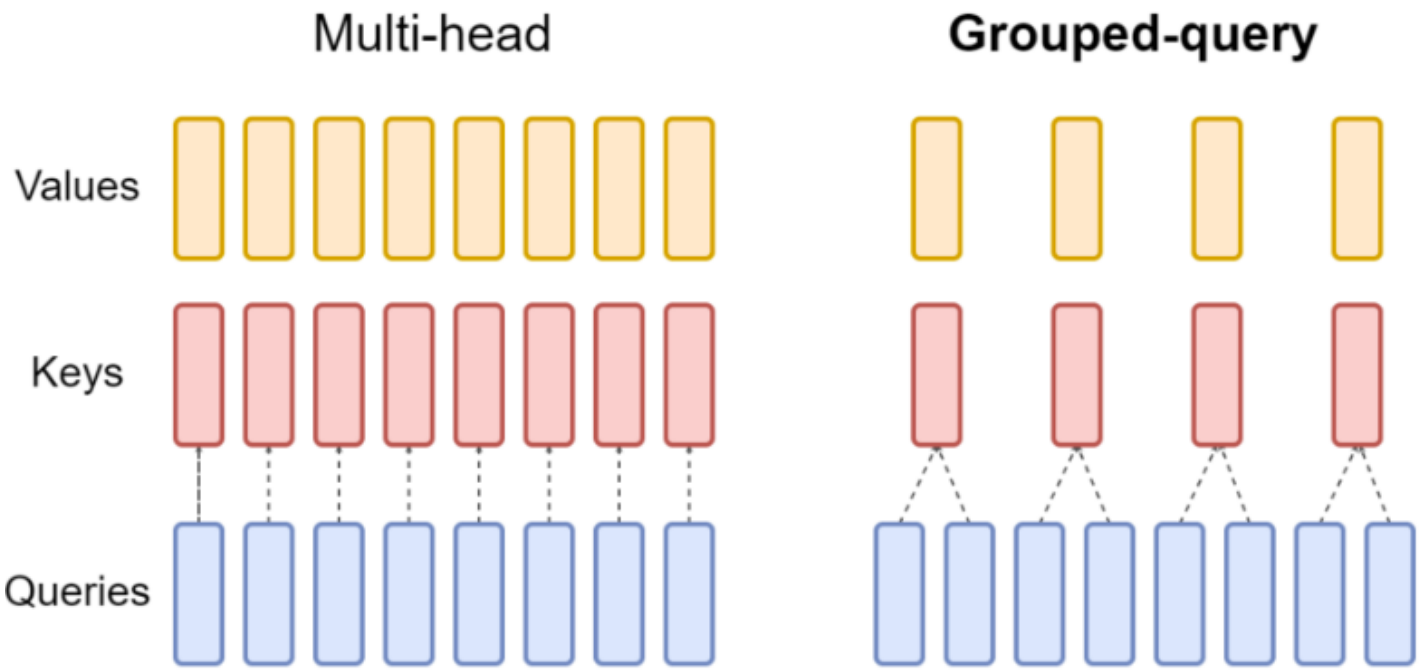
Key = (**1+cache_len, heads**, head_dim)，我们假定pos等于cache_len + 1，下方代码第17行中，我们的key实际是保存在上节课说的kv-cache当中，一整块kv-cache所有transformer块共享的，**key_cache + layer_offset 找到属于当前transformer块的key-cache起始地址。**

```
1   __global__ void multi_head_attention_kernel(int32_t pos, int32_t seq_len,
    float* query,float* score_ptr, float* output, float* key_cache,
2    float* value_cache, int32_t kv_dim, int32_t kv_mul, int32_t head_num, int32_t
     head_size,int32_t layer_offset) {
3       int head = blockIdx.x;
4       if (head >= head_num) {
5           return;
6       }
7       float* query_head = query + head * head_size; // (head_dim)
8       float* score_head = score_ptr + head * seq_len;
9       float scale = 1.f / sqrtf(head_size);
10      int32_t head_offset = (head / kv_mul) * head_size;
11      // kv_mul = 1
12      // head_offset = head*head_size
13      // kv_dim = dim
14      // t * dim t∈(0, pos)或者t∈(0, cache_len+1)
15      for (int t = threadIdx.x; t <= pos; t += blockDim.x) {
16          // key_head 等于(head_dim)
17          float* key_head = key_cache + layer_offset + t * dim + head_offset;
18          此处定位到kv-cache中属于当前transformer块的存储位置，随后key_head像
    query_head一样加上head_offset定位到第h个头的起始位置，随后再加上t×kv_dim对位到历史步
    和当前步长的位置，
19          其中t∈(0，pos)。另外，由于此处是Mha，所以kv_dim等于dim，综上：
20
21          // key_head = (1+cache_len,head, head_dim) dim = head * head_dim
22          // key_head[t,h] key_head, 一个head_dim维度的数组
23
24          key_head和query_head做一个内积
```

