

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

Christina Koutsou,<sup>a,\*</sup> Vassil Vassilev<sup>a</sup> and David Lange<sup>a</sup>

<sup>a</sup>Compiler Research Group, Princeton University,  
Princeton, NJ 08544, USA

E-mail: [christinakoutsou22@gmail.com](mailto:christinakoutsou22@gmail.com), [vassil.vassilev@cern.ch](mailto:vassil.vassilev@cern.ch),  
[david.lange@cern.ch](mailto:david.lange@cern.ch)

GPUs have become increasingly popular for their ability to perform parallel operations efficiently, driving interest in General-Purpose GPU Programming. Scientific computing, in particular, stands to benefit greatly from these capabilities. However, parallel programming systems such as CUDA introduce challenges for code transformation tools due to their reliance on low-level hardware management primitives. These challenges make implementing automatic differentiation (AD) for parallel systems particularly complex.

CUDA is being widely adopted as an accelerator technology in many scientific algorithms from machine learning to physics simulations. Enabling AD for such codes builds a new valuable capability necessary for advancing scientific computing.

Clad is an LLVM/Clang plugin for automatic differentiation that performs source-to-source transformation by traversing the compiler's internal high-level data structures, and generates a function capable of computing derivatives of a given function at compile time. In this paper, we explore how we recently extended Clad to support GPU kernels and functions, as well as kernel launches and CUDA host functions. We will discuss the underlying techniques and real-world applications in scientific computing. Finally, we will examine current limitations and potential future directions for GPU-accelerated differentiation.

*Fifth MODE Workshop on Differentiable Programming for Experiment Design (MODE2025)*  
8-13 June 2025  
Kolymbari, Crete, Greece

---

\*Speaker

## 1. Introduction

It is difficult to challenge the necessity of gradient computation. The first example that usually comes to mind is their presence in physics equations and, thus, in simulations of our world. However, their applications cover a lot more areas, such as control theory, finance, and optimization problems, the most famous being backpropagation in ML.

Consequently, it is imperative that we are in a position to calculate the gradient of a function. A trustworthy but undoubtedly inefficient way of doing this is to write the derivatives by hand. Unfortunately, this would require more and more time as the codebase becomes larger. But if we notice that computing the gradient of a function boils down to a handful of differentiation rules each time, it is then evident that this process can be automated. This is where Automatic Differentiation (AD) becomes useful.

Nowadays, the rise of AI showcased the power of General Purpose GPU programming and offered a new device to be used for implementation, where gradients are often needed. This is justified by the fact that the architecture of the GPUs provides the perfect layout for parallel execution that is also energy and cost efficient, not tailored to any specific application, and programmable through C and C++, provided that programmers use programming models that fit their microarchitectures well. For example, CUDA, HIP, etc fit closely to the underlying vendor hardware and offer ways to squeeze every bit of performance. However, that comes at the cost of a non-straight-forward application programming interface (API) and includes data management and transfer from and to the accelerator hardware. Nevertheless, all these benefits make GPUs very appealing for offloading code and implementing hybrid CPU and GPU applications. It is, therefore, becoming increasingly necessary to extend AD coverage to GPUs, but enabling gradient descent optimizations with AD techniques is still challenging.

In this paper, we present how we added support of gradient- reverse-mode AD- computation of CUDA functions to Clad, an LLVM AD plugin. Our study contains a classic tensor contraction algorithm of two 3-rank tensors. We demonstrate how differentiable Black-Scholes works and then we apply our implementation to a larger scale benchmark from the LULESH suite.

## 2. Background

In order to facilitate the process of explaining and understanding the contributions presented in this paper, we introduce concepts and tools that will be referenced throughout.

**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. There are several approaches to AD: Operator overloading or tracing, where we record and replay the linearized sequence of instructions; source transformation, which is complicated to implement because it requires rebuilding of the entire language infrastructure around the tool, but yields better performance overall; compiler-based source transformation AD, which reuses the compiler language infrastructure and is also difficult to implement, but combines better performance and less maintenance work.

