Convolutions with cuDNN

Oct 1, 2017

12 minute read

Convolutions are one of the most fundamental building blocks of many modern computer vision model architectures, from classification models like VGGNet, to Generative Adversarial Networks like InfoGAN to object detection architectures like Mask R-CNN and many more. While these and other deep learning models have shown to perform exceptionally well in their particular task, they also take notoriously long to train on conventional hardware (CPUs). To fit training within reasonable time frames, you'll probably need access to at least one GPU. Luckily, most popular deep learning libraries have support for GPUs. If that is the case, you'll often find that the core convolution primitives are not implemented by those frameworks themselves, but outsourced to one particular library: *cuDNN*.

CuDNN is a closed-source low-level library for deep learning *primitives* developed by NVIDIA. These *primitives* include operations like convolutions, activation functions like sigmoids or rectified linear units as well as pooling operations. All of the major deep learning frameworks like TensorFlow, Caffe2 or MXNet base many of their GPU kernels on cuDNN. In this post, I'll walk you through the implementation of a basic convolution operation with cuDNN. Given that the library is very low-level, this is quite a lot more work than you'd expect.

Setup

To execute the code I'll discuss in this post, you will need access to an NVIDIA GPU. If you don't have one yet, I recommend grabbing a Titan X if you have a spare wad of cash lying around, or a cute little Jetson TX2 which you can get for $300 if you're a student and for around $600 otherwise. Whatever your device, it will require a compute capability of at least 3.0 to run cuDNN kernels. You'll also need the cuDNN library, which you can download here.

Basic Overview

CuDNN is a CUDA library that abstracts various high performance deep learning kernels, such as convolutions or activations. The basic programming model consists of describing the operands to the kernels, including their shape and memory layout; describing the algorithms we want to perform; allocating memory for cuDNN to operate on (a *workspace*) and finally executing the actual operations. Even though the bulk of the work is hidden behind functions, we'll still need to perform one or the other `cudaMalloc`, `cudaMemcpy` and other low-level CUDA operations ourselves.

Walkthrough

The basics out of the way, let me now walk you through the code for a basic cuDNN convolution operation. We begin by including the necessary header and creating an object of `cudnnHandle_t` type, which will serve as a sort of *context* object, connecting the various operations we need to piece together for our convolution kernel:

```
#include <cudnn.h>

int main(int argc, char const *argv[]) {
  cudnnHandle_t cudnn;
  cudnnCreate(&cudnn);
}
```

One thing to note is that `cudnnCreate`, like *all other* cuDNN routines, returns an error code of `cudnnStatus_t` type. As such, it makes sense to define a macro that checks this status object for any error condition and aborts the execution of our program if something went wrong. We can then simply wrap any library function we call with that macro:

```
#define checkCUDNN(expression)
  {
    cudnnStatus_t status = (expression);
    if (status != CUDNN_STATUS_SUCCESS) {
      std::cerr << "Error on line " << __LINE__ << ": "
                << cudnnGetErrorString(status) << std::e
      std::exit(EXIT_FAILURE);
    }
  }
```

leaving us with

```
checkCUDNN(cudnnCreate(&cudnn));
```

For the remainder of the tutorial, I will assume we're dealing with a single `image` object loaded using OpenCV. You can use this function to load such an object:

```
#include <opencv2/opencv.hpp>

cv::Mat load_image(const char* image_path) {
  cv::Mat image = cv::imread(image_path, CV_LOAD_IMAGE_C
  image.convertTo(image, CV_32FC3);
  cv::normalize(image, image, 0, 1, cv::NORM_MINMAX);
  return image;
}

cv::Mat image = load_image("/path/to/image.png");
```

