Creating Custom Layers in Tensorflow

Matt Angus

University of Waterloo

m2angus@gsd.uwaterloo.ca

July 5, 2017

Introduction

Some Terminology

- Host - code/memory
- Device - code/memory

General Steps

1. Copy data from host memory to device memory
2. Load program onto device, and exicute
3. Copy Results from device memory to host memory

Writing and Calling CUDA Code

```
__global__ void cudaKernel(){
  //do something on the gpu
}

int main() {
  cudaKernel<<<1,1>>>();
}
```

- __ global__ marks device code
- <<<>>> marks a call from host to device

A word on pointers

There are two types of pointers, host pointers and device pointers. Device pointers can be created manipulated with

- `cudaMalloc()`
- `cudaFree()`
- `cudaMemcpy()`

Analagous to C equivalents `malloc()`, `free()`, `memcpy()`.

Passing data between host and device

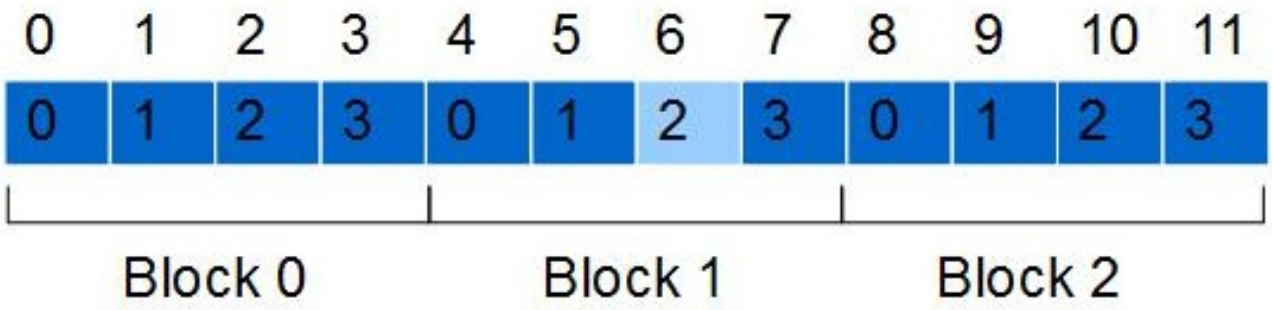
```
__global__ void doWork(int* a){
  //a points to a memory address on the device
}
int main() {
  int a = 10; int *d_a;
  int size = sizeof(int);

  cudaMalloc((void**)&d_a, size);
  cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice)
  doWork<<<1,1>>>(d_a);
  cudaMemcpy(&a, d_a, size, cudaMemcpyDeviceToHost)

  cudaFree(d_a);
  return 0;
}
```

Making it Parallel

- Call `doWork<<<N,M>>>(d_a);`
- N blocks and M threads per block
- Use `blockIdx.x` and `threadIdx.x` to split up work



- `index = threadIdx.x + blockIdx.x * M`
- `index = 2 + 1 * 4 = 6`

Making it Parallel

```
__global__
void doWork(int* a, int* b, int* c, int N);

void launchKernel(int* a, int* b, int* c, int N) {
  int *d_a; int* d_b; int* d_c;
  int size = sizeof(int)*N;

  cudaMalloc((void**)&d_a, size);
  cudaMalloc((void**)&d_b, size);
  cudaMalloc((void**)&d_c, size);
  cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice)
  cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice)
```

Making it Parallel

```
  const int t_per_b = 512; //multiple of 32
  doWork<<<(N + t_per_b-1) / t_per_b,
           t_per_b>>>(d_a, d_b, d_c, N);
  cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost)

  cudaFree(d_a);cudaFree(d_b);cudaFree(d_c);
}
__global__
void doWork(int* a, int* b, int* c, int N){
  int i = threadIdx.x + blockIdx.x * M;
  if(i < N)
    c[i] = a[i] + b[i];
}
```

Some Things to Note

- Does not handle shared access
- `doWork<<<A, B, C, D>>>()`
  - A: # Blocks
  - B: # Threads/Block
  - C: Amount of shared memory
  - D: Device stream

Tensorflow Overview

- Data is stored in tensors (nd-arrays)
- Build a computation graph with operations on tensors
- flow tensors through operations

The Bad News

- Don't know how to do something?
  - start here: `https://fossies.org/dox/tensorflow-1.2.0/`
  - better get digging (through code)

`InferenceContext` member functions

Status	**Merge** (**ShapeHandle** in0, **ShapeHandle** in1, **ShapeHandle** *out) **TF_MUST_USE_RESULT**
Status	**MergePrefix** (**ShapeHandle** s, **ShapeHandle** prefix, **ShapeHandle** *s_out, **ShapeHandle** *prefix_out) **TF_MUST_USE_RESULT**
Status	**Merge** (**DimensionHandle** d0, **DimensionHandle** d1, **DimensionHandle** *out) **TF_MUST_USE_RESULT**
Status	**Subshape** (**ShapeHandle** s, **int64 start**, **ShapeHandle** *out) **TF_MUST_USE_RESULT**
Status	**Subshape** (**ShapeHandle** s, **int64 start**, **int64 end**, **ShapeHandle** *out) **TF_MUST_USE_RESULT**