**Clad.** Clad [1] 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. This produced source code is compiled along with the rest of the source code and included in the binary file. Clad can also be instructed to output the derivative source code to a file or to the terminal for transparency and to allow source manipulation. In the latter case, the user has to manually include the altered code in their original program and call it directly, instead of using Clad's API functions to execute what Clad originally produced as the derivative.

**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 are interested in the derivative with respect to many inputs of the function.

**CUDA.** NVIDIA GPUs provide a massively parallel compute platform designed around thousands of lightweight threads. CUDA (Compute Unified Device Architecture) [2] is the programming model and runtime system that exposes this platform to developers. CUDA organizes parallel execution into a hierarchy: threads are grouped into blocks, and blocks are organized into grids. Each thread has a unique index within this hierarchy, which determines the data it processes. CUDA also provides a memory hierarchy, ranging from high-latency global memory to low-latency shared memory and registers, enabling efficient parallel algorithms. This model has become a standard way to program GPUs for scientific computing, machine learning, and numerical simulations.

**CUDA kernel.** A CUDA kernel is a global function of void return type executed in parallel on a GPU.

**Kernel and Device Function Execution.** Kernels are launched by the host (CPU) with a certain grid configuration. Device (GPU) functions can only be called inside kernels. They cannot be launched similarly to kernels to create a new grid configuration for them, instead, each thread running the kernel will execute the device function as many times as it is called.

Previous work on AD on CUDA includes efforts by the team developing Enzyme, another AD tool that computes the derivative at the LLVM IR level [3]. Enzyme supports the derivation of plain device functions, but not of kernels. The only way kernel derivation can occur using Enzyme is by moving the body of the kernel to a `__device__` GPU function and calling Enzyme to differentiate it inside the original kernel. Additionally, to derive kernel calls it is required by the user to move the body of the kernel to another GPU function, define the forward and reverse passes of this function and pass them to Enzyme's API.

Tapenade is a source transformation AD tool that partially supports CUDA. For parallel code, it extracts information from the original code and memory accesses to ensure that no race conditions occur in the gradient [4]. However, this approach is limited by Tapenade's custom CUDA language parser.

Clad, has partial CUDA support in derivation of device functions [5] and this work significantly expands its capabilities for reverse-mode AD.

### 3. Implementation

When introducing a new framework, there are always challenges to ensure compatibility. Here, due to the GPU's architecture and the way it interacts with the CPU, we had to think of the memory layout, the parallel execution potentially leading to race conditions and the fact that communication between CPU and GPU can only occur through CUDA's API.

### 3.1 Race conditions

In the reverse pass, the operations are reversed, which signifies that we can no longer rely on the original function to claim that no write-race conditions are present. 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, leading to undefined behavior. That is, the nature of the gradient transforms benign read-race conditions into write-race conditions. To avoid that, the operations on this memory address are made atomic.

If two threads can access the same memory address, that address must reside either in the shared block memory or in the global memory of the GPU. We focus on the latter, which corresponds to all the pointer parameters of a kernel. These variables, acting as kernel parameters, are allocated through `cudaMalloc()`, a function invoked by the CPU, and passed to the kernel when launched. When deriving a kernel, the adjoint parameters are recorded by Clad as variables on which atomic operations are to be used. Any write operation during the reverse pass on these parameters or on an index of them if they represent array bases, is made atomic.

An example of a write-race condition in the gradient is illustrated in Figure 1, where `_d_val` points to a global memory address on which all threads attempt to perform a plus-assign operation, while in the original code the parameter `val` is only accessed for reading purposes and is not altered at all.

