Enabling reverse-mode automatic differentiation of CUDA kernels

The end… For now

Previously on “CUDA kernel derivation”

- Enable the use of clad::gradient for CUDA kernels
- Execute the derived kernel with the grid configuration specified by the user
- Support the use of CUDA built-in variables in the original and the derived kernel:
  - threadIdx, blockIdx, blockDim, gridDim, warpSize
- Support the use of the shared memory keyword in the original and the derived kernel
  - Implemented but blocked
- Specify the input and output parameters to differentiate the function accordingly
- Eliminate write-race conditions
  - Simple solution implemented

clad::gradient on CUDA kernels

```
__global__ void kernel(int *a) {
  *a *= *a;
}
// clad::gradient(kernel);
void kernel_grad(int *a, void *_temp__d_a0) __attribute__((global)) {
```

→ Function pointer returned to user

```
        int *_d_a = (int *)_temp__d_a0;
        kernel_grad<<<~~1, 1~~>>>(a, _d_a);
}             kernel_grad(a, _d_a);

                                                 device
void kernel_grad(int *a, int *_d_a) __attribute__((~~global~~)) {
        int _t0 = *a;
        *a *= *a;
        {
        *a = _t0;
        int _r_d0 = *_d_a;
        *_d_a = 0;
        *_d_a += _r_d0 * *a;
        *_d_a += *a * _r_d0;
        }
}
```

Previously on “CUDA kernel derivation”

- Enable the use of clad::gradient for CUDA kernels ✅
- Execute the derived kernel with the grid configuration specified by the user
- Support the use of CUDA built-in variables in the original and the derived kernel:
  - threadIdx, blockIdx, blockDim, gridDim, warpSize
- Support the use of the shared memory keyword in the original and the derived kernel
  - Implemented but blocked
- Specify the input and output parameters to differentiate the function accordingly
- Eliminate write-race conditions
  - Simple solution implemented

execute CUDA kernels

The user must use `execute_kernel` instead of `execute` and `execute_kernel` can only be used with CUDA kernels:

```
auto error = clad::gradient(fake_kernel);
error.execute_kernel(dim3(1), dim3(1), a, d_a); // CHECK-EXEC: Use execute() for non-global CUDA kernels

auto test = clad::gradient(kernel);
test.execute(a, d_a); // CHECK-EXEC: Use execute_kernel() for global CUDA kernels
```

The user must provide a grid configuration and they also have the ability to specify the amount of shared memory to utilize and the stream to execute the kernel on:

```
Option 1:
  auto test = clad::gradient(F);
  test.execute_kernel(grid, block, x, dx); // shared memory = 0 & stream = nullptr
Option 2:
  auto test = clad::gradient(F);
  cudaStream_t stream;
  cudaStreamCreate(&stream);
  test.execute_kernel(grid, block, shared_mem, stream, x, dx);
```

Previously on “CUDA kernel derivation”

- Enable the use of clad::gradient for CUDA kernels ✅
- Execute the derived kernel with the grid configuration specified by the user ✅
- Support the use of CUDA built-in variables in the original and the derived kernel: ✅
  - threadIdx, blockIdx, blockDim, gridDim, warpSize
- Support the use of the shared memory keyword in the original and the derived kernel
  - Implemented but blocked
- Specify the input and output parameters to differentiate the function accordingly
- Eliminate write-race conditions
  - Simple solution implemented

CUDA builtins and Output and Input args of the CUDA kernel

CUDA kernels are void functions, so the output is included in the argument list. Thus, to compute the reverse-mode derivative of the kernel we cannot rely on a return statement.

