GPU Accelerated Automatic Differentiation With Clad

Ioana Ifrim, Vassil Vassilev and David J Lange

Department of Physics, Princeton University, Princeton, NJ 08544, USA

E-mail: ii3193@princeton.edu, vassil.vassilev@cern.ch, david.lange@princeton.edu

Abstract.

Automatic Differentiation (AD) is instrumental for science and industry. It is a tool to evaluate the derivative of a function specified through a computer program. The range of AD application domain spans from Machine Learning to Robotics to High Energy Physics. Computing gradients with the help of AD is guaranteed to be more precise than the numerical alternative and have a low, constant factor more arithmetical operations compared to the original function. Moreover, AD applications to domain problems typically are computationally bound. They are often limited by the computational requirements of high-dimensional parameters and thus can benefit from parallel implementations on graphics processing units (GPUs).

Clad aims to enable differential analysis for C/C++ and CUDA and is a compiler-assisted AD tool available both as a compiler extension and in ROOT. Moreover, Clad works as a plugin extending the Clang compiler; as a plugin extending the interactive interpreter Cling; and as a Jupyter kernel extension based on xeus-cling.

We demonstrate the advantages of parallel gradient computations on GPUs with Clad. We explain how to bring forth a new layer of optimization and a proportional speed up by extending Clad to support CUDA. The gradients of well-behaved C++ functions can be automatically executed on a GPU. The library can be easily integrated into existing frameworks or used interactively. Furthermore, we demonstrate the achieved application performance improvements, including (≈10x) in ROOT histogram fitting and corresponding performance gains from offloading to GPUs.

1. Introduction

Many tasks in science and industry are or can be amenable to gradient-based optimization. Gradient-based optimization is an effective way to find optimal parametrization for multiparameter processes that can be expressed with an objective function. The optimization algorithms rely on a function's derivatives or gradients. Multiple approaches to obtain gradients of mathematical functions exist, including manual, numerical, symbolic and automatic/algorithmic. Automatic/algorithmic differentiation (AD) is a computer program transformation that relies on the fact that every program can be decomposed into elementary operations to which the chain rule can be applied. Unlike numeric or symbolic differentiation, AD enables computation of exact gradients and is free of systematic and round-off errors. In addition, AD provides highly efficient means to control the computational complexity of the produced gradients and derivatives [1].

AD is not new, and has become increasingly popular due to the *backpropagation* technique in machine learning (ML). ML models are trained using large data sets, with an optimization procedure relying on the computation of derivatives for error correction. The scalable AD-produced gradients combined with recent increases in computational capabilities are an enabling

factor for sciences such as oceanology and geosciences [2]. High Energy Physics (HEP) is well positioned to benefit from gradient-based optimizations allowing for better parameter fitting, sensitivity analysis, Monte Carlo simulations and general statistical analysis for uncertainty propagation through parameter tracking [3]. Data-intensive fields such as ML and HEP advance their scalability by employing various parallelization techniques and hardware. For ML, it is common that part, if not all, of the computations are run on GPGPU-accelerators, taking advantage of parallel computing platforms and programming models, such as CUDA. HEP has invested in systems for GPGPUs for real-time data processing [4] and clustering algorithms [5]. It has also started to reorganize its data analysis and statistical software to be more susceptible towards parallelism. Packages such as RooFit [6] can see large speed-ups via gradient-based optimization developments and hardware acceleration support.

The AD program transformation has two distinct modes: *forward* and *reverse* accumulation mode. The forward accumulation mode has a lower computational cost for functions taking fewer inputs and returning more outputs. The reverse accumulation mode is better suited to "reducing" functions that have fewer outputs and more inputs. The reverse accumulation mode is more difficult to implement, but is critical for many use cases as the execution time complexity of the produced gradient is independent of the number of inputs. The implementation of efficient reverse accumulation AD for parallel systems such as CUDA is challenging as it is a non-trivial task to preserve the parallel properties of the original program.

This paper describes our progress with the CUDA support of the compiler-assisted AD tool, Clad. We demonstrate the integration of Clad with interactive development services such as Jupyter Notebooks [7], Cling [8], Clang-Repl [9] and ROOT [10].

2. Background

The AD program transformation is particularly useful for gradient-based optimization because it offers upper bound asymptotic complexity computation guarantees. That is, the cost of the reverse accumulation scales linearly as $O(m)$ where m is the number of output variables. Moreover, the cheap gradient principle [11] demonstrates that the "cost of computing the gradient of a scalar-valued function is nearly the same (often within a factor of 5) as that of simply computing the function itself" [12]. These guarantees make gradient-descent (also known as error backpropagation in ML) optimization computationally feasible in areas such as deep learning.

AD typically starts by building a computational graph, or a directed acyclic graph of mathematical operations applied to an input. There are two approaches to AD, which differ primarily in the amount of work done before the program execution. The *tracing* approaches construct and process a computational graph for the derivative, by exploiting the ability to overload operators in C++. They perform an evaluation at the time of execution, for every function invocation. *Source transformation* approaches generate a derivative function at compile time to create and optimize the derivative only once. The tracing approach, which is implemented in C++ by ADOL-C[13], CppAD [14], and Adept [15], is easier to implement and adopt into existing codebases. Source transformation shows better performance, but it usually covers a subset of the language. In C++, Tapenade [16] has a custom language parsing infrastructure.

Being a language of choice for Machine Learning, Python hosts one of the most sophisticated AD systems. *JAX* is a trace-based AD system that differentiates a sub-language of Python oriented towards ML applications [17]. *Dex* aims to give better AD asymptotic and parallelism guarantees than JAX for loops with indexing [18]. Trace-based ML AD tools are developed to exploit ML workflow characteristics such as having little branching, no branching depending on input data or probabilistic draws at runtime, and almost no adaptive algorithms. However, investigations into further generalized approaches to AD have shown superior performance [19].

HEP projects that exploit AD include work on MadJAX [20], ACTS [21] and ROOT [22]. Gradients are used in objective-function optimizations, or in the propagation of uncertainty

in different applications. A few projects aim to introduce AD-based pipelines for end-to-end sensitivity analysis in bigger scale systems such as reconstruction or down-stream analysis tasks [23, 24]. The community creates events and documents to capture the growing interest [25].

Recent advancements of production quality compilers like Clang allow tools to reuse the language parsing infrastructure, making it easier to implement source transformation AD. ADIC [26], Enzyme [27] and Clad [28] are compiler-based AD tools using source transformation. The increasing importance of AD is evident in newer programming languages such as Swift and Julia where it is integrated deep into the language [29, 30]. Clad is a compiler-based source transformation AD tool for C++. Clad uses the high-level compiler program representation called *abstract syntax tree* (AST). It can work as an extension to the Clang compiler, as an extension to the C++ interpreter Cling, or Clang-Repl, and is available in the ROOT software package. Clad implements forward and reverse accumulation modes and can provide higher order derivatives. It produces C++ code of the gradient, allowing for easy inspection and verification.

3. Design and Implementation

The Clang frontend builds a high-level program representation in the form of an AST