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

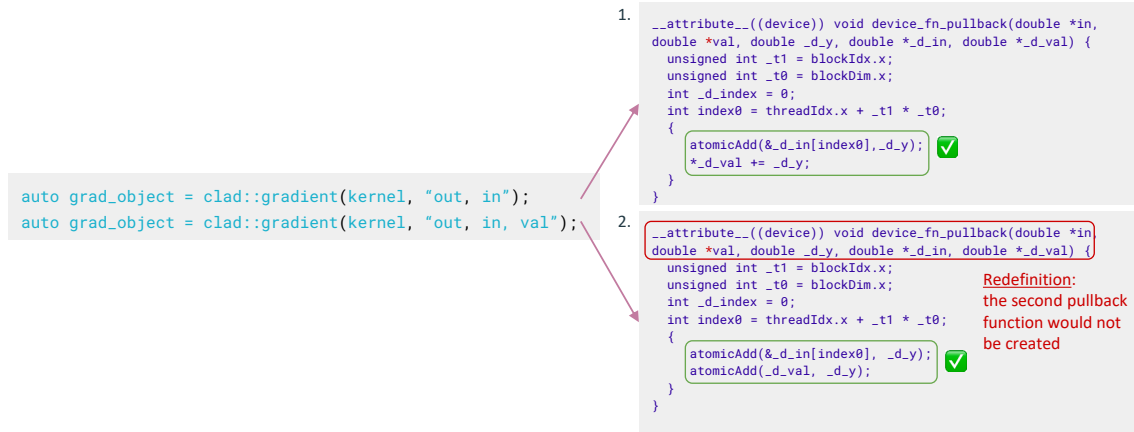
// auto grad_object = clad::gradient(add_kernel, "out, val");
void add_kernel_grad(int *out, int val, int *_d_out, int *_d_val) {
    ...
    int _r_d0 = _d_out[index0];
    _d_val += _r_d0;      → atomicAdd(&_amp;_d_val, _r_d0);
}
```

**Figure 1:** Example of a write-race condition in the reverse pass. In the original pass, all threads read the same value from the global address of `val`, but in the reverse pass, all threads try to write to the same memory address of `_d_val` simultaneously. By making this operation atomic, we’re ensuring that no race conditions occur, without the need for any locks.

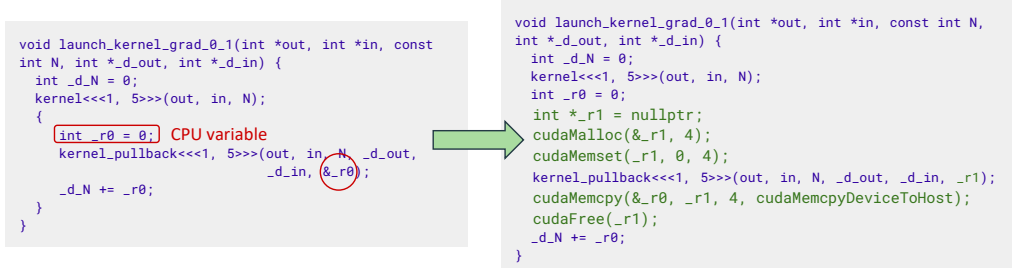
In case we derive a kernel with internal calls to device functions, we can keep track of which of those pointer parameters are passed to these calls. Specifically, atomic operations should be used on the kernel derivatives passed to the gradients of the device calls, known as pullbacks.

However, depending on the differentiation request, different combinations of call arguments can be global, resulting in pullback functions with the same signature, but different bodies. Clad already avoids redefinition errors by storing the pullbacks created and checking against them before differentiating any new device call. Hence, only one of these pullback functions would be created and this would only be safe for the first specified configuration. This is illustrated in Figure 2.

An easy workaround to this problem is to append to the pullback function’s name the index of every global parameter. This is a technique that is in a way already utilized in Clad for the top-level function to be derived.



**Figure 2:** Example of a pullback redefinition. Two different derivation requests result in the same function signature for the gradient (pullback) of a function call, while they differ in their parameters that reside in global memory. To avoid a compiler error of function redefinition, the second pullback wouldn't be created, leading to race conditions for the execution of the second gradient request. A simple solution is to append the indices of the parameters that correspond to global call arguments for each pullback to its name.

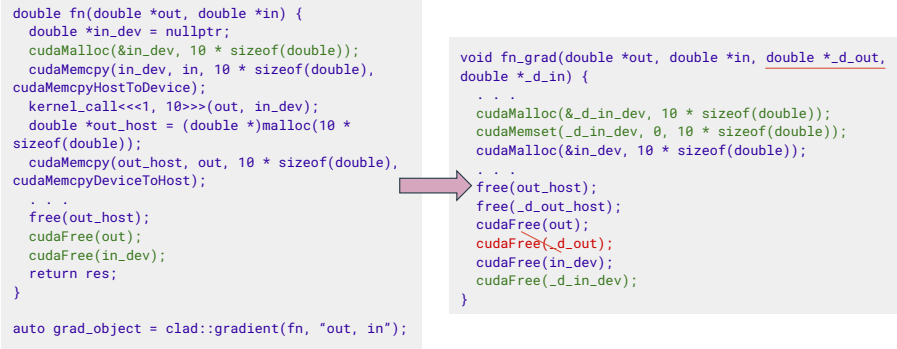


**Figure 3:** Example of a kernel pullback launch. Clad ensures that each variable passed to the kernel's gradient is allocated on the GPU and initialized, and afterwards, retrieved and freed appropriately.

### 3.2 CPU-GPU interaction

One way a CPU and GPU interact is when we derive a CPU function that includes a kernel call. Since Clad will derive the kernel as well, we need to make sure that every variable passed to the kernel pullback is properly allocated and set in the GPU, and retrieve the results before freeing them in the end.

Additionally, Clad mimics the allocation and de-allocation pattern of the original host (CPU) function by cloning these operations for the derivatives of the variables concerned, while also initializing them when needed. Specifically for `cudaFree()`, as void functions include their output in their parameter list, Clad avoids duplicating the operation for the derivative when this value was requested by the user and is, hence, an output of interest. In Figure 4, for instance, `out` is part of the argument list in the differentiation request. Therefore, the user provides `_d_out` to the gradient function and is interested in its value. Consequently, although in the original code `out` is not needed at the end of the function's execution, this is not the case for `_d_out` in the gradient function, which leads us to omit calling `cudaFree()` on this memory address.



