Pytorch 算子的实现与优化——Transformer

CS433 并行与分布式程序设计

Name: 易文龙　张若涵　　Student ID: 519030910068　519030910029

1 Pytorch 框架 CUDA 算子实现方法

1.1 调研 Transformer 网络用到的 GPU 算子情况

利用助教提供的 kernel_count.py 脚本，分别输出 CPU 环境下和 GPU 环境下执行 Transformer 模型推理时底层设备 kernel 的调用情况。

```
epsilon : 44
check_uniform_bounds : 44
uniform_kernel_cpu : 44
fill_cpu : 44
copy_kernel : 44
addmm_impl_cpu_ : 44
fill_out : 44
copy_ : 44
div_cpu : 44
baddbmm_with_gemm : 44
softmax_lastdim_kernel_impl : 44
bernoulli_scalar_cpu_ : 44
mul_cpu : 44
LayerNormKernelImpl : 44
clamp_min_cpu : 44
#Dispatch : 44
```

图 1: CPU 环境下底层设备 kernel 调用

```
epsilon : 44
check_uniform_bounds : 44
uniform_kernel_cpu : 44
fill_cpu : 44
copy_kernel : 44
copy_ : 44
addmm_cuda : 44
fill_out : 44
div_true_cuda : 44
baddbmm_cuda : 44
host_softmax : 44
fused_dropout : 44
LayerNormKernelImpl : 44
clamp_min_scalar_cuda : 44
```

图 2: GPU 环境下底层设备 kernel 调用

如图所示，可以看到 Transformer 网络在调用了 7 个 GPU 算子来提升运行效率，分别是:

- $addmm\_cuda$
- $baddbmm\_cuda$
- $host\_softmax$
- $fused\_dropout$
- $LayerNormKernelImpl$
- div_true_cuda
- $clamp\_min\_scalar\_cuda$

对前四个进行进一步调研。

1.2 *addmm_cuda* 和 *baddbmm_cuda*

addmm 算子用来执行矩阵 mat1 和 mat2 的矩阵乘法。矩阵 input 被添加到最终结果中。baddbmm 算子用来执行 batch1 和 batch2 执行批量的矩阵相乘，然后和输入相加，得到最终结果。

在矩阵乘法中，进行了不同数据的大量相同计算操作（相乘并累加），这种计算是特别适合使用 GPU 来计算，因为 GPU 拥有大量简单重复的计算单元，通过并行就能极大的提高计算效率。

Pytorch 中最终通过调用 cublasSgemm CUDA 库函数来实现矩阵乘法。

```
template <>
void gemm<float>(CUDABLAS_GEMM_ARGTYPES(float)) {
  // See Note [Writing Nondeterministic Operations]
  globalContext().alertCuBLASConfigNotDeterministic();
  cublasHandle_t handle = at::cuda::getCurrentCUDABlasHandle();
  cublasOperation_t opa = _cublasOpFromChar(transa);
  cublasOperation_t opb = _cublasOpFromChar(transb);
  _cublasAdjustLdLevel3(transa, transb, m, n, k, &lda, &ldb, &ldc);
  GEMM_CHECK_ARGVALUES(float);
  TORCH_CUDABLAS_CHECK(cublasSgemm(
      handle, opa, opb, m, n, k, &alpha, a, lda, b, ldb, &beta, c, ldc));
}
```

图 3: **cublasSgemm CUDA 库函调用**

1.3 *host_softmax*

因为 softmax 的输入常为长向量或是大矩阵，为提高计算效率，考虑针对公式并行处理。

$$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

常见的并行思路为：

- ReduceMax（并行）：得到输入每一行的最大值
- BroadcastSub: 各点减去最大值，得到 z_j
- Exp: 求 e^{z_j}
- ReduceSum（并行）：对 e^{z_j} 求和
- BroadcastDiv：除法得到结果

