第6次课程-Nsight compute的使用和算子的优化

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

在开始本节课的内容学习之前请务必先学习第5次课程：

第五次课程-RMSNorm算子的CUDA实现

视频

2024-07-10 23-00-59.mkv

另外一种写法

在之前的实现中，我们进行了同一 warp 内线程之间的规约操作。然而，这种方法存在一个问题：当你的 blockDim，即每个 block 中的线程数量，大于一个 warp 的大小时（通常为32），会导致多个线程具有相同的 lane_id（例如，多个线程的 lane_id 都是0，多个是1等），从而引起重复计算的情况。

```
1    const int lane_id = tid % warpSize;
```

`tid ∈ 0, ..., 1023` （表示线程 ID 从 0 到 1023），将线程id取模 `warpSize` 后得到 `lane_id ∈ 0, ..., 30` （表示 lane id 的范围）。因此，多个线程会共享相同的 `lane_id`，例如 `lane_id == 0` 会被多个线程拥有。

```
1   for (int i = 0; i < size; i += 32) {
2       sum += in[i] * in[i];
3   }
4
5   tid = 0
6   tid = 32
7   tid = 64 都在这么做
```

```
1   for (int i = 1; i < size; i += 32) {
2       sum += in[i] * in[i];
3     }
4
5   tid = 1, in[1]+in[33]+in[65]
6   tid = 33 in[1]+in[33]+in[65]
7   tid = 65 都在这么做。对同一个下标下的数据进行加和。
```

尽管我已经将每个block内的线程数量设置为了1024个，但实际上参与有效运算的线程数目却显得不足。具体来说，我现在设定的是每个block中有64个线程，而一个warp的标准大小是32个线程。因

此，线程索引（tid）的范围是从0到63。当tid在0到31之间时，对应的lane_id（即warp内的相对位置）也处于0到31的区间；而当tid位于32到63之间时，lane_id同样覆盖0到31的范围。

在一个warp执行规约操作时，32个线程会对其局部数据进行归约处理：

在第一轮迭代中：

- lane_id 0的数据与lane_id 16的数据相加；
- lane_id 1的数据与lane_id 17的数据相加；

进入第二轮迭代：

- 在这一阶段，lane_id 0（初始值加上来自lane_id 16的贡献）将再次与lane_id 8（其初始值加上来自lane_id 24的贡献）相加；

但是这里有一个问题，在每次迭代开始前，多个不同的线程lane id都会等于0。例如，当tid为0、32或64时，lane_id均为0，这导致该位置的数据被不同线程重复计算。

```
1   static __global__ void row_rmsnorm_f32(const float* in, const float* wei,
    float* out, const int size, const float eps) {
2     const int tid = threadIdx.x; // 线程 id 多个线程是同时进来的。
3     const int lane_id = tid % warpSize; // 这个线程id在warp内的编号
4
5     float sum = 0.0f;
6     // printf("lane id:%d\n", lane_id);
7     // lane_id从0到31都有，是同时执行的
8     for (int i = lane_id; i < size; i += warpSize) {
9       sum += in[i] * in[i];
10    }
11    using WarpReduce = cub::WarpReduce<float, 32>;
12    __shared__ typename WarpReduce::TempStorage temp;
13    __shared__ float shared_val;
14    sum = WarpReduce(temp).Reduce(sum, cub::Sum());
```

如果我们配置了blockDim等于128，看随后的输出打印中lane_id = 0一共被打印了4次，也就是说一共有4个线程都执行了如下相同的运算，演示代码在：https://github.com/zjhellofss/KuiperLLama/blob/course6/kuiper/source/op/kernels/cuda/rmsnorm_kernel.cu：

```
1   for i = lane id; i < size; i += warpSize
```

1. 我们将blockDim配置成128的情况，也就是一个block有128个线程。

2. 我们将blockDim配置成32的情况，也就是一个block有32个线程，这种情况下不会有线程重复计算的情况发生，也就是 `printf` 不同 `lane_id` 的次数只会有一个。请看视频演示。

劣势

在以上的这种写法中，实际上只有32个线程参与计算工作，就如上面分析的那样，就算配置超过32个线程的block块中实际有大量做重复计算的线程。

改进后的写法

为了让尽可能多的线程参与工作，我们将原先基于warp规约改为block之间规约。在开始分析代码之前，我们先假设一个block中的线程数量是128个，需要处理的数据是1024个，为了简单起见，我们数据值均为1。我们打算采用如下的规约模式：

1. 每个线程需要处理的数据数量是8个，因为1024/128 = 8；
2. 以线程0为例，它负责将数据下标0，32，64，... 位置上共32个数据求和，以线程1为例，它负责将数据下标1，33，65，... 位置上共32个数据的求和，以此类推，每个线程的求和结果均为8；

怎么对128个线程上的局部和做规约?