```
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");
```

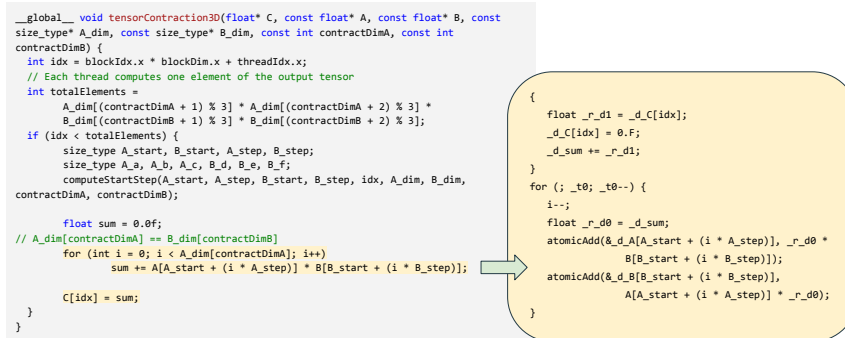
```
void fn_grad(double *out, double *in, double *_d_out,
double *_d_in) {
    . . .
    cudaMalloc(&_d_in_dev, 10 * sizeof(double));
    cudaMemset(_d_in_dev, 0, 10 * sizeof(double));
    cudaMalloc(&in_dev, 10 * sizeof(double));
    . . .
    free(out_host);
    free(_d_out_host);
    cudaFree(out);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

**Figure 4:** Example of handling `cudaMalloc()` and `cudaFree()` calls. Clad mimics the allocation and de-allocation pattern of the original function for the derivative variables, while ensuring that they are properly initialized and not freed when they are requested by the user.

Lastly, `cudaMemcpy` is handled as a common function call, which means that its execution order will change. In its reverse pass, we swap the direction of the transfer of the data, and swap the destination and source addresses. However, as the derivative is updated in each step, the operation has to change from a plain copy assignment to a plus-assign operation. Thus, a custom derivative is provided for `cudaMemcpy`.