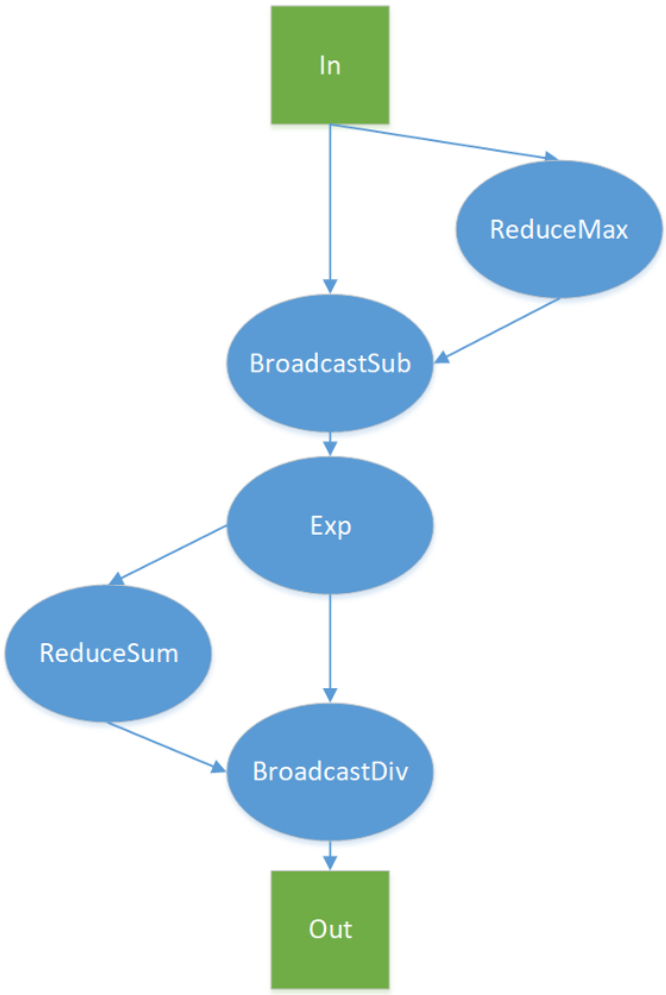

图 4: **Softmax 并行思路**

Pytorch 中 cunn_SoftMaxForward kernel 实现了 host_softmax 的核心功能 cunn_SoftMaxForward kernel 中先实现了 ReduceMax 和 ReduceSum 这两个步骤，然后借助实现的 Epilogue 整体计算中间三个步骤。

```
// find the max
accscalar_t threadMax = ilpReduce<MaxFloat, ILP, scalar_t, accscalar_t>(
    shift, input, classes, MaxFloat<scalar_t, accscalar_t>(), -at::numeric_limits<accscalar_t>::max());
accscalar_t max_k = blockReduce<Max, accscalar_t>(
    sdata, threadMax, Max<accscalar_t>(), -at::numeric_limits<accscalar_t>::max());

// reduce all values
accscalar_t threadExp = ilpReduce<SumExpFloat, ILP, scalar_t, accscalar_t>(
    shift, input, classes, SumExpFloat<scalar_t, accscalar_t>(max_k), static_cast<accscalar_t>(0));
accscalar_t sumAll = blockReduce<Add, accscalar_t>(
    sdata, threadExp, Add<accscalar_t>(), static_cast<accscalar_t>(0));

Epilogue<scalar_t, accscalar_t, outscalar_t> epilogue(max_k, sumAll);
```

图 5: **Pytorch host_softmax 实现核心**

1.4 *fused_dropout*

Dropout：设置一个固定的概率 p. 对每一个神经元都以概率 p 随机丢弃（即置 0）。因为 Dropout 的输入常为长向量或是大矩阵，为提高计算效率，考虑并行处理。

Pytorch 中 fused_dropout_kernel 实现了 fused_dropout 的核心功能，基于 totalElements 数分块并行，随机生成分块概率矩阵筛选元素。

```
IndexType rounded_size = ((totalElements - 1)/(blockDim.x * gridDim.x * UNROLL)+1) *
    blockDim.x * gridDim.x * UNROLL;
for (IndexType linearIndex = idx;
    linearIndex < rounded_size;
    linearIndex += gridDim.x * blockDim.x*UNROLL) {
```