1. 对128个线程每32个进行一次reduce，每32个也就是以一个warp（共32个线程）作为单位进行规约。也就是说thread 0-31（同属一个warp）做一次规约，将值存到共享内存中sh_mem中。线程32-63做一次规约，将值存到共享内存中下一个位置中。线程64到95做一次规约，将值存到共享内存的再下一个位置中。以32个线程为一个单位，做一次规约。
2. 经过上一步骤，**在共享内存中sh_mem数组中，**第0个位置存放线程0-31的求和结果，也就是256，第1个位置存放线程32-63的求和结果，同样是256。第2个坐标位置存放线程64-95的求和结果，同样是256。第3个坐标位置存放线程96-127的求和结果。
3. 在第2步的基础上，我们需要对存储在共享内存 `sh_mem` 中的局部求和结果进行进一步的规约求和操作，以得到全局总和（即对这四个局部求和结果进行规约）。具体来说，就是将这四个局部和数值相加，从而得出最终的全局总和。

我们来看看具体实现：

```
template<int32_t BLOCK_DIM>
static __global__ void row_rmsnorm_f32(const float* in, const float* wei,
float* out,
                                       const int size, const float eps) {
  const int tid = threadIdx.x;

  float sum = 0.0f;
    // tid = 0 , in[0]+in[128]+in[256]+...+
  for (int i = tid; i < size; i += blockDim.x) {
    sum += in[i] * in[i];
  }

```

```
12    using BlockReduce = cub::BlockReduce<float, BLOCK_DIM>;
13    __shared__ typename BlockReduce::TempStorage temp;
14    __shared__ float shared_val;
15    sum = BlockReduce(temp).Sum(sum);
16    if (threadIdx.x == 0) {
17      shared_val = sum;
18    }
19    __syncthreads();
20    sum = shared_val;
21    const float scale = rsqrtf(sum / static_cast<float>(size) + eps);
22    for (int i = tid; i < size; i += blockDim.x) {
23      out[i] = scale * in[i] * wei[i];
24    }
25  }
```

首先看代码6-9行，我们可以看到的是一个块（block）中的所有线程正在对自己负责的数据进行累加。例如，线程0负责累加位于数据索引0、128、256处的值；线程1负责累加位于索引1、129、257处的值；线程2负责累加位于索引2、130、258处的值，以此类推。每个线程计算出来的求和结果存储在其本地寄存器变量 `sum` 中。

我们先来看看 `BlockReduce` 的一种可能得实现

```
1   template<const int NUM_THREADS=128>
2   // 传进来的参数val，是每个线程累计的局部和
3   __device__ __forceinline__ float block_reduce_sum(float val) {
4     // always <= 32 warps per block (limited by 1024 threads per block)
5     constexpr int NUM_WARPS = (NUM_THREADS + WARP_SIZE - 1) / WARP_SIZE;
6     int warp = threadIdx.x / WARP_SIZE;
7     int lane = threadIdx.x % WARP_SIZE;
8     static __shared__ float shared[NUM_WARPS]; // 共享内存数据
9
10    val = warp_reduce_sum<WARP_SIZE>(val);
11      // 0-31线程上的做一个规约，32-63线程上的做一个规约，32个线程的各自规约结果放到val
    上。
12    if (lane == 0)
13        shared[warp] = val; // shared[0]存放0-31个线程上的结果，shared[1]存放32-63
    上的结果。
14    __syncthreads();
15
16     // tid = 0的时候，val = shared[0] ,
17     // tid = 1的时候，val = shared[1]
18     // tid = 2的时候，val = shared[2]
19     // tid = 3的时候，val = shared[3]
20    val = (lane < NUM_WARPS) ? shared[lane] : 0.0f;
21    val = warp_reduce_sum<NUM_WARPS>(val);
```

```
22      return val;
```

1. 在第9行中，我们以32个线程为一个单位进行规约操作，即在一个 warp 内进行规约（这一过程已在前面详细描述过）。
2. 随后，我们将线程0至线程31的局部求和结果存放到共享内存 `shared[0]` 中，线程32至线程63的局部求和结果存放到共享内存 `shared[1]` 中，依此类推。
3. 接下来，在第14行，我们将 `shared[0]` 到 `shared[3]` 这四个局部和进行进一步的累加操作。这个累加同样在一个 warp 内完成，具体而言，是由 warp 内的线程0至线程3 负责累加 `shared[0]` 至 `shared[3]` 的值。而该 shared_mem数组中剩下的位置（即下标4至下标31），由于没有额外的数据需要处理，会被分配一个值为0的数据来进行累加操作，但这不会影响最终的累加结果。

