Auto Converter

Inhaltsverzeichnis

Auto Converter....1
- Basic Function Conversion....2
  - Original Function Specification....2
  - Tensor Function Header....2
  - CPU Code....2
    - Function Shape Registration....2
    - Function Invokation....2
    - Function Registration....3
    - Actual CPU Implementation....3
  - GPU Code....4
  - DimSize....4
- Special Cases....4
  - Template Functions....4
- Type Conversion....5
- Tensorflow Custom Op with Template (not used)....5
- Util Function Conversion....5
  - GPU....5
- Others....5
- Limitations....6
- Const....7

Basic Function Conversion

Original Function Specification

```
// Let be Y be of type Grid
void funcName(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, Z z_out);
};
```

CPU Code

Function Shape Registration

```
// Y -> B, Z -> C
Register_OP("FuncName")
      .Input("in_y: B")       // y
      .Input("in_z: C")       // z
      .Output("out_z: C")           // z
      SetShapeFn( ::tensorflow::shape_inference::InferenceContext* context) {
                  context->set_output(0, 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();

            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();

            // Retrieve dimensions
            long batches = y_in_tensor.shape().dim_size(0);
            long depth   = y_in_tensor.shape().dim_size(1);
            long height  = y_in_tensor.shape().dim_size(2);
            long width   = y_in_tensor.shape().dim_size(3);
            DimSize dimSize = DimSize(batches, depth, height, width);

            func_nameFunctor<Device>()(
                  context->eigen_device<Device>(),
                  dimSize,
                  y_in_data, z_in_data, z_out_data);
      }
}
```

Function Registration

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

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

Actual CPU Implementation

```
template <>
struct funcNameFunctor<CPUDevice> {
      void operator()(const CPUDevice& d, Dimsize dimSize, const Y y_in,
                              const Z z_in, Z z_out) {
            for(int i = 0; i < dimSize.lengthOf(standardLength(Z)); i++) {
                z_out[i] = z_in[i];
            }

            FluidSolver fluidSolver = FluidSolver(Vec3i(dimSize.width,
                        dimSize.height, dimSize.depth));

            for(int i_b = 0; i_b < dimSize.batches; i_b++) {
                  B b = convert(y_in);
                  Z z = convert(z_out);

                  funcName(b, c);
            }
      }
};
```

GPU Code

```
// Define the CUDA kernel.
__global__ void funcNameCudaKernel(const Y y_in, const Z z_in, 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, Z z_out) {
            // Launch the cuda kernel.

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

DimSize

The converter tries to fill the values of DimSize as much as possible. The success depends on the parameters of the function. All parameters are expected to be at least an array with a batch dimension. With, height and depth can only be identified if at least one parameter is of type Grid. In the case of a 2D simulation depth will one.

```
// Stores dimensions of the simulation
class DimSize {
public:
    long batches;
    long width;
    long height;
    long depth;   // 1 in case of 2D
}
```

Convert

The conversion from a Tensorflow type to the required Mantaflow type strongly depends on the original type. (tbc.)

Special Cases

Template Functions

```
TKernel() template <class T>
void funcName(Grid<T> grid, T t) { }
```

A template function gets prepreprocessed and is split into multiple operators that are no longer dependend on a generic type.

```
=> T = {bool, int, float, Vec3}
=> Manta Func Call:= funcName<T>(...)
=> Tensor Func Call:= func_name_T(...)
```

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 (currently via TKernel; subject to change)
- Create Build file
- Build Tensorflow custom ops
-

Limitations

- Functions with return value not supported: generic shape inference impossible
- Functions with more than one none const parameter not supported: Tensorflow custom op allows only one return value
- Only supports float precision caluclation. (Mantaflows Real type is always interpreted as float)
- Generic types have limited support. Concrete templete implementation can not be interpreted by the current Mantaflow parser. Additionally, nested templates are not correctly interpreted.

Appendix A: Type Conversion

	Mantaflow	**Tensorflow**
Basic Type		
	Real	float
	Float	float
	Int	int32
	Bool	bool
Vec3		
	Vec3	float*
Generic		
	T	Basic Type / float*
Grid		
	MACGrid	float*
	FlagGrid	int32*
Generic Grid		
	Grid<T>	Basic Type*

Tensorflow Custom Op with Template (not used)

- Additional Op Register: `.Attr("T: {float, float*}")`
- Additional Kernel Register:
  - ```
    REGISTER_KERNEL_BUILDER(
       Name("FuncName")
       .Device(DEVICE_CPU)
       .TypeConstraint<float*>("T"),
       FuncNameOp<float*>);
    ```
  - ```
    REGISTER_KERNEL_BUILDER(
       Name("FuncName")
       .Device(DEVICE_CPU)
       .TypeConstraint<float>("T"),
       FuncNameOp<float>);
    ```

Const

I commented that __ldg part in my kernel and executed by normal execution, and vice versa. In both cases it gives me correct multiplication result. I am confused with the time difference I am getting between these executions, because its huge almost more than 100X!
In case of __ldg it gives me: Time taken is 0.014432 (ms)
And in case of normal execution without __ldg it gives me: Time taken is 36.858398 (ms)
From the CUDA C Programming Guide

> Global memory accesses for devices of compute capability 3.x are cached in L2 and for devices of compute capability 3.5, may also be cached in the read-only data cache described in the previous section; they are not cached in L1.

...

> Data that is read-only for the entire lifetime of the kernel can also be cached in the read-only data cache described in the previous section by reading it using the __ldg() function (see Read-Only Data Cache Load Function). When the compiler detects that the read-only condition is satisfied for some data, it will use __ldg() to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the const and __restrict__ qualifiers increases the likelihood that the compiler will detect the read-only condition.

The read only cache accesses have a much lower latency than the global memory accesses. Because matrix multiplication accesses the same values from memory many times, caching in the read only cache gives a huge speedup (in memory bound applications).