# Describing Operands

Next, we need to describe the three data structures that participate in the convolution operation: the *input* tensor, the *output* tensor and the *kernel* tensor. CuDNN provides quite a lot of flexibility for to the description of tensors. This is particularly important when it comes to the memory layout, which can be either `NHWC` or `NCHW`. In either of these two formats, `N` specifies the *batch dimension*. CuDNN generally allows its operations to be performed on batches of data (often images), so the first dimension would be for individual images. `H` and `W` stand for the *height* and *width* dimension. Lastly, `C` is the *channels* axis, e.g. for red, green and blue (RGB) channels of a color image. Some deep learning frameworks, like TensorFlow, prefer to store tensors in NHWC format (where channels change most frequently), while others prefer putting channels first. CuDNN provides easy integration for both layouts (this sort of flexibility is one of its core selling points). Let's take a look how to describe the input tensor, which will later on store the image we want to convolve:

```
cudnnTensorDescriptor_t input_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&input_descripto
checkCUDNN(cudnnSetTensor4dDescriptor(input_descriptor,
                                      /*format=*/CUDNN_T
                                      /*dataType=*/CUDNN
                                      /*batch_size=*/1,
                                      /*channels=*/3,
                                      /*image_height=*/i
                                      /*image_width=*/in
```

The first thing we need to do is declare and initialize a `cudnnTensorDescriptor_t`. Then, we use `cudnnSetTensor4dDescriptor` to actually specify the properties of the tensor. For this tensor, we set the format to be `NHWC`. The remainder of the options tell cuDNN that we'll be convolving a single image with three (color) channels, whose pixels are represented as floating point values (between 0 and 1). We also configure the height and width of the tensor. We do the same for the output image:

```
cudnnTensorDescriptor_t output_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&output_descripto
checkCUDNN(cudnnSetTensor4dDescriptor(output_descriptor,
                                      /*format=*/CUDNN_T
                                      /*dataType=*/CUDNN
                                      /*batch_size=*/1,
                                      /*channels=*/3,
                                      /*image_height=*/i
                                      /*image_width=*/im
```

Leaving us with the kernel tensor. CuDNN has specialized construction and initialization routines for kernels (filters):

```
cudnnFilterDescriptor_t kernel_descriptor;
checkCUDNN(cudnnCreateFilterDescriptor(&kernel_descripto
checkCUDNN(cudnnSetFilter4dDescriptor(kernel_descriptor,
                                      /*dataType=*/CUDNN
                                      /*format=*/CUDNN_T
                                      /*out_channels=*/3
                                      /*in_channels=*/3,
                                      /*kernel_height=*/
                                      /*kernel_width=*/3
```

The parameters are essentially the same, since kernels are small images themselves. The only difference is that the `batch_size` is now the number of output channels (`out_channels`) or feature maps. Note, however, that I've switched the format argument to `NCHW`. This will make it easier to define the kernel weights later on.

# Describing the Convolution Kernel

With the parameter description out of the way, we now need to tell cuDNN what kind of (convolution) operation we want to perform. For this, we again declare and configure a *descriptor*, which, you may notice, is an overarching pattern in cuDNN code:

```
cudnnConvolutionDescriptor_t convolution_descriptor;
checkCUDNN(cudnnCreateConvolutionDescriptor(&convolution
checkCUDNN(cudnnSetConvolution2dDescriptor(convolution_d
                                           /*pad_height=
                                           /*pad_width=*
                                           /*vertical_st
                                           /*horizontal_
                                           /*dilation_he
                                           /*dilation_wi
                                           /*mode=*/CUDN
                                           /*computeType
```

If you're unfamiliar with the common hyperparameters (read: knobs) of convolutions, you can find out more about them here. If you're a pro at convolutions, you'll understand that the first two parameters to `cudnnSetConvolution2dDescriptor` after the descriptor control the zero-padding around the image, the subsequent two control the kernel stride and the next two the dilation. The `mode` argument can be either `CUDNN_CONVOLUTION` or `CUDNN_CROSS_CORRELATION`. These are basically the two ways we can compute the weighted sum that makes up a single convolution pass – for our purposes (and convolutions in CNNs as we know them) we want `CUDNN_CROSS_CORRELATION`. The last argument is the data type we're operating on.

