



Enabling reverse-mode automatic differentiation of CUDA kernels

Records of painful experiences

Objectives till now:

- 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 builtin variables in the original and the derived kernel
- Support the use of the shared memory keyword in the original and the derived kernel
- Specify the input and output parameters to differentiate the function accordingly
- Eliminate write-race conditions

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)) {
```

```
    int *_d_a = (int *)_temp__d_a0;
```

```
    kernel_grad<<<1,1>>>(a, _d_a);
```

```
}
```

- Configuration at compile time: need to call gradient as many times as the desired grid configurations
- Call of a global function inside another global one is prohibited

```
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;
```

```
    }
```

```
}
```

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);  
  
void kernel_grad(int *a, int *_d_a) __attribute__((global)) {  device  
    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;  
    }  
}
```

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);
```

CUDA builtin variables supported

The original kernel may include any CUDA builtin variable from the following list:

- threadIdx, blockIdx, blockDim, gridDim, warpSize

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

void add_kernel_grad(int *out, int *in, int *_d_out, int *_d_in) {
    unsigned int _t1 = blockIdx.x;
    unsigned int _t0 = blockDim.x;
    int _d_index = 0;
    int index0 = threadIdx.x + _t1 * _t0;
    int _t2 = out[index0];
    out[index0] += in[index0];
    {
        out[index0] = _t2;
        int _r_d0 = _d_out[index0];
        _d_in[index0] += _r_d0;
    }
}
```

__shared__ keyword in CUDA kernels

Shared memory can only be declared inside a global kernel

→ The produced kernel containing the derived body of the original kernel is now a device function instead of a global one

Talks about altering the signature of the derivative kernel to eliminate the need of the wrapper function. Maybe something like this:

```
// clad::gradient(add_kernel, "in, out");
```

```
void add_kernel_grad_wrapper(int *out, int *in, int N, void *_d_out, void *_d_in, void *_d_N)  
    void add_kernel_grad(int *out, int *in, int N, int *_d_out, int *_d_in)
```

```
void add_kernel_grad(int *out, int *in, int N, int *_d_out, int *_d_in, int *_d_N0)
```

→ dummy name to avoid redeclaration error inside function body

→ can be nullptr

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;  
    }  
}
```

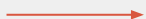
```
clad::gradient(add_kernel_4, "in, out");
```

```
void add_kernel_4_grad_0_1(int *out, int *in, int N, int  
*_d_out, int *_d_in) {  
    int _d_N = 0;  
    bool _cond0;  
    int _t2;  
    unsigned int _t1 = blockIdx.x;  
    unsigned int _t0 = blockDim.x;  
    int _d_index = 0;  
    int index0 = threadIdx.x + _t1 * _t0;  
    {  
        _cond0 = index0 < N;  
        ...  
    }  
    if (_cond0) {  
        {  
            out[index0] = _t2;  
            int _r_d0 = *_d_out[index0];  
            *_d_out[index0] = 0;  
            *_d_in[index0] += _r_d0;  
        }  
    }  
}
```






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);
    }
}
```

Objectives till now:

- 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 builtin variables in the original and the derived kernel 
- 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

Future work

- Explore more options to efficiently handle the write-race conditions
- Ensure the correct kernel derivation as a pullback function
- Add demos
- Extend support of CUDA builtins and functions
- Optimize the number of local variables created



Thank you for your attention

Any questions?