Google Summer of Code

Clad::Reverse mode automatic differentiation of GPU kernels

HEP Software Foundation

Group: The Compiler Research Group
Mentors: Vassil Vassilev, Parth Arora, Alexander Penev
Student: Christina Koutsou

Introduction

- **What is Automatic Differentiation?**

Automatic Differentiation (AD) is a technique used by computers to compute the gradient (or derivative) of a function by breaking it down into elementary steps or operations.

- **What is Clad?**

Clad is a clang plugin for automatic differentiation that performs source-to-source transformation and produces a function capable of computing the derivatives of a given function at compile time.

Current status

Support of host and device functions

```
__device__ double fn(double *a) {
   return *a * *a;
}

__global__ void compute(double* d_a, double* d_result) {
  auto fn_grad = clad::gradient(fn, "a");
  fn_grad.execute(d_x, d_result);
}

int main(void) {
  (...) // memory allocations and initializations
  compute<<<1, 1>>>(d_a, d_result);
  cudaDeviceSynchronize();
  // copy back result to CPU
  cudaMemcpy(result.data(), d_result, N * sizeof(double),
                                    cudaMemcpyDeviceToHost);
  return 0;
}
```

Goal

Support of global functions (kernels) as well

```
__global__ void fn(double *a) {
   *a *= *a;
}

int main(void) {
  (...) // memory allocations and initializations
  auto fn_grad = clad::gradient(fn, "a");
  fn_grad.execute(d_x, d_result);
  cudaDeviceSynchronize();
  // copy back result to CPU
  cudaMemcpy(result.data(), d_result, N * sizeof(double),
                                    cudaMemcpyDeviceToHost);
  return 0;
}
```

Issues categories

- General support: The ability to produce and execute a kernel’s derivative function without it necessarily being correct
  - Handle kernel calls
- Support of kernel’s nature: Derive the kernel with respect to its characteristics
  - Concerns the kernel’s body

General Support: Kernel’s derivative compilation

General Support: Kernel’s derivative execution

Kernel calls require a grid configuration

We need to create a grid configuration when creating the call to the derived function

- Detect if the function to derive is a kernel
  - If true, create a configuration expression of (grid size, block size, shared memory size, stream ID) = (1,1,0,0) and assign it to configExpr
  - Else, assign cofigExpr to nullptr
- Pass the configuration expression as an argument to ActOnCallExpr()

We need to create a grid configuration when calling the derived function for execution

- Overload of execute() function:
  execute(...args, grid_size, block_size, shared_mem_size, stream)
- Same with execute_helper() and execute_with_default_args()
- Call the function as:

```
return f<<<grid_size, block_size, shared_mem_size,
          stream>>>(static_cast<Args>(args)...,
                    static_cast<Rest>(nullptr)...);
```

Support of kernel's characteristics: Specify output argument to derive

Kernels are void functions and, as a result, we cannot derive based on a return statement

⇩

- Overload of gradient() function for void functions → add an extra argument for a user-specified output
- Pass argument to the derivation request
- Create its derivative variable
- Add it and its derivative to the variable list whose expressions are derived

```
void foo(int *in, int *out){
  *out = 2 * *in;
}

// reverse pass:
// da = dout
// din = 2 * da ( or else din = 2
* dout)

void foo_grad(int *in, int *out,
clad::array_ref<int> _d_in) {
     * _d_out = 1;
      int _t0;
      _t0 = *out;
      *out = 2 * *in;
      {
         *out = _t0;
         int _r_d0 = * _d_out;
         * _d_out -= _r_d0;
         * _d_in += 2 * _r_d0;
      }
}
```

Support of kernel's characteristics: Account for write race conditions in computation of the derivative value

- `* _d_in += 2 * _r_d0; -> * _d_in = 2 * _r_d0;`
- When arrays are concerned:
  - Use array subscripts for all derived variables

```
__global__ void compute(double *in, double *out,
double val)
{
  int index = threadIdx.x;
  out[index] = in[index] + val;
}
```

$$\Downarrow$$

```
double _r_d0 = _d_out[index0];
* _d_val = _r_d0; -> * _d_val[index0] = _r_d0;
```

Support of kernel's characteristics: Handling of CUDA built-in objects

CUDA Built-ins that will be supported:

- threadIdx
- blockIdx
- blockDim
- gridDim
- __shared__ macro

Benefits

The rise of AI has shed light into the power of GPUs

The notion of General Purpose GPU Programming is becoming more and more popular

Implementation of mathematics and operations needed for scientific computing are getting adjusted to GPU's architecture

This project widens Clad's use range and audience and helps it become an even more powerful tool

Thank you for your attention!

Any Questions?