#### 4. Case Study

To evaluate the relevance of our contributions, we tested it on a classic tensor contraction code example of two rank-3 tensors, as tensor contractions are heavily used in ML, and their derivatives are needed for backpropagation. The math formula,  $C_{ijlm} = \sum_{k=1}^K A_{ijk} \cdot B_{klm}$ , is translated into code as shown in Figure 5. Clad successfully derived the example and produced correct results that were verified against the results of the same code using PyTorch’s backpropagation method instead.



```
_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;
    }
}
```

```
{
    float _r_d1 = _d_C[idx];
    _d_C[idx] = 0.0f;
    _d_sum += _r_d1;
}
for (; _t0; _t0--) {
    i--;
    float _r_d0 = _d_sum;
    atomicAdd(&_d_A[A_start + (i * A_step)], _r_d0 *
    B[B_start + (i * B_step)]);
    atomicAdd(&_d_B[B_start + (i * B_step)],
    A[A_start + (i * A_step)] * _r_d0);
}
```

**Figure 5:** Tensor contraction demo

Another example of a use case was taken directly from NVIDIA’s CUDA samples repository [6]: it is the implementation of the Black-Scholes economic model. There, we are interested in the "Greeks", which are the partial derivatives of the prices with respect to specific variables, in order to see the sensitivity of the price to certain changes in the parameters. Clad again managed

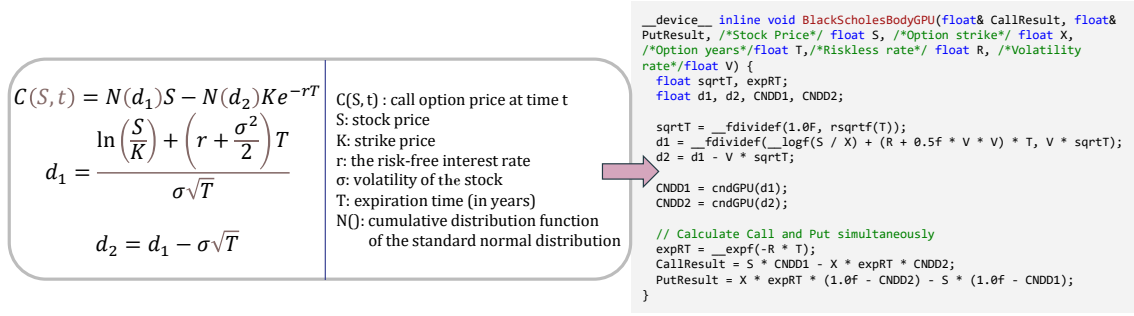


Figure 6: Black-Scholes demo

to produce the derivative code and obtained correct results, which were cross-checked against the simple derivative formulas, the “Greeks”, written by hand and evaluated on the same inputs.

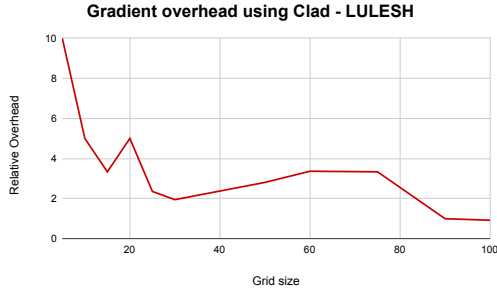
## 5. Benchmarks

To build on the scale of applications that Clad can support, we chose LULESH [7], a common example of a larger physics codebase. 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, by dividing the volume into a grid.