Ok, are we finally done? Why is this so much code compared to `tf.nn.conv2d`? Well, the answers are (1) no and (2) because we're many layers beneath the level of abstraction we usually enjoy working with. We need two more things: a more detailed description of the convolution algorithm we want to use and the physical memory to operate on. Let's begin with the first:

```
cudnnConvolutionFwdAlgo_t convolution_algorithm;
checkCUDNN(
    cudnnGetConvolutionForwardAlgorithm(cudnn,
                                        input_descriptor
                                        kernel_descripto
                                        convolution_desc
                                        output_descripto
                                        CUDNN_CONVOLUTIO
                                        /*memoryLimitInB
                                        &convolution_alg
```

Here, we pass the descriptors we previously defined, as well as two further very important arguments: a *preference* for the kind of algorithm we want cuDNN to use for the convolution as well as an upper bound on the amount of memory available for the convolution. The latter we can set to zero, if we don't have a limit. The former can be one of a few choices. Above, I specify `CUDNN_CONVOLUTION_FWD_PREFER_FASTEST`, which tells cuDNN to use the fastest algorithm available. In memory constrained environments, we may instead prefer `CUDNN_CONVOLUTION_FWD_SPECIFY_WORKSPACE_LIMIT` and then specify a non-zero value for the memory limit. After the call to `cudnnGetConvolutionForwardAlgorithm` returns, `convolution_algorithm` will contain the actual algorithm cuDNN has decided to use, for example

- `CUDNN_CONVOLUTION_FWD_ALGO_GEMM`, which models the convolution as an explicit matrix multiplication,
- `CUDNN_CONVOLUTION_FWD_ALGO_FFT`, which uses a Fast Fourier Transform (FFT) for the convolution or
- `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD`, which employs the Winograd algorithm to perform the convolution.

You can have a look at this paper if you're interested in different algorithms and implementations for convolutions.

Next, we ask cuDNN how much memory it needs for its operation:

```
size_t workspace_bytes = 0;
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(cudnn
                                                   input
                                                   kerne
                                                   convo
                                                   outpu
                                                   convo
                                                   &work
std::cerr << "Workspace size: " << (workspace_bytes / 10
          << std::endl;
```

# Allocating Memory

At this point, we need to allocate the required resources for the convolution. The number of buffers and memory requirements for each buffer will differ depending on which algorithm we use for the convolution. In our case, we need four buffers for the workspace, the input and output image as well as the kernel. Let's allocate the first three on the device directly:

```
void* d_workspace{nullptr};
cudaMalloc(&d_workspace, workspace_bytes);

int image_bytes = batch_size * channels * height * width

float* d_input{nullptr};
cudaMalloc(&d_input, image_bytes);
cudaMemcpy(d_input, image.ptr<float>(0), image_bytes, cu

float* d_output{nullptr};
cudaMalloc(&d_output, image_bytes);
cudaMemset(d_output, 0, image_bytes);
```

Note that I got the `batch_size`, `channels`, `height` and `width` variables from the `cudnnGetConvolution2dForwardOutputDim` function, which tells you the dimension of the image after the convolution. I omit it here because it's already a lot of code and the output dimensions are identical to the input anyway. The kernel we'll want to first allocate and populate on the host and then copy to the GPU device:

```
// Mystery kernel
const float kernel_template[3][3] = {
  {1,  1, 1},
  {1, -8, 1},
  {1,  1, 1}
};

float h_kernel[3][3][3][3];
```

```
for (int kernel = 0; kernel < 3; ++kernel) {
  for (int channel = 0; channel < 3; ++channel) {
    for (int row = 0; row < 3; ++row) {
      for (int column = 0; column < 3; ++column) {
        h_kernel[kernel][channel][row][column] = kernel_
      }
    }
  }
}

float* d_kernel{nullptr};
cudaMalloc(&d_kernel, sizeof(h_kernel));
cudaMemcpy(d_kernel, h_kernel, sizeof(h_kernel), cudaMem
```

Note how I first declare a “template” for the 3 by 3 kernel use and then copy it into the three input and three output dimensions of the actual kernel buffer. That is, we’ll have the same pattern three times, once for each channel of the input image (red, green and blue), and that whole kernel three times, once for each output feature map we want to produce.

Can you guess what that kernel does? Read on to find out!

The Convolution (finally)

At last, we can perform the actual convolution operation:

```
const float alpha = 1, beta = 0;
checkCUDNN(cudnnConvolutionForward(cudnn,
                                   &alpha,
                                   input_descriptor,
                                   d_input,
                                   kernel_descriptor,
                                   d_kernel,
                                   convolution_descripto
                                   convolution_algorithm
                                   d_workspace,
                                   workspace_bytes,
                                   &beta,
                                   output_descriptor,
                                   d_output));
```

Here, `alpha` and `beta` are parameters that can be used to mix the input and output buffers (they’re not really useful for us). The rest of the parameters are basically everything we declared and configured up to this point. Note that the `d_workspace` is allowed to be `nullptr` if we pick a convolution algorithm that does not require additional memory.