使用Nsight-Compute查看指标

分析方法

进入你Cuda的安装路径，比如我安装在以下路径中：

```
1   /usr/local/cuda-11.8/nsight-compute-2022.3.0
```

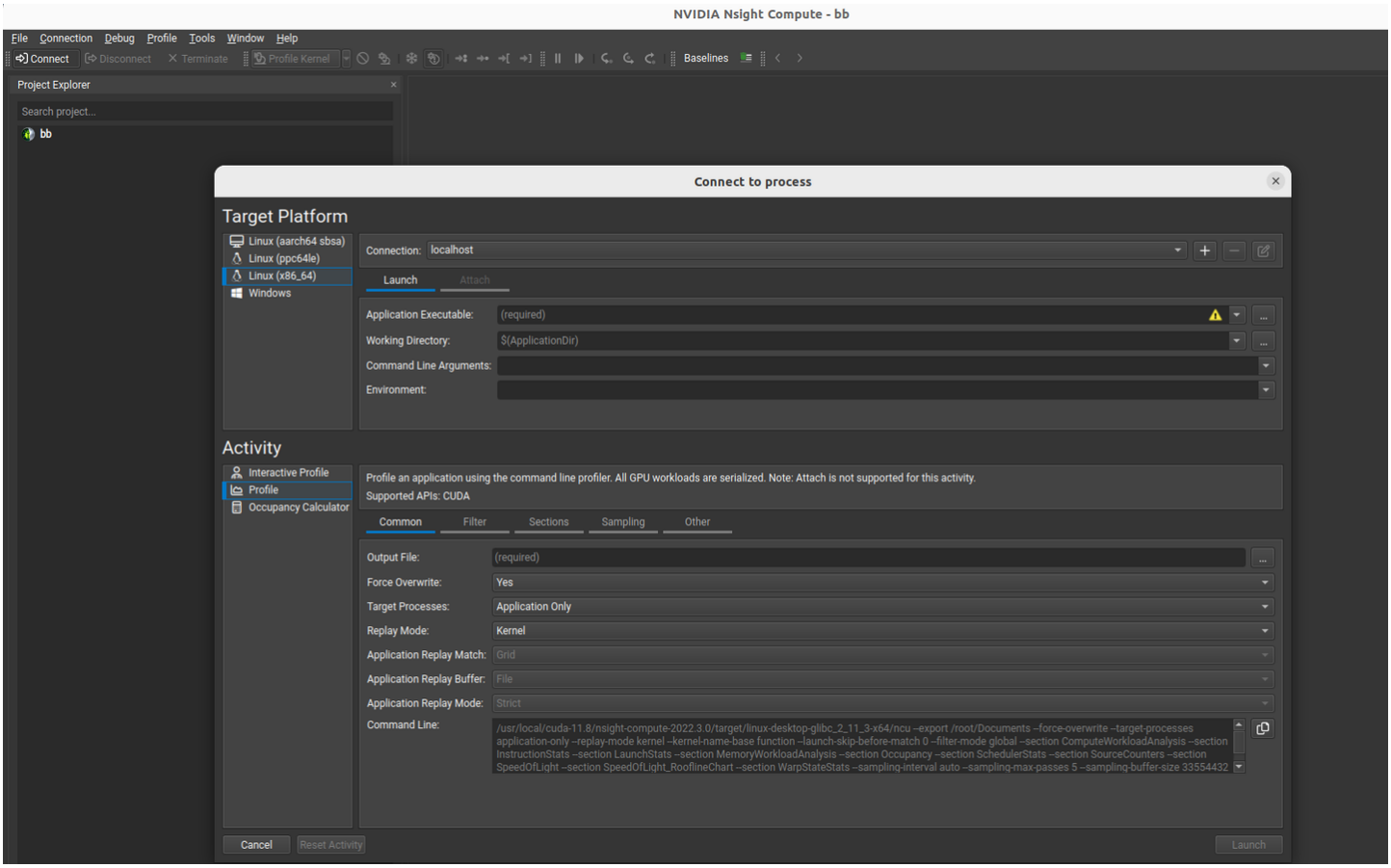
为什么不直接点击Nsight-Compute的图标启动程序呢？主要原因是因为分析核函数的运行状态需要root权限

```
1   sudo ./nv-nsight-cu
```