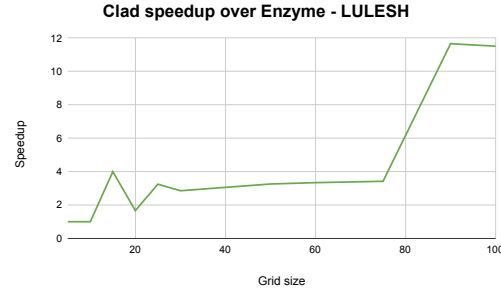
Initially, the version of LULESH used was the one altered by the Enzyme team, so that we could verify the results of Clad. It is important to note that this version of LULESH only included device function derivation, since Enzyme does not support kernel derivation like Clad does. Clad managed to differentiate through it, yielding correct results, with the only additional modifications needed being some `const_cast` conversions. When looking into the performance, by taking advantage of the fact that Clad produces source code that can be compiled by C++ compilers, we noticed that the performance of the gradient was significantly degraded by the usage of some storing operations to tapes that were actually redundant. Therefore, we could directly alter the source code given by Clad and improve the performance. Unfortunately, even after those changes, there was still a large overhead compared to the execution of the original function, deeming the improvement of the tape’s implementation necessary. After reworking the tape, Clad managed to limit the gradient’s execution to less than four times overhead compared to the original function for grid dimensions above 20 and even noted a relative overhead of 1 as the grid size increased to 90 and beyond, indicating that the execution times of the original and the gradient functions were nearly identical. Moreover, Clad outperformed Enzyme for all tested grid sizes, as shown in Figure 8. These results relied entirely on the code produced by Clad, without any manual source manipulation, and, hence, could be improved further by commenting out unnecessary tape operations.

The code used for the tests is available at <https://github.com/kchristin22/clad/tree/lulesh-1.0-device> (commit 0a5c143), while Enzyme’s code can be found at <https://github.com/wsmoses/Enzyme-GPU-Tests>. For each input, results were evaluated at least five times, with the geometric mean taken as the final value. The benchmarks were run on an NVIDIA GeForce GTX 1650 GPU.





**Figure 7:** Time overhead of Clad’s gradient computation relative to the original function (times the original runtime), measured on Enzyme’s LULESH version using Clad instead.



**Figure 8:** Speedup of Clad compared to Enzyme (Enzyme runtime / Clad runtime).

## 6. Future Work

As mentioned above, tape operations can have a major effect on performance. Thus, a combination of To-Be-Recorded (TBR) and Activity analyses that limits the storing operations to the absolute minimum would reduce Clad’s output without requiring manual intervention from the user. It would also be beneficial to reduce the amount of GPU variables created, because access to GPU memory is slow and the memory is limited.

As far as extension of support is considered, we would like to build on the advantage of being able to derive kernels in comparison to other AD tools, by supporting shared memory usage in CUDA. Moreover, an interesting challenge would be to handle synchronization functions, such as `__syncthreads()` and `cudaDeviceSynchronize()`. An easier one is to support other CUDA math and host functions. Lastly, we want to extend our benchmark and application support.

## 7. Conclusion

In a nutshell, what we have accomplished is that we managed to support reverse-mode AD on CUDA and produce correct results by using atomic operations properly and detecting memory allocation patterns. All this with minimal to no extra modification to the original code. This is demonstrated in the Black-Scholes example and in physics benchmarks like LULESH and, recently, RS Bench. Having ensured correctness, we continue to focus on performance.

## Acknowledgments

The authors would like to thank Parth Arora for his valuable insights and mentorship during the implementation of this project.

## References

- [1] V. Vassilev, M. Vassilev, A. Penev, L. Moneta, and V. Ilieva. *Clad — Automatic Differentiation Using Clang and LLVM*. *J. Phys.: Conf. Ser.* **608** 012055



- [2] NVIDIA. *CUDA C++ Programming Guide v13.0*. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>.
- [3] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert, *Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme*, in proceedings of *The International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. **SC '21**. New York, NY, USA.
- [4] Jan Hückelheim and Laurent Hascoët, *Automatic Differentiation of Parallel Loops with Formal Methods*, in proceedings of the International Conference on Parallel Processing, **ICPP '22**.
- [5] Ioana Ifrim, Vassil Vassilev, and David J. Lange. *GPU Accelerated Automatic Differentiation With Clad*. **20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research**.
- [6] NVIDIA. *CUDA Samples repository v12.5*. **Black-Scholes**.
- [7] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*, **LLNL-TR-641973**. Technical Report.