The only thing left to do at this point is copy the resulting image back to the host and do something with it. We also need to release any resources we allocated, of course:

```
float* h_output = new float[image_bytes];
cudaMemcpy(h_output, d_output, image_bytes, cudaMemcpyDe
// Do something with h_output ...

delete[] h_output;
cudaFree(d_kernel);
cudaFree(d_input);
cudaFree(d_output);
cudaFree(d_workspace);

cudnnDestroyTensorDescriptor(input_descriptor);
cudnnDestroyTensorDescriptor(output_descriptor);
cudnnDestroyFilterDescriptor(kernel_descriptor);
cudnnDestroyConvolutionDescriptor(convolution_descriptor

cudnnDestroy(cudnn);
```

Here is a function you can use to save the image:

```
void save_image(const char* output_filename,
                float* buffer,
                int height,
                int width) {
  cv::Mat output_image(height, width, CV_32FC3, buffer);
  // Make negative values zero.
  cv::threshold(output_image,
                output_image,
                /*threshold=*/0,
                /*maxval=*/0,
                cv::THRESH_TOZERO);
  cv::normalize(output_image, output_image, 0.0, 255.0,
  output_image.convertTo(output_image, CV_8UC3);
  cv::imwrite(output_filename, output_image);
}

// ...

save_image("cudnn-out.png", h_output, height, width);
```

# Running the Code

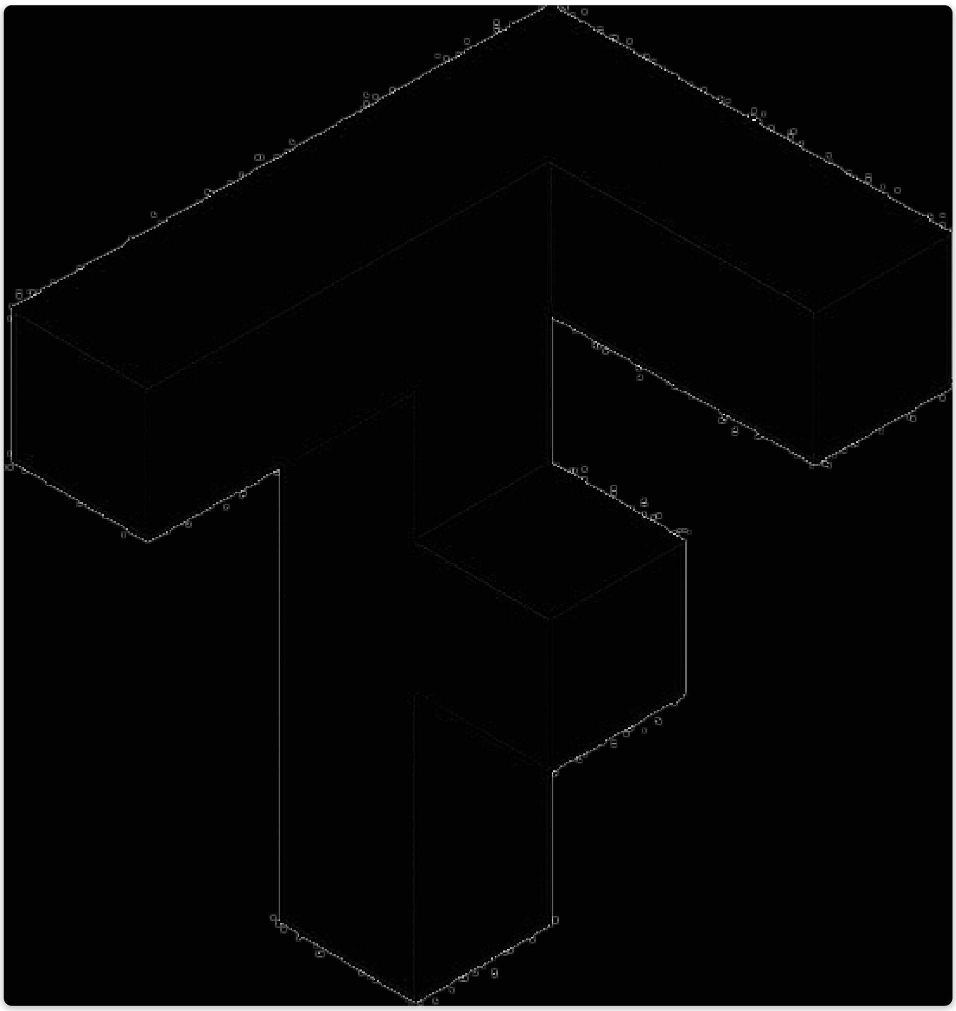
Nearly 200 lines later, we have our basic convolution operation in place. So let's try it out! You can compile this code using a Makefile like this:

```
CXX := nvcc
TARGET := conv
CUDNN_PATH := cudnn
HEADERS := -I $(CUDNN_PATH)/include
LIBS := -L $(CUDNN_PATH)/lib64 -L/usr/local/lib
CXXFLAGS := -arch=sm_35 -std=c++11 -O2

all: conv

conv: $(TARGET).cu
        $(CXX) $(CXXFLAGS) $(HEADERS) $(LIBS) $(TARGET).
        -lcudnn -lopencv_imgcodecs -lopencv_imgproc -lop

.phony: clean
```

```
clean:
	rm $(TARGET) || echo -n ""
```

