Auto Converter

Inhaltsverzeichnis

Original Function Specification

```
// s is const
X func_name(Y y, Z z, ...) { }
```

Function Conversion

Function Header Adjustment Step

```
void func_name(X* x, const Y y, Z z, ...) { }
```

Tensor Function Header

```
template <typename Device>
struct func_nameFunctor {
        void operator()(const Device& d, const Y y_in, const Z z_in,  X* x_out, Z z_out);
};
```

CPU Code

Function Shape Registration

```
// X -> A, Y -> B, Z -> C
Register_OP("func_name")
        .Input("in_y: B")               // y
        .Input("in_z: C")               // z
        .Output("out_x: A")             // x
        .Output("out_z: C")             // z
        SetShapeFn( ::tensorflow::shape_inference::InferenceContext* context) {
                context->set_output(0, ?);                      // x
                context->set_output(1, context->input(1));      // z
                return Status::OK();
        });
```

Function Invokation

```
class func_nameOp : public OpKernel {
public:
        explicit func_nameOp(OpKernelConstruction* context) : OpKernel(context) {}

        void Compute(OpKernelContext* context) override {
                // Input Tensors
                const Tensor& y_in_tensor = context->input(0);
                const B* y_in_data = y_in_tensor.flat<B>().data();

                const Tensor& z_in_tensor = context->input(1);
                const C* z_in_data = z_in_tensor.flat<C>().data();
                // Output Tensors
                Tensor* x_out_tensor = NULL;
                OP_REQUIRES_OK(context, context->allocate_output(0, ?,
                                  &x_out_tensor));
                x_out_data = x_out_tensor.flat<A>().data();

                Tensor* z_out_tensor = NULL;
                OP_REQUIRES_OK(context, context->allocate_output(1, z_in_tensor.shape(),
                                  &z_out_tensor));
                z_out_data = z_out_tensor.flat<C>().data();

                // Input data
                const Y y_in = Y(y_in_data);       // B -> Y
                const Z z_in = Z(z_in_data);       // C -> Z

                // Output data
                X x_out = X(x_out_data);    // A -> X
                Z z_out = Z(z_out_data);    // C -> Z

                func_nameFunctor<Device>()(
                    context->eigen_device<Device>(),
                    y_in, z_in, x_out, z_out);
        }
}
```

Function Registration

```
// Register the CPU Kernel
REGISTER_KERNEL_BUILDER(Name("func_name").Device(DEVICE_CPU),
    func_nameOp<DEVICE_CPU>);

// Register the GPU Kernel
#ifdef GOOGLE_CUDA
REGISTER_KERNEL_BUILDER(Name("func_name").Device(DEVICE_GPU),
    func_nameOp<DEVICE_GPU>);
#endif  // GOOGLE_CUDA
```

Actual CPU Implementation

```
template <>
struct func_nameFunctor<CPUDevice> {
    void operator()(const CPUDevice& d, const Y y_in, const Z z_in,  X* x_out, Z z_out) {  }
};
```

GPU Code

```
// Define the CUDA kernel.
__global__ void func_nameCudaKernel(const Y y_in, const Z z_in,  X* x_out, Z z_out) { }

// Define the GPU implementation that launches the CUDA kernel.
template <>
void func_nameFunctor<GPUDevice> {
    void operator()(const GPUDevice& d, const Y y_in, const Z z_in,  X* x_out, Z z_out) {
        // Launch the cuda kernel.

        int block_count = 1024;
        int thread_per_block = 20;
        ExampleCudaKernel
            <<<block_count, thread_per_block, 0, d.stream()>>>(y_in, z_in, x_out,
                                                                z_out);
    }
};
```

Type Conversion

Mantaflow	**Tensorflow**
MACGrid	float*
FlagGrid	int32*
Grid<Real>	float*
Vec3	float*

Util Function Conversion

- Mantaflow Class: Create constructor for Tensorflow type
- Create read-only class variants
- Add batch dimension

GPU

- Duplicate code
- Substitute __inline for __device__

Others

- Specify function to convert
- Create Build file
- Build Tensorflow