

# Bringing Automatic Differentiation to CUDA with Compiler-Based Source Transformations

Why, what and how?

Christina Koutsou, Dr. Vassil Vassilev, Dr. David Lange  
*Compiler Research Group, Princeton University*

# Background

- 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 Reverse-mode AD?

Reverse-mode AD automatically computes the derivative of a function, but **performs the operations in reverse order** while also **swapping the left and right hand-side expressions** of these operations. This type of AD is used when we're interested in the derivative with respect to many inputs of the function.

- 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**.

# Background

- **What is CUDA?**

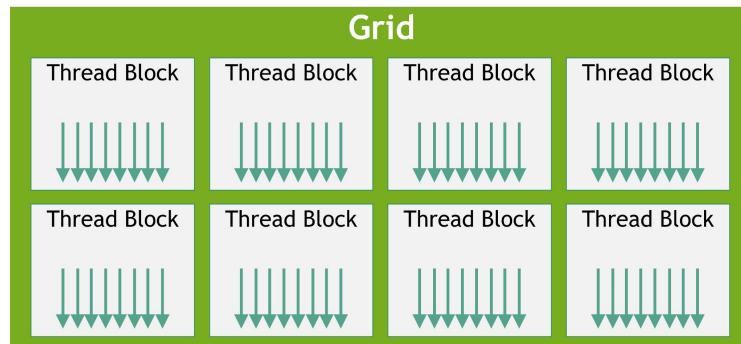
CUDA is an API used to program NVIDIA GPUs.

- **What is a CUDA kernel?**

A CUDA kernel is a global void function run by CUDA on a GPU. They are launched by the host (CPU) with a certain grid configuration.

- **How can non-global device functions be accessed?**

Device (GPU) functions can only be called inside kernels. They cannot be launched similarly to kernels in order to create a new grid configuration for them, rather, each thread running the kernel will execute the device function as many times as it's called.



# Motivation

- **Why do we need gradients?**
  - Physics equations and simulations
  - Optimization problems (i.e. Finance or backpropagation in ML)
- **Why do we need Automatic Differentiation?**
  - Differentiate code that is more complex than just the basic math functions *automatically*
- **Why do we need gradients in GPUs?**
  - Faster execution
  - Cheaper and more energy efficient
  - Not tailored to specific applications
  - Compatible with C/C++

**Great for offloading  
computation and enabling  
hybrid CPU/GPU applications**

# Case study - Tensor contraction

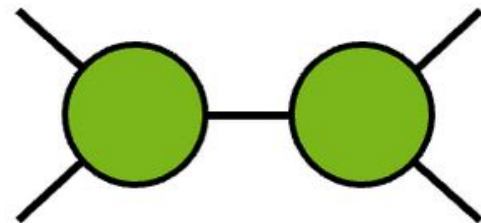
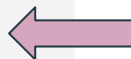
```
__global__ void tensorContraction3D(float* C, const float* A, const float* B, const
size_type* A_dim, const size_type* B_dim, const int contractDimA, const int
contractDimB) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // Each thread computes one element of the output tensor
    int totalElements =
        A_dim[(contractDimA + 1) % 3] * A_dim[(contractDimA + 2) % 3] *
        B_dim[(contractDimB + 1) % 3] * B_dim[(contractDimB + 2) % 3];
    if (idx < totalElements) {
        size_type A_start, B_start, A_step, B_step;
        size_type A_a, A_b, A_c, B_d, B_e, B_f;
        computeStartStep(A_start, A_step, B_start, B_step, idx, A_dim, B_dim,
contractDimA, contractDimB);

        float sum = 0.0f;
        // A_dim[contractDimA] == B_dim[contractDimB]
        for (int i = 0; i < A_dim[contractDimA]; i++)
            sum += A[A_start + (i * A_step)] * B[B_start + (i * B_step)];

        C[idx] = sum;
    }
}
```