Where I'm also linking in OpenCV to load and store the image we're convolving. You should set `CUDNN_PATH` to your CUDNN installation directory to `make` successfully. Once built, you could, for example, convolve the TensorFlow logo:

to get ...



a convolved TensorFlow logo. Hooray! As you can see, the kernel I used in this example is a basic edge detector.

Bonus: Sigmoid Activation

I mentioned that besides convolutions, cuDNN also has efficient implementations of activation functions (both forward and backward passes). Here is an example of how you'd add a sigmoid pass to the convolution code so far:

```
// Describe the activation
cudnnActivationDescriptor_t activation_descriptor;
checkCUDNN(cudnnCreateActivationDescriptor(&activation_c
checkCUDNN(cudnnSetActivationDescriptor(activation_descr
                                        /*mode=*/CUDNN_A
                                        /*reluNanOpt=*/C
                                        /*relu_coef=*/0)

// Perform the forward pass of the activation
checkCUDNN(cudnnActivationForward(cudnn,
                                  activation_descriptor,
                                  &alpha,
                                  output_descriptor,
                                  d_output,
                                  &beta,
                                  output_descriptor,
                                  d_output));
```

```
// Release resources
cudnnDestroyActivationDescriptor(activation_descriptor);
```

Done!

Outro

In this post I outlined the minimal steps required to perform a convolution with cuDNN, NVIDIA's high performance library for neural network primitives. The one thing you should notice is how much effort it was to get a basic convolution working. This might indeed seem bad, but remember that cuDNN is not intended to be a high-level, user-facing library. It is intended to be a building block for deep learning frameworks like Caffe2 or TensorFlow, who then add a nice API on top of it, often even in a higher-level language like Python or Lua. Nevertheless, you may be one of the lucky people who gets to hack on deep learning framework backends. In that case, I hope this post was useful to get an idea how to perform a basic convolution with cuDNN. You can find the complete code in this Gist here.

Cheers

Related Posts

Non-Blocking Parallelism for Services in Go

a.k.a. the "tickler" pattern

Why We Should Encourage Cheating On Exams

Questioning what it means to be a cheater

Finding Joy Or Meaning In Your Work

Two parameters worth checking your day job against

Of Hammers And Nails: Solving The Right Problems

The technologist's trap

Making the World Smaller: Facebook, Internships

An email correspondence that made the world smaller

Making the World Smaller: Interviews, Google

An email correspondence that made the world smaller

Making the World Smaller: Internships, Applying and Making it in Big Tech

An email correspondence that made the world smaller

Making the World Smaller: Facebook, Internships, Software Engineering

An email correspondence that made the world smaller

Making the World Smaller: Internships, Getting Noticed, Getting Started in Industry

An email correspondence that made the world smaller

Making the World Smaller: Google, Internships, Going Above and Beyond

An email correspondence that made the world smaller