❖ Solution: Include the output parameter in the argument list of the clad::gradient call:

```
__global__ void add_kernel_4(int *out, int *in, int N) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < N) {
        int sum = 0;
        // Each thread sums elements in steps of warpSize
        for (int i = index; i < N; i += warpSize) {
        sum += in[i];
        }
        out[index] = sum;
  }
}

clad::gradient(add_kernel_4, “in, out”);
```

Previously on “CUDA kernel derivation”

- Enable the use of clad::gradient for CUDA kernels ✅
- Execute the derived kernel with the grid configuration specified by the user ✅
- Support the use of CUDA built-in variables in the original and the derived kernel: ✅
    - threadIdx, blockIdx, blockDim, gridDim, warpSize
- Support the use of the shared memory keyword in the original and the derived kernel
    - Implemented but blocked
- Specify the input and output parameters to differentiate the function accordingly ✅
- Eliminate write-race conditions
    - Simple solution implemented

Handle write-race conditions

When two or more threads read from the same memory address, when computing the reverse mode of the kernel these threads attempt to write to the same memory address. To ensure that no write-race condition occurs, the operations on this memory address are made atomic:

```
__global__ void add_kernel_6(double *a, double *b) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  a[index] = b[0];
}

void add_kernel_6_grad(double *a, double *b, double *_d_a, double *_d_b) {
         ...
         {
         a[index0] = _t2;
         double _r_d0 = _d_a[index0];
         _d_a[index0] = 0.;
         _d_b[0] += _r_d0;  ──────────→  atomicAdd(&_d_b[0], _r_d0);
         }
}
```

Today’s topics

- Handle device pullbacks
- Correctly derive functions that include CUDA host functions and kernel launches

Device pullbacks

```
__device__ double device_fn(double *in, double val) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  return in[index] + val;
}

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

clad::gradient(kernel, “out, val”);
```

```
__attribute__((device)) void device_fn_pullback(double *in,
double val, double _d_y, double *_d_val) {
        unsigned int _t1 = blockIdx.x;
        unsigned int _t0 = blockDim.x;
        int _d_index = 0;
        int index0 = threadIdx.x + _t1 * _t0;
        *_d_val += _d_y;
}
```

should we use atomic add?

```
void kernel_with_device_call_2_grad_0_2(double *out, double
*in, double val, double *_d_out, double *_d_val) {
        int _d_index = 0;
        int index0 = threadIdx.x;
        double _t0 = out[index0];
        out[index0] = device_fn_2(in, val);
        {
        out[index0] = _t0;
        double _r_d0 = _d_out[index0];
        _d_out[index0] = 0.;
        double _r0 = 0.;
        device_fn_2_pullback(in, val, _r_d0, &_r0);
        atomicAdd(_d_val, _r0);
        }
}
```

local for each thread

No need for any change!

Device pullbacks

```
__device__ double device_fn(double *in, double *val) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  return in[index] + *val;
}

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

clad::gradient(kernel, “out, val”);
```

```
__attribute__((device)) void device_fn_pullback(double *in,
double *val, double _d_y, double *_d_val) {
        unsigned int _t1 = blockIdx.x;
        unsigned int _t0 = blockDim.x;
        int _d_index = 0;
        int index0 = threadIdx.x + _t1 * _t0;
        *_d_val += _d_y;
}
```

should we use atomic add?

```
void kernel_with_device_call_2_grad_0_2(double *out, double
*in, double *val, double *_d_out, double *_d_val) {
        int _d_index = 0;
        int index0 = threadIdx.x;
        double _t0 = out[index0];
        out[index0] = device_fn_2(in, val);
        {
        out[index0] = _t0;
        double _r_d0 = _d_out[index0];
        _d_out[index0] = 0.;
        device_fn_2_pullback(in, val, _r_d0, _d_val);
        atomicAdd(_d_val, _r0);
        }
}
```

Atomic ops should be used

Solutions

Can we use atomic operations for all adjoints inside device functions as well?

- NO! Atomic operations can only be used on memory addresses that reside in the global GPU memory

Ok, so we have to distinguish the global variables from the rest.

Facts:

- All the adjoint arguments of the derivative kernel are global
- Device functions are only called inside kernels, and, hence, so are device pullbacks

↓

Keep a record of global args and pass them to the device pullback differentiation request.
This works recursively for nested device functions as well.