图 6: **Pytorch fused_dropout 实现核心**

2 Pytorch 框架中对 CUDA Runtime API 的调用

2.1 CUDA Runtime API

Runtime API 是基于 Driver API 之上开发的一套 API。Runtime API 内部封装了各情况下为线程使用何种 context，避免了涉及 context 的底层操作。Runtime API 基本都是以 cuda 开头的。

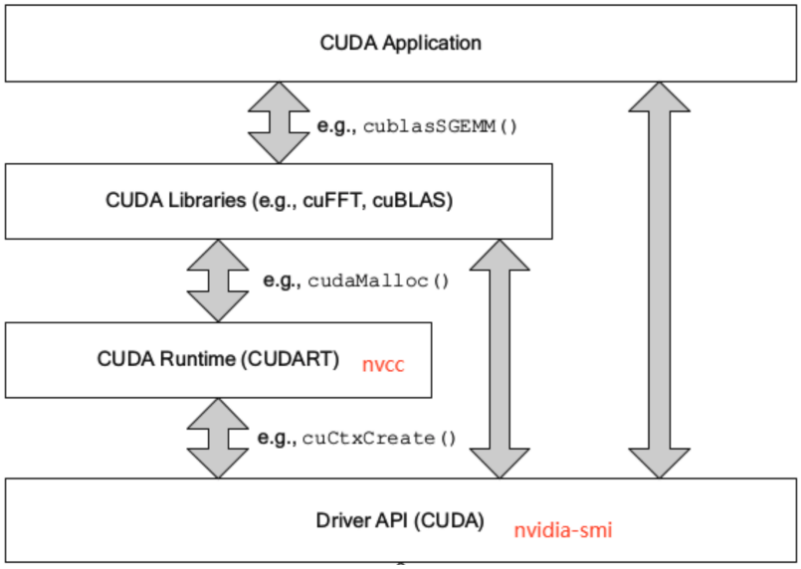

图 7: **CUDA Runtime API**

2.2 Pytorch 调用 CUDA Runtime API

CUDA Runtime 相关头文件: #include <cuda_runtime.h> 和 #include <cuda_runtime_api.h>，其中 cuda_runtime_api.h 是 cuda_runtime.h 的子集。cuda_runtime_api.h 是纯 C 接口和实现。

Pytorch 主要在以下文件目录下引用 CUDA Runtime 相关头文件：aten/ 和 c10/。torch/csrc/ 和 caffe2/等目录下也有调用了部分关于设备和内存信息的 API。

2.2.1 aten/目录下的 CUDA Runtime API

具体算子文件中，如 SoftMax.cu，SparseCUDATensorMath.cu，SparseCsrTensorMath.cu，SparseMatMul.cu，在算子的计算过程中调用了 CUDA Runtime API。如：

- cudaMemcpyAsync, cudaMemcpy：在主机和设备之间拷贝数据
- cudaGetDevice: 获取当前正在使用的设备

在 cache 分配相关文件中，如 CachingHostAllocator.cpp，CUDACachingAllocator.cpp 调用了大量 CUDA Runtime API。如：

- cudaMalloc, cudaFree, cudaMemGetInfo: 分配查询内存

- cudaHostAlloc, cudaFreeHost: 分配页锁定内存
- cudaIpcOpenMemHandle, cudaIpcCloseMemHandle: 进程间内存句柄
- cuda 事件相关操作：cudaEventQuery，cudaEventDestroy，cudaEventRecord，cudaEventElapsedTime，cudaEventSynchronize，cudaEventCreateWithFlags

2.2.2 c10/目录下的 CUDA Runtime API

CUDAFunctions.h 中利用 cudaMemcpyAsync, cudaStreamSynchronize 封装了 C10_CUDA_API memcpy_and_sync stream_synchronize。

CUDAStream.h 利用 cudaStreamQuery, cudaStreamGetPriority,cudaDeviceGetStreamPriorityRange 封装了 query, priority 等新接口，供 c10/目录下其他文件调用。