此处需要和上方对齐的是head_size就是head_dim只不过是变量名称上的不同，cuda的每个线程块(block)负责一个头的注意力计算。

```
1    float* query_head = query + head * head_size;
```

我们上文中说过，query的维度是head×dim，由于我们是多头注意力，query的维度为(head，head_size），所以定位到当前线程块处理的第head个头。有一点需要注意，第10行中的kv_mul，这里的kv_mul是gha(分组注意力中querys和keys、values的比值关系)，我们这里使用的是LLama7b，所以kv_mul恒等于1，如图所示。



所以对于维度为(heads，cache_len+1,head_dim）的key矩阵，key_head就是就是偏移到head_dim位置的，总共要经过cache_len次循环，**每次循环中都对head_dim个值求内积。**

再来看看key_head的索引方法，layer_offset是当前transformer块的偏移量，因为k被保存在一整个kv_cache中（前面的课时已经讲过了），所以我们要先定位到当前的transformer块对应的kv_cache起始位置，也就是加上layer_offset，也就是：

```
1   int32_t layer_offset = layer_index * seq_len * kv_dim;
```

此处用layer_offset定位到kv-cache中属于当前transformer块的存储位置，随后key_head像query_head一样加上head_offset定位到第h个头的起始位置，随后再加上t×kv_dim对位到历史步和当前步长的位置，其中t∈(0，pos)。另外，由于此处是Mha，所以kv_dim等于dim，综上：

```
1   key_cache + layer_offset + t * kv_dim + head_offset;
```

此处定位到了当前处理的时间步t，t∈(0，pos)的key张量中某个头的head_dim个值的起始位置，从此处开始就是计算对应位置和对应头的q@k。

q_head matmul **key_head scores = (1,head_dim)@(head_dim, 1+cache_len) = (1,cache_len+1)**

score_head 维度是**(1,cache_len+1)**

Key Head和Query Head 各自head_dim个值的内积，完成黄字的过程。

```
float score = 0.0f;
// query @ key 逐个头相乘，从上面的代码可以看出

for (int i = 0; i < head_size; i += 4) {
    float4 key_head_float4 = *reinterpret_cast<float4*>(key_head + i);
    float4 query_head_float4 = *reinterpret_cast<float4*>(query_head + i);
    ...
    ...
}
scale = 1./ sqrt(head_dim)
score *= scale;
score_head[t] = score; // q@k的值

// score_head
```

从此处开始计算query和key张量对应位置head_dim个元素的内积，内积值为score，我们将t∈(0,pos)位置的内积值，也就是score放入到score_head中，随后再对score_head数组中的多个score值求softmax。以上过程在Meta的llama2/3实现则有：

```
scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
scores = F.softmax(scores.float(), dim=-1).type_as(xq)
```