Device pullbacks

```
__device__ double device_fn(double *in, double *val) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  return in[index] + *val;
}

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

clad::gradient(kernel, “out, in”);
clad::gradient(kernel, “out, in, val”);
```

1.

```
__attribute__((device)) void device_fn_pullback(double *in,
double *val, double _d_y, double *_d_in, double *_d_val) {
         unsigned int _t1 = blockIdx.x;
         unsigned int _t0 = blockDim.x;
         int _d_index = 0;
         int index0 = threadIdx.x + _t1 * _t0;
         {
                 atomicAdd(&_d_in[index0],_d_y);
                 *_d_val += _d_y;                    ✅
         }
}
```

Redefinition: the second pullback function wouldn’t be created

2.

```
__attribute__((device)) void device_fn_pullback(double *in,
double *val, double _d_y, double *_d_in, double *_d_val) {
         unsigned int _t1 = blockIdx.x;
         unsigned int _t0 = blockDim.x;
         int _d_index = 0;
         int index0 = threadIdx.x + _t1 * _t0;
         {
                 atomicAdd(&_d_in[index0], _d_y);
                 atomicAdd(_d_val, _d_y);            ✅
         }

}
```

Solution: append globals call args to the device pullback name

```
1)    __attribute__((device)) void device_fn_pullback_0_1_3(double *in,
                                  double *val, double _d_y, double *_d_val)
2)    __attribute__((device)) void device_fn_pullback_0_1_3_4(double *in,
                                  double *val, double _d_y, double *_d_val)
```

Kernel pullbacks

```
__global__ void kernel(int *out, int *in, int N) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < N) {
    out[index] = in[index];
  }
}

void launch(int *out, int *in, int N) {
  kernel<<<1, 5>>>(out, in, N);
}

clad::gradient(launch, “out, in”);
```

```
void launch_kernel_grad_0_1(int *out, int *in, const int N, int
*_d_out, int *_d_in) {
        int _d_N = 0;
        kernel<<<1, 5>>>(out, in, N);
        {
                int _r0 = 0;
                kernel_pullback<<<1, 5>>>(out, in, N, _d_out,
                                          _d_in, &_r0);
                _d_N += _r0;
        }
}
```

CPU variable

```
void launch_kernel_grad_0_1(int *out, int *in, const int N, int *_d_out, int *_d_in) {
        int _d_N = 0;
        kernel<<<1, 5>>>(out, in, N);
        {
                int _r0 = 0;
                int *_r1 = nullptr;
                cudaMalloc(&_r1, 4);
                cudaMemset(_r1, 0, 4);
                kernel_pullback<<<1, 5>>>(out, in, N, _d_out,
                                          _d_in, _r1);
                cudaMemcpy(&_r0, _r1, 4, cudaMemcpyDeviceToHost);
                cudaFree(_r1);
                _d_N += _r0;
        }
}
```

CUDA Malloc and Free functions

```
double fn(double *out, double *in) {
  double *in_dev = nullptr;
  cudaMalloc(&in_dev, 10 * sizeof(double));
  cudaMemcpy(in_dev, in, 10 * sizeof(double),
cudaMemcpyHostToDevice);
  kernel_call<<<1, 10>>>(out, in_dev);
  double *out_host = (double *)malloc(10 * sizeof(double));
  cudaMemcpy(out_host, out, 10 * sizeof(double),
cudaMemcpyDeviceToHost);
  double res = 0;
  for (int i=0; i < 10; ++i) {
        res += out_host[i];
  }
  free(out_host);
  cudaFree(out);
  cudaFree(in_dev);
  return res;
}

clad::gradient(fn, "out, in");
```

```
void fn_memory_grad(double *out, double *in, double *_d_out,
double *_d_in) {
        int _d_i = 0;
        int i = 0;
        clad::tape<double> _t1 = {};
        double *_d_in_dev = nullptr;
        double *in_dev = nullptr;
        cudaMalloc(&_d_in_dev, 10 * sizeof(double));
        cudaMemset(_d_in_dev, 0, 10 * sizeof(double));
        cudaMalloc(&in_dev, 10 * sizeof(double));
        cudaMemcpy(in_dev, in, 10 * sizeof(double),
                                cudaMemcpyHostToDevice);
        kernel_call<<<1, 10>>>(out, in_dev);
        . . .
        free(out_host);
        free(_d_out_host);
        cudaFree(out);
        ~~cudaFree(_d_out);~~
        cudaFree(in_dev);
        cudaFree(_d_in_dev);
}
```