2.3 Pytorch 中最常调用的 CUDA Runtime API——cudaGetLastError

cudaGetLastError：返回运行时调用的最后一个错误。在 Pytorch 中常用来进行异常处理，返回报错信息。

```
bool query() const {
  if (!is_created_) {
    return true;
  }

  cudaError_t err = cudaEventQuery(event_);
  if (err == cudaSuccess) {
    return true;
  } else if (err != cudaErrorNotReady) {
    C10_CUDA_CHECK(err);
  } else {
    // ignore and clear the error if not ready
    cudaGetLastError();
  }

  return false;
}
```

图 8: **cudaGetLastError 异常处理**

2.4 CUDA Runtime API PyTorch 背景下的作用

CUDA Runtime API 帮助 Pytorch 实现以下功能：

- 异常处理，返回错误信息
- 内存、内存间通信相关操作
- 流相关操作
- 事件相关操作

通过 CUDA Runtime API 的功能封装，可以进一步简化 PyTorch 中 cuda 算子的实现。

3 CUDA 算子优化

3.1 SoftMax.cu 优化

优化原先 cunn_softMaxForward kernel 中 blockReduceSum 的 warp reduce 部分。利用 Warp 级别原语 ___shfl_xor_sync，提升 warp reduce 部分效率。

```
__device__ __forceinline__ AccumT
blockReduce(AccumT* smem, AccumT val,
            const Reduction<AccumT>& r,
            AccumT defaultVal)
{
  // To avoid RaW races from chaining blockReduce calls together, we need a sync here
  __syncthreads();

  smem[threadIdx.x] = val;

  __syncthreads();

  AccumT warpVal = defaultVal;

  // First warp will perform per-warp reductions for the remaining warps
  uint32_t mask = (((uint64_t)1) << (blockDim.x / C10_WARP_SIZE)) - 1;
  if (threadIdx.x < C10_WARP_SIZE) {
    int lane = threadIdx.x % C10_WARP_SIZE;
    if (lane < blockDim.x / C10_WARP_SIZE) {
#pragma unroll
      for (int i = 0; i < C10_WARP_SIZE; ++i) {
        warpVal = r(warpVal, smem[lane * C10_WARP_SIZE + i]);
      }
```

图 9: **原始 BlockReduceSum 代码**

```
template <template<typename> class Reduction, typename AccumT>
__device__ __forceinline__ AccumT
blockReduceSum(AccumT* smem, AccumT val,
               AccumT defaultVal)
{
  // To avoid RaW races from chaining blockReduce calls together, we need a sync here
  __syncthreads();

  smem[threadIdx.x] = val;

  __syncthreads();

  AccumT warpVal = defaultVal;

  // First warp will perform per-warp reductions for the remaining warps
  uint32_t mask = (((uint64_t)1) << (blockDim.x / C10_WARP_SIZE)) - 1;
  if (threadIdx.x < C10_WARP_SIZE) {
    int lane = threadIdx.x % C10_WARP_SIZE;
    if (lane < blockDim.x / C10_WARP_SIZE) {
#pragma unroll
      for (int msk = 16; msk > 0; msk >>= 1){
        warpVal += __shfl_xor_sync(0xffffffff, warpVal, msk, 32);
      }
```

图 10: **优化后 BlockReduceSum 代码**

表 1: softmax 测试结果

	优化前	优化后
10 次 softmax 耗时 1	0.5590	0.5126
10 次 softmax 耗时 2	0.5467	0.5084
平均耗时	0.5529	0.5105

3.2 softmax kernel 优化结果

```
(labpy37) group12@38f495981d79:~$ python test_softmax_dropout.py
Softmax elapsed time: 0.5126
(labpy37) group12@38f495981d79:~$ python test_softmax_dropout.py
Softmax elapsed time: 0.5084
(labpy37) group12@38f495981d79:~$ conda deactivate
(base) group12@38f495981d79:~$ conda activate py37
(py37) group12@38f495981d79:~$ python test_softmax_dropout.py
Softmax elapsed time: 0.5590
(py37) group12@38f495981d79:~$ python test_softmax_dropout.py
Softmax elapsed time: 0.5467
```