Overview of Steps

- Register the C++ operation. Define inputs, outputs, and types.
- Implement the kernel for a specific device (GPU, CPU)
- Create a python wrapper
- Implement Gradient (if we have time)

Useful Tensorflow Classes

- `OpKernel`
  - Main op class to inherit from
- `Tensor`
  - Wrapper for data with useful functions
- `OpKernelContext`
  - Context for op when computation begins
  - Contains input, allocates output
- `InferenceContext`
  - Used to infer and validate shape of input and output
  - Used when building the graph
- `Status`
  - communicate to and from TF if everything is okay

Useful Tensorflow Macros

- `TF_RETURN_IF_ERROR(stmnt)`
  - Catches any errors thrown by running `stmnt`
  - returns a bad `Status` with the correct message
- `OP_REQUIRES_OK(cntx,stmnt)`
  - execute `stmnt`
  - ensure that the status of `cntx` is "ok".
- `OP_REQUIRES(cntx,boolean,err)`
  - if `boolean == false` throw `err`
- `CUDA_1D_KERNEL_LOOP(ind,N)`
  - handles indexing into the 1d array
  - calculates ind based on `blockIdx` etc.

Let's Get Coding!

```
git@github.com:mattangus/TF-Custom-Op-Workshop.git
git fetch --all
git checkout skeleton
```

Registering the Op

Define interface between for operation. This indicates to TF what it should expect when building the graph.

```
REGISTER_OP("CustomAdd")
    .Input("a: T")
    .Input("b: T")
    .Output("c: T")
    .Attr("T: {int32, float32, float64}")
    .SetShapeFn(ShapeFn);
```

Registering the Op

Instantiate the kernel with a type.

```
REGISTER_KERNEL_BUILDER(
   Name("CustomAdd")
   .Device(DEVICE_CPU)
   .TypeConstraint<double>("T"),
   CustomAddOp<double>);
```

Shape Inference

```
ShapeHandle a_shape;
TF_RETURN_IF_ERROR(c->WithRank(c->input(0),
    4, &a_shape));

ShapeHandle b_shape;
TF_RETURN_IF_ERROR(c->WithRank(c->input(1),
    4, &b_shape));
```

Shape Inference

```
for(int i = 0; i < 4; i++)
{
  DimensionHandle a_dim = c->Dim(a_shape,i);
  DimensionHandle b_dim = c->Dim(b_shape,i);
  if (c->Value(a_dim) != c->Value(b_dim))
    return errors::InvalidArgument(
  "a and b dim missmatch");
}

c->set_output(0, c->input(0));
```

Allocating Output

```
Tensor* output_tensor = nullptr;
OP_REQUIRES_OK(context,
  context->allocate_output(0,
  a_tensor.shape(),&output_tensor));

auto output = output_tensor->flat<dtype>();
```

Add the Tensors

```
const int N = output.size();
const dtype* a = a_flat.data();
const dtype* b = b_flat.data();
dtype* c = output.data();
for(int i = 0; i < N; i++)
{
  c[i] = a[i] + b[i];
}
```

Changing to the GPU

```
DEVICE_CPU > DEVICE_GPU
```

```
launchAddKernel<dtype>(a_flat.data(),
  b_flat.data(), output.data(), N);
```

```
template void launchAddKernel<double>(
  const double* a, const double* b,
  double* c, int N);
```

Changing to the GPU

```
CUDA_1D_KERNEL_LOOP(index, N)
{
  c[index] = a[index] + b[index];
}
```

```
AddKernel<dtype><<<
  (N + kThreadsPerBlock - 1) / kThreadsPerBlock,
  kThreadsPerBlock>>>(a, b, c, N);
```

Leverage the Template

```
#define REGISTER_KERNEL(type) \
        REGISTER_KERNEL_BUILDER( \
        Name("CustomAdd") \
        .Device(DEVICE_GPU) \
        .TypeConstraint<type>("T"), \
        CustomAddOp<type>) \

REGISTER_KERNEL(int);
REGISTER_KERNEL(float);
REGISTER_KERNEL(double);

#undef REGISTER_KERNEL
```

Leverage the Template

```
#define ADD_KERNEL_TYPE(type) \
template void launchAddKernel<type>( \
  const type* a, const type* b, \
  type* c, int N) \

ADD_KERNEL_TYPE(int);
ADD_KERNEL_TYPE(float);
ADD_KERNEL_TYPE(double);

#undef ADD_KERNEL_TYPE
```

Questions?