CUDA Memcpy function

```
double fn(double *out, double *in) {
  double *in_dev = nullptr;
  cudaMalloc(&in_dev, 10 * sizeof(double));
  cudaMemcpy(in_dev, in, 10 * sizeof(double),
cudaMemcpyHostToDevice);
  kernel_call<<<1, 10>>>(out, in_dev);
  double *out_host = (double *)malloc(10 * sizeof(double));
  cudaMemcpy(out_host, out, 10 * sizeof(double),
cudaMemcpyDeviceToHost);
  double res = 0;
  for (int i=0; i < 10; ++i) {
        res += out_host[i];
  }
  free(out_host);
  cudaFree(out);
  cudaFree(in_dev);
  return res;
}

clad::gradient(fn, “out, in”);
```

```
void fn_memory_grad(double *out, double *in, double *_d_out,
double *_d_in) {
        . . .
        cudaMemcpy(in_dev, in, 10 * sizeof(double),
                                  cudaMemcpyHostToDevice);
        kernel_call<<<1, 10>>>(out, in_dev);
        . . .
        kernel_call_pullback<<<1, 10>>>(out, in_dev, _d_out,
                                           _d_in_dev);
        {
                unsigned long _r0 = 0UL;
                cudaMemcpyKind _r1 =
                        static_cast<cudaMemcpyKind>(0U);
                clad::custom_derivatives::cudaMemcpy_pullback(
                in_dev, in, 10 * sizeof(double),
                cudaMemcpyHostToDevice, _d_in_dev, _d_in,
                &_r0, &_r1);
        }
        . . .
}
```

CUDA Memcpy function

```
void fn_memory_grad(double *out, double *in, double *d_out,
double *_d_in) {
        . . .
        cudaMemcpy(in_dev, in, 10 * sizeof(double),
                                cudaMemcpyHostToDevice);
        kernel_call<<<1, 10>>>(out, in_dev);
        . . .
        kernel_call_pullback<<<1, 10>>>(out, in_dev, _d_out,
                                          _d_in_dev);
        {
                unsigned long _r0 = 0UL;
                cudaMemcpyKind _r1 =
                        static_cast<cudaMemcpyKind>(0U);
                clad::custom_derivatives::cudaMemcpy_pullback(
                in_dev, in, 10 * sizeof(double),
                cudaMemcpyHostToDevice, _d_in_dev, _d_in,
                &_r0, &_r1);
        }
        . . .
}
```

- Reverse cudaMemcpyKind:
  - From cudaMemcpyHostToDevice to cudaMemcpyDeviceToHost
  - From cudaMemcpyDeviceToHost to cudaMemcpyHostToDevice
- Reverse source and destination addresses
  - Use auxiliary destination address for the cudaMemcpy operation that will be used to perform the plus-assign operation of the adjoint later
    - Destination address is in GPU: Use custom kernel to perform the atomicAdd ops
    - Destination address is in CPU: Use for loop to add the values to each position of the destination address

Demo

```
void launchTensorContraction3D(float* C, const float* A, const float* B, const size_type D1, const size_type D2, const size_type D3,
const size_type D4, const size_type D5) {
        float *d_A = nullptr, *d_B = nullptr, *d_C = nullptr;
        // Allocate device memory and copy data from host to device
        // initialize other data
        . . .

        // Launch the kernel
        tensorContraction3D<<<1, 256>>>(d_C, d_A, d_B, d_A_dim, d_B_dim, /*contractDimA=*/2, /*contractDimB=*/0);

        // Copy the result from device to host
        cudaMemcpy(C, d_C, C_size, cudaMemcpyDeviceToHost);

        // Free device memory
        . . .
}

auto tensor_grad = clad::gradient(launchTensorContraction3D, "C, A, B");
tensor_grad.execute(&C[0][0][0][0], &A[0][0][0], &B[0][0][0], D1, D2, D3, D4, D5, &gradC[0][0][0][0], &gradA[0][0][0], &gradB[0][0][0]);
```

Future work

- Enable support of shared memory
- Handle synchronization functions, like __syncthreads() and cudaDeviceSynchronize()
- Extend support of CUDA math and host functions
- Benchmark applications
- Optimize the number of local variables created using an analysis

Acknowledgments

Thank you for your attention