首先，让我们关注scale值的计算，它等于这里的math.sqrt(head_dim)。接下来，我们会在Python代码中看到对score张量的最后一个维度进行softmax运算的处理。已知scores的维度为 (bs, n_local_heads, seqlen, cache_len + seqlen)。以我们的示例来说，bs代表batch size，其值为1，seqlen也为1，因此q@k.T的结果scores维度实际上是(1, heads, 1, cache_len + 1)。

随后我们会对最后一维进行softmax运算，实际上就是对cache_len + 1的scores这个维度进行softmax处理，得到最后一维数值的方法是将长度为1的query（其维度为head_dim）与长度为cache_len + 1、维度同样为head_dim的key进行内积运算。这一步骤与我们的C++实现保持一致，就像我们上面说的那样。

可以看到求自注意力的最后一步是将scores乘以values，meta在pytorch中的实现如下：

```
values = values.transpose(
    1, 2
)  # (bs, n_local_heads, cache_len + seqlen, head_dim)
# (bs, n_local_heads, seqlen, cache_len + seqlen)
scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
...
...
output = torch.matmul(scores, values)
output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)
```

以上我们已经知道scores的维度为(bs,heads,1,cache_len+1)，对于当前头来说，**scores_head也就是(1, cache_len+1)**。values的维度为(bs,heads,cache_len+1,head_dim)，**对于当前头此处也就是values_head = (cache_len+1,head_dim)**，我们将cache_len个score值依次对values张量进行加权。

已知，value = (bs, n_heads, cache_len + seqlen, head_dim) = (1,cache_len+1,heads,head_dim)

layer_offset同样用于定位到当前value-cache的起始位置

t * kv_dim = t * dim

value[t,h] 得到一个value_head，value_head的维度是(head_dim)

```
// score_head经过了softmax运算
float* output_head = output + head * head_size;
head_offset = layer_offset + (head / kv_mul) * head_size;
for (int i = threadIdx.x; i < head_size; i += blockDim.x) {
    float output = 0.0f;
    #pragma unroll
    for (int t = 0; t <= pos; t++) {
        float* value_head = value_cache + head_offset + t * kv_dim;
        float score = score_head[t];  // 确定了score值
         output += score * value_head[i]; // 确定了values值
    }
    output_head[i] = output;
}
```

简单来说，这里就是在将(1,cache_len+1)维度的scores矩阵和(cache_len+1,**head_dim**)维度的value矩阵进行矩阵相乘并得到最终的结果放入到output_head中，与上方的output = torch.matmul(scores,values)达成了一致。

```
for (int i = threadIdx.x; i < head_size; i += blockDim.x)
```

在这个循环中我们定位到了values矩阵的第二维；

```
for (int t = 0; t <= pos; t++)
```

在这个循环中我们定位了values矩阵的第一维和scores矩阵的第二维，并依次对第i列的head_dim个数据进行加权求和。

Add算子的实现

在学习的过程中，add算子可能是我们接触到的一个最简单算子，它的作用就是将两个输入的张量进行逐个元素的加和并将结果写入到输出张量的对应位置中。

```
__global__ void add_kernel_cu_fp32(int32_t size, const float* in1, const
float* in2, float* out) {
  int32_t tid = threadIdx.x + blockDim.x * blockIdx.x;
  if (tid >= size) {
    return;
  }
  float in_val1 = in1[tid];
  float in_val2 = in2[tid];
  out[tid] = in_val1 + in_val2;
}
```

现在分别有两个指向输入的in1和in2指针，其中数据的数量为size个，我们配置了n个线程对size个数据进行逐个相加的工作。假设size为1024，线程数量n将它配置等于1024，总共有32个线程块，这样一来每个线程块中的线程数量是32。这里因为数据量不是特别多，所以我们配置一个线程处理一个位置元素的相加计算

```
int32_t tid = threadIdx.x + blockDim.x * blockIdx.x;
```

blockIdx.x表示线程块的编号，在上例中就是从0-31，因为总共有32个线程块，而线程块的容量为32（表示每块有4个线程），也就是blockDim.x等于32. 我们用python来模拟这个过程，可以看出线程号计算依次打印出0-1024。

```
In [2]: blockDimx = 32

In [3]: blockCount = 32

In [4]: for blockIdx in range(blockCount):
   ...:     for threadIdx in range(blockDimx):
   ...:         print(threadIdx + blockDimx * blockIdx)
   ...:
0
1
2
```

```
3
4
5
6
7
8
...
...
...
1023
```