Tensor derivatives are used for backpropagation in ML

$$C_{ijlm} = \sum_{k=1}^K A_{ijk} \cdot B_{klm}$$



contraction between  
two rank-3 tensors

# Case study - Black-Scholes model

$$C(S, t) = N(d_1)S - N(d_2)Ke^{-rT}$$

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

$C(S, t)$	(call option price)
$N()$	(cumulative distribution function)
$T = (T_1 - t)$	(time left til maturity (in years))
$S$	(stock price)
$K$	(strike price)
$r$	(risk free rate)
$\sigma$	(volatility)

"The Greeks" measure the **sensitivity** of the value of a derivative product or a financial portfolio **to changes in parameter values**. They are **partial derivatives** of the price with respect to the parameter values.

```
__device__ inline void BlackScholesBodyGPU(float& CallResult, float&
PutResult, /*Stock Price*/ float S, /*Option strike*/ float X,
/*Option years*/float T, /*Riskless rate*/ float R, /*Volatility
rate*/float V) {
    float sqrtT, expRT;
    float d1, d2, CNDD1, CNDD2;

    sqrtT = __fdividef(1.0f, rsqrtf(T));
    d1 = __fdividef(__logf(S / X) + (R + 0.5f * V * V) * T, V * sqrtT);
    d2 = d1 - V * sqrtT;

    CNDD1 = cndGPU(d1);
    CNDD2 = cndGPU(d2);

    // Calculate Call and Put simultaneously
    expRT = __expf(-R * T);
    CallResult = S * CNDD1 - X * expRT * CNDD2;
    PutResult = X * expRT * (1.0f - CNDD2) - S * (1.0f - CNDD1);
}
```

# Challenges in implementation

GPU architecture and its interaction with the CPU introduce new problems that need to be addressed:

- Memory hierarchy of GPU (global, shared and local memories)
- Parallel execution of computations leading to race conditions
- Communication between CPU and GPU only achieved through CUDA's API

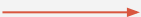
Let's dive a **bit** deeper . . .



# 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(int *out, int val) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    out[index] += val;
}

// auto grad_object = clad::gradient(add_kernel, "out, val");
void add_kernel_grad(int *out, int val, int *_d_out, int *_d_val) {
    ...
    int _r_d0 = _d_out[index0];
    _d_val += _r_d0;            atomicAdd(&_d_val, _r_d0);
}
```

# Where should we use atomic operations?

- Does a write-race condition occur in the assignment of every parameter that is a pointer to a memory address?
  - No! A write race condition can only occur on a memory address where two or more threads have access to. In our cases so far, this translates into **pointers to a global memory address**
- How do we know if a parameter resides in global memory?
  - Global memory is allocated only through the use of **cudaMalloc()**, which is **called by the CPU**

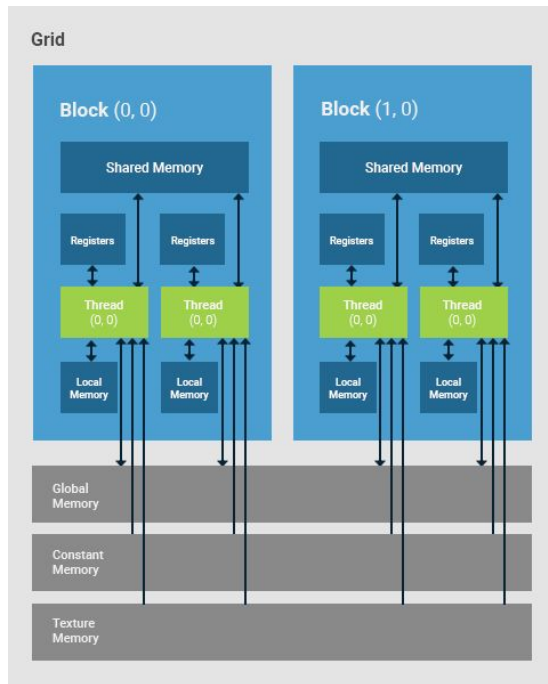
```
cudaMalloc(&device_in, 10 * sizeof(double));  
cudaMalloc(&device_out, 10 * sizeof(double));  
...  
kernel<<< /* #blocks */ 1, /* #threads per block*/ 10 >>>(device_out, device_in);
```

# Device vs Global call arguments

- Every kernel parameter that is a pointer is global
- Device functions are only called inside kernels
- Not all device parameters are necessarily global



Keep a record of global args  
passed to device function  
**gradients (pullbacks)** in the  
differentiation request.  
This works recursively for  
nested device functions as  
well.



# Device vs Global call arguments