图 11: **优化后 softmax kernel 效果**

3.3 Gemm 优化

在本次实验中我们对 Cublas 的矩阵乘法进行了重写，实现思路如下：

- 基础的实现
- 矩阵分块与资源分配

显然我们不能只使用一个 block 计算一个超大矩阵，这样会造成大量 SM（Streaming Multiprocessor）的闲置浪费，这就需要对矩阵进行分块计算。

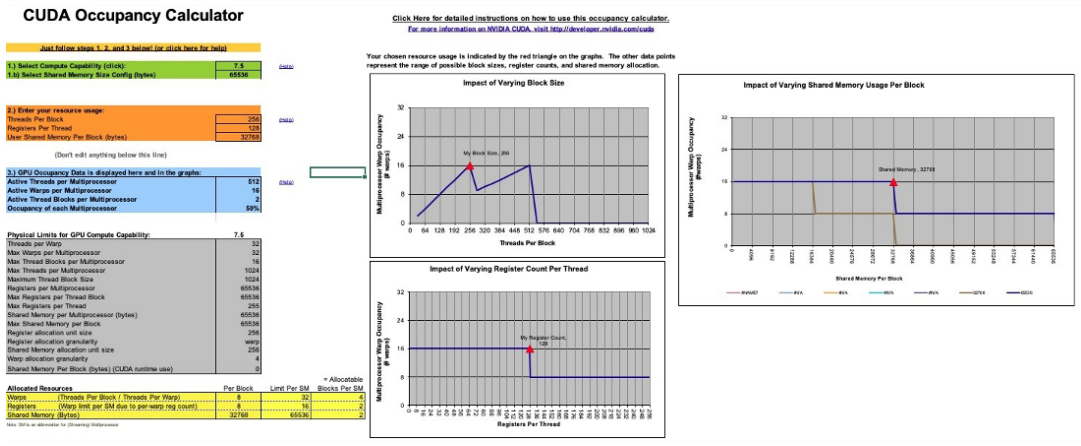

图 12: **gemm 效率调研**

在选取了合适的 block 资源配置，利用 Shared Memory 降低访存延迟，做好循环展开之后，SGEMM Kernel 的性能已经能达到一个不错的水平。

```
Miniconda3-4.6.14-Linux-x86_64.sh  kernel_count.py  lab2_baseline  miniconda
(py37) group12@38f495981d79:~$ python test_transformer.py
Transformer elapsed time: 4.0533
(py37) group12@38f495981d79:~$ conda deactivate
(base) group12@38f495981d79:~$ conda activate labpy37
(labpy37) group12@38f495981d79:~$ python test_transformer.py
Transformer elapsed time: 3.5578
(labpy37) group12@38f495981d79:~$
```

图 13: **gemm 优化效率测试**

3.4 进一步提升

我们可以使用向量读取指令 LDS.128 优化 Shared Memory 访问（对应 float4 数据类型），这能大幅减少访存指令的数量，进一步提升计算访存比，由此我们需要将 A 矩阵做一次转置我们的 kernel 为 256 个线程计算 128x128 的分块，为了能够合并访问 Shared Memory，我们将 256 个线程划为二维最终单个线程计算 2x2 个 4x4 的结果，如下图所示。