所以`int32_t tid = threadIdx.x + blockDim.x * blockIdx.x`的值依次是0到1023，每个线程负责处理一个位置的输入数据。

```
float in_val1 = in1[tid];
float in_val2 = in2[tid];
out[tid] = in_val1 + in_val2;
```

随后这多个线程分别取出对应位置tid的值，in_val1和in_val2进行相加并将值放入到输出out[tid]中，另外我们要知道和CPU计算不同的是，这1024个线程不是依次执行的，而是并行执行的，也就是线程id从0到1023并没有执行的先后顺序。

Embedding算子的实现

比如现在有一个大小为65200×512的词表，65200就表示对应输入的单词数量，512就表示映射后的向量维度，我们以下图作为一个例子，图中表的一行表示一个单词，表中的一列表示单词对应的向量。

1	2	3	4
4	3	1	7
6	1	3	4

以以上的表格为例我们的单词数量为8，每个单词的维度是4。如果我们的输入编号是[0，3，6]，那么我们Embedding算子将取出对应位置的单词向量，以上表格为例，取出对应的向量为：

[1,2,3,4]

[4,3,1,7]

[6,1,3,4]

综上来说，我们需要为这个算子准备三个输入，一个是词表权重，在本例中就是这个以上的8×4大小的表格，以及需要取出的输入编号，也就是[0,3,6]，最后一个输入将用于索引到的权重向量，也就是以上的三个向量。**所以在Cuda算子的实现中，我们配置了N个线程块，N表示单词的数量，每个线程块中的线程数量为M，M个线程会负责搬运每个单词对应的weight_dim个数值。**

```
__global__ void emb_kernel_cu_fp32(int32_t vocab_size, int32_t token_num, int32_t weight_dim,
                                   const int32_t* input_ptr, const float* weight_ptr,
                                   float* output_ptr) {
  int32_t token_idx = blockIdx.x;
  if (token_idx >= token_num) {
    return;
  }
  int32_t token = input_ptr[token_idx];
  if (token >= vocab_size) {
    return;
  }

  float* output_ptr_start = output_ptr + token_idx * weight_dim;
  const float* weight_ptr_start = weight_ptr + token * weight_dim;

  for (int32_t i = threadIdx.x; i < weight_dim; i += blockDim.x) {
    output_ptr_start[i] = weight_ptr_start[i];
  }
```

换句话说，我们需要将input token size×weight dim个数据复制或者说搬运到输出空间中，那么我们就配置input token size个线程块，每个线程块中的所有线程会一起搬运对应的weight dim个权重数据。

```
  int32_t token_idx = blockIdx.x;
  if (token_idx >= token_num) {
    return;
  }
```

获取到当前线程块需要处理的单词编号，对于上例的输入来说，token_idx的值依次是0，3，6。从以下的代码的代码中定位到所需要取的词表行：

```
    float* output_ptr_start = output_ptr + token_idx * weight_dim;
    const float* weight_ptr_start = weight_ptr + token * weight_dim;
```

weight_ptr + token * weight_dim; 这行代码表示我们现在需要定位到词表的某一行，同样地，output_ptr_start指向输出张量某一行的起始地址，比如现在我们要搬运词表中的第0行，[1,2,3,4] 那么weight_ptr_start就指向1所在的位置。

```
    for (int32_t i = threadIdx.x; i < weight_dim; i += blockDim.x) {
      output_ptr_start[i] = weight_ptr_start[i];
    }
```

随后就是将每个单词对应的dim维度的向量搬运到对应的输出张量中。