```
void kernel_grad_0_2(double *out, double *in, double
*val, double *_d_out, double *_d_val) {
    . . .
    device_fn_2_pullback(in, val, _r_d0, _d_val);
    . . .
}
```

```
void kernel_grad_0_2(double *out, double *in, double
*val, double *_d_out, double *_d_in, double *_d_val)
{
    . . .
    device_fn_2_pullback(in, val, _d_in, _d_val);
    . . .
}
```

```
void kernel_grad_0_2(double *out, double *in, double
val, double *_d_out, double *_d_val) {
    . . .
    device_fn_2_pullback(in, val, _r_d0, &_r0);
    . . .
}
```

local to each thread

# Unique device gradients

```
auto grad_object = clad::gradient(kernel, "out, in");  
auto grad_object = clad::gradient(kernel, "out, in, val");
```

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_in, double *_d_val)`
- 2) `__attribute__((device)) void device_fn_pullback_0_1_3_4(double *in, double *val, double _d_y, double *_d_in, double *_d_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.

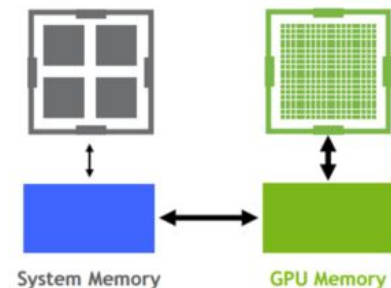
```
__attribute__((device)) void device_fn_pullback(double *in,  
double *val, double _d_y, double *_d_in, double *_d_val) {  
    unsigned int _t1 = blockIdx.x;  
    unsigned int _t0 = blockDim.x;  
    int _d_index = 0;  
    int index0 = threadIdx.x + _t1 * _t0;  
    {  
        atomicAdd(&_d_in[index0],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

# CPU and GPU interaction - Deriving a CPU function with CUDA calls

When deriving a CPU function, we need to make sure that **every parameter passed to the kernel pullback is allocated and set in the GPU**

```
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);  
    cudaMemcpy(_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;  
}
```

```
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; CPU variable  
        kernel_pullback<<<1, 5>>>(out, in, N, _d_out,  
                                   _d_in, &_r0);  
        _d_N += _r0;  
    }  
}
```



# Gradient correctness: Malloc and Free

```
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);
    . . .
    free(out_host);
    cudaFree(out);
    cudaFree(in_dev);
    return res;
}

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



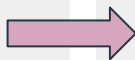
When deriving a memory allocation or deallocation expression, we **allocate/deallocate the derivative** as well and **set it** if needed

```
void fn_grad(double *out, double *in, double *_d_out,
    double *_d_in) {
    . . .
    cudaMalloc(&_d_in_dev, 10 * sizeof(double));
    cudaMemset(_d_in_dev, 0, 10 * sizeof(double));
    cudaMalloc(&in_dev, 10 * sizeof(double));
    . . .
    free(out_host);
    free(_d_out_host);
    cudaFree(out);
    cudaFree(_d_out);
    cudaFree(in_dev);
    cudaFree(_d_in_dev);
}
```

# Gradient correctness: Memcpy

In the reverse-mode AD, the order of cudaMemcpy calls and kernel launches is swapped. Thus, we now have to **reverse the source and destination addresses** and the kind of the copy call and transform the copy to a **plus-assign operation**.