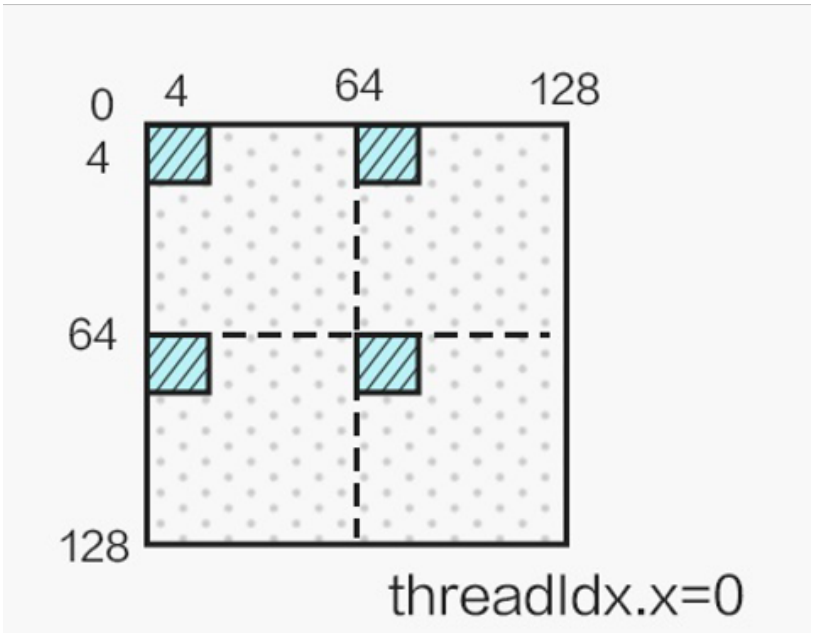

图 14: **线程划分示意图**

代码实现见 Aten/native/cuda/cuSgemm_refactory.cu 文件。

4 实验结果

借助助教提供的 test.py 进行优化效果测试。运行 3 次结果分别如下。

```
================================================
Test1 diff: 0.07940991967916489
Test2 diff: 0.09208536893129349
Test1 running time: 21170151 ns
Test2 running time: 36616292 ns
================================================
```

图 15: **第 1 次测试结果**

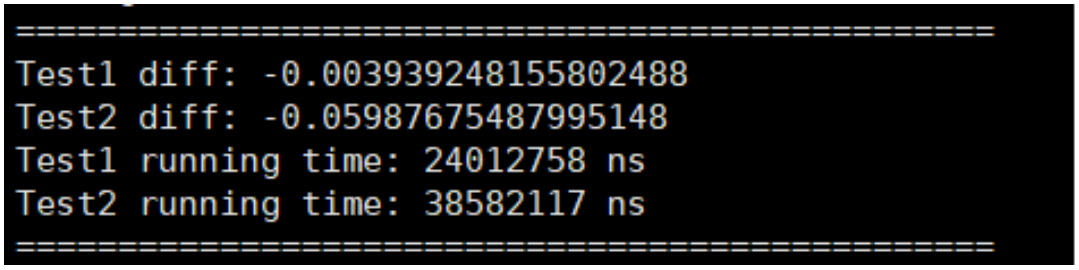

图 16: **第 2 次测试结果**

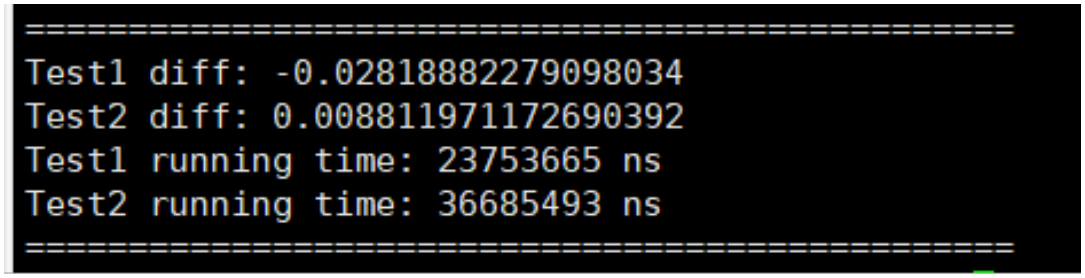

图 17: **第 3 次测试结果**

表 2: 测试结果

	Trial 1	Trial 2	Trial 3	Avg.
Test1 diff	0.0794	-0.0039	-0.0282	0.0158
Test2 diff	0.0921	-0.0599	0.0088	0.0137
Test1 running time	21170151ns	24012758ns	23753665ns	22978858ns
Test2 running time	36616292ns	38582117ns	36685493ns	37294634ns