Custom CUDA operation for Tensorflow

Oleksandr Myronov

Overview

- Tensorflow provides C++ API for defining new operations, including GPU
- The kernel for the operation must be defined in separate file and compiled using nvcc
- The operation class is defined in another file and compiled with gcc, producing the library
- The library can be loaded into TF runtime and used as any other operation
- You must have tensorflow-gpu package and CUDA toolkit installed. The recommended method for installing all dependencies is using Anaconda.

  If you install the libraries manually, be sure to:
  - Install the CUDA toolkit
  - Install cudnn library
  - Install tensorflow-gpu, it must match the your version of CUDA and cudnn

Declare and define the operation

1. Define the kernel in a separate file, compiled with nvcc
   a. Explicitly state that GPU is to be used
      `#define EIGEN_USE_GPU`
   b. Declare the kernel
      __global__ void MaxKernel(...)
   c. Declare the kernel launcher
      void MaxKernelLauncher(...)
2. Define the operation definition using TensorFlow C++ API, compiled with gcc
   a. Register the nex operation, provide input and output types and shapes
      REGISTER_OP("MaxKernel").Input("input: float32").Output("output: float32").SetShapeFn(<shape_func>);
      <shape_func> is a function that validates the inputs' shape and sets the output shape
   b. Declare a class for your operation
      `class MaxOp : public OpKernel`
   c. Define the Compute method
      `void Compute(OpKernelContext* ctx) override {`
      This is the main op method, you have to allocate the memory for any output and temp tensors (available through ctx->allocate<...> methods) and compute the kernel using the launcher.

Custom operation

```
__global__ void MaxKernel(const float* d_array,  float* d_max,
                                    const int batch_size,
                                    const int seq_len,
                                    const int n_features,
                                    const int k) {

   int index;
   for (int sample = 0; sample < batch_size; sample++){
       for (int feature = 0; feature < n_features; feature++){
           reduceMaxIdxOptimizedWarpShared(
                   d_array + sample * n_features * seq_len + feature * seq_len,
                   seq_len,
                   d_max+sample*n_features + feature,
                    &index,
                   FLT_MAX );
               __syncthreads();
       }
   }
   __syncthreads();
   return;
}
```

This operation computes the maximum along the 3rd axis in the 3D matrix.

Maximum implementation

```
__device__ void reduceMaxIdxOptimizedWarpShared(
               const float* __restrict__ input,
               const int size, float* maxOut,
               int* maxIdxOut,
               float ignore)
{
   __shared__ float sharedMax1;
   __shared__ int sharedMaxIdx1;


   if (0 == threadIdx.x)
   {
       sharedMax1 = 0.0f;
       sharedMaxIdx1  = 0;
   }

   __syncthreads();

   float localMax1 = 0.0f;
   int localMaxIdx1 = 0;
...
```

```
...
   const float warpMax1 = warpReduceMax(localMax1);
   const int warpMaxXY1 =
        warpBroadcast(localMaxIdx1, warpMax1 == localMax1);
   const int lane = threadIdx.x % warpSize;
   if (lane == 0)
   {
       atomicMax(&sharedMax1, warpMax1);
   }
   __syncthreads();
   if (lane == 0)
   {
       if (sharedMax1 == warpMax1)
       {
           sharedMaxIdx1 = warpMaxXY1;
       }
   }
   __syncthreads();
   if (0 == threadIdx.x)
   {
       *maxOut = sharedMax1;
       *maxIdxOut = sharedMaxIdx1;
   }
}
```

https://github.com/apriorit/cuda-reduce-max-with-index/blob/master/src/ReduceMaxIdxOptimizedWarpShared.h

Compile the library

```
//compile.sh

# First, get the necessary compile and link flags from out TF installation. These are some compiler params
And include and lib paths
TF_CFLAGS=( $(python -c 'import tensorflow as tf; print(" ".join(tf.sysconfig.get_compile_flags()))') )
TF_LFLAGS=( $(python -c 'import tensorflow as tf; print(" ".join(tf.sysconfig.get_link_flags()))') )

# Next, compile the kernel
nvcc -std=c++11 -c -o cuda_op_kernel.cu.o
      cuda_op_kernel.cu.cc
      ${TF_CFLAGS[@]} ${TF_LFLAGS[@]} -D GOOGLE_CUDA=1 -x cu -Xcompiler -fPIC

# Then, compile the library
g++ -std=c++11 -shared
      cuda_op_kernel.cc
      cuda_op_kernel.cu.o
      -o cuda_op_kernel.so -fPIC ${TF_CFLAGS[@]} ${TF_LFLAGS[@]} -O2
```

Using the library

```
import tensorflow as tf
import numpy as np

# Our input is an 3D matrix of random values from 0 to 1000
input = np.random.uniform(0, 100, size=(1024, 64, 512),)

# Load the library
mod = tf.load_op_library('./cuda_op_kernel.so')

# Create session
with tf.Session() as sess:
      result = mod.max_kernel(input).eval() # this line performs the computations

# sess.run can be used for better performance
```

Performance

The performance analysis of the operation revealed that is is about 2 times slower than the standard TF implementation.

CPU method is faster because there is not transfer time, however in most deep learning situations chunks of input data are stored on the GPU and rarely transferred.