选择菜单栏的Project，New Project：

选择Launch部分，填写：

1. Application Execuable：可执行程序
2. Working Directory：可执行程序的工作目录
3. Comand Line Arguments：命令行参数

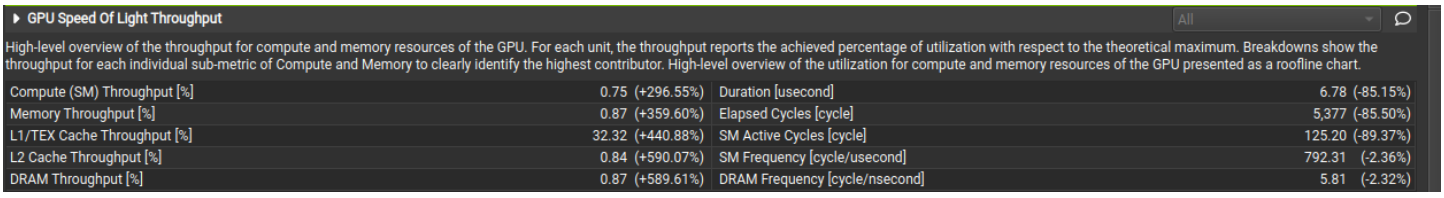


选择Activity中的profile部分，填写：

1. Output file
2. Filter: 选择你要调试的核函数名称

随后就可以点击右下角的Launch开始分析。

结果分析

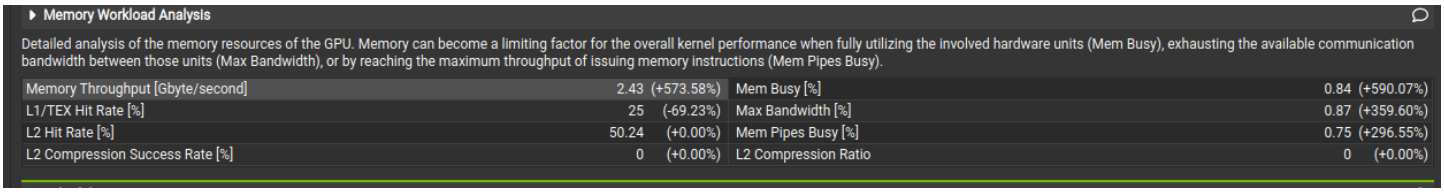


我们可以看到，和基于warp的写法相比，以下的指标有了显著的提升:

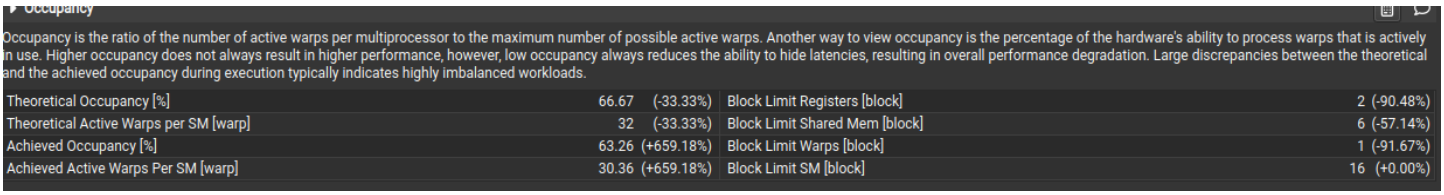
1. **Elapsed Cycles [cycle]**: 核函数执行过程中经过的总周期数，顾明思义，因为参与有效计算的线程数量变多了自然能更快地完成工作。
2. **Duration**: 核函数的执行时间，和第一条同理。

另外，**左侧的计算吞吐量(Compute Throughput)、内存吞吐量(Memory Throughput)和两级Cache吞吐量都有显著的增长，更多的线程在同一时刻处理更多数据**。

我们再来看具体的内存资源指标分析：



可以看出在内存资源指标分析中， 内存吞吐率(Memory Throughput)虽然不高（主要是数据量小），但是和基于Warp的版本相比却有了明显的提升。



Achieved Active Warps Per SM [warp]: 这表示每个流式多处理器（SM）上实际活跃的线程束（warp）的数量。从这一指标来看，基于 block 的实现方法相比基于 warp shuffle（通常称为 warp reduce）的方法，具有更高的活跃 warp 数量。此外，实际的 SM 占用率（Achieved Occupancy）也较高，这里的占用率是指实际活跃的 warp 数量与理论最大活跃 warp 数量之比。那么为何基于 warp shuffle 的方法占用率较低呢?

当 SM 的计算吞吐量较低时，这通常是因为线程束的占用率不足。这意味着没有足够的线程来掩盖内存访问延迟，导致多数线程大部分时间都在等待数据准备，而非积极参与计算。具体来说，在一个 warp 中，如果有部分线程在等待数据准备，那么整个 warp 都不能有效地被调度来进行计算任务。由于 warp 是 GPU 调度的基本单位，因此，即使只有一个或少数几个线程因等待数据而停滞，整个 warp 的执行效率也会受到显著影响。”

结果对比

见单元测试test/test_op/test_cu_rmsnorm.cpp，请看视频演示