```
double fn(double *out, double *in) {  
    . . .  
    kernel_call<<<1, 10>>>(out, in_dev);  
    double *out_host = (double *)malloc(10 * sizeof(double));  
    cudaMemcpy(out_host, out, 10 * sizeof(double),  
               cudaMemcpyDeviceToHost);  
    . . .  
    return res;  
}  
  
auto grad_object = clad::gradient(fn, "out, in");
```



```
void fn_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, &_amp;r0, &_amp;r1);  
    . . .  
}
```



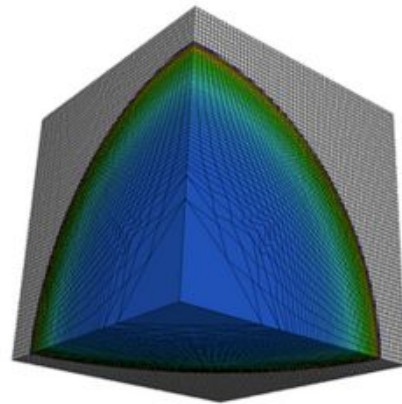
# What have we accomplished?

- Deriving CPU functions containing **kernel invocations** and typical CUDA host functions **without** extra effort from **the user to allocate and initialize GPU variables**
- Users can run both the **original** computations and their **gradient** with a **SINGLE call**

```
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
    . . .
}
```

# Scaling up- LULESH

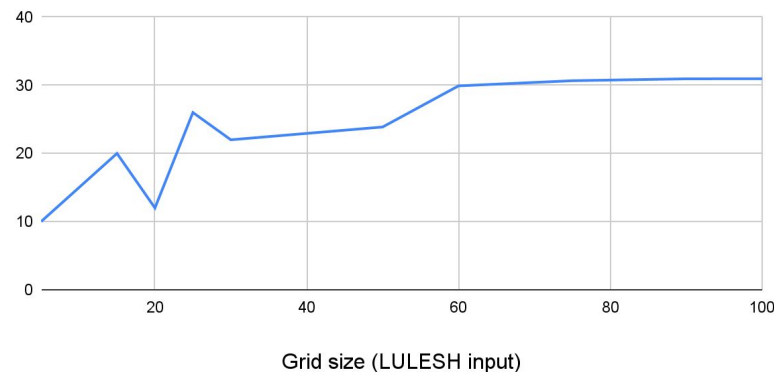
- **What is LULESH?**
  - LULESH is a simplified version of an unstructured explicit shock hydrodynamics solver that models the motion of materials relative to each other when subject to forces. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh.
- **Why benchmark LULESH?**
  - LULESH is representative of real physics codebases
  - It's widely used as an HPC benchmark



# Scaling up- LULESH

- **Any modifications needed for correctness?**
  - Preliminary benchmarks using Enzyme's version of LULESH: Refactor kernels into device functions - **no kernel launch derivation support**
  - With Clad:
    - Some conversions to `const_cast`
    - Redundant storing operations are removed **from the gradient only for performance purposes**
- **What have we observed?**
  - Correct derivatives
  - Room for performance improvement

Overhead of Clad function from the original function - LULESH



\*Benchmarks done on Enzyme's version of LULESH, but using Clad

# Challenges and Future work

- Optimization:
  - Reduce GPU variables created as access to GPU memory is slow and the memory is limited
  - Optimize storing operations
- Extend support:
  - Enable support of shared memory
  - Handle synchronization functions, like `__syncthreads()` and `cudaDeviceSynchronize()`
  - Extend support of CUDA math and host functions
  - **Extend benchmark and application support**

# Outline

- Enabled CUDA support with appropriate atomic operations, capable of detecting allocation patterns and **retrieving correct gradients**
- Enabled AD for CUDA code **(mostly) without requiring code modification**:
  - Black-Scholes from NVIDIA's CUDA samples repository
  - Physics simulation codebases: RSBench and LULESH
- Conducted preliminary performance studies



# Thank you!

5th MODE Workshop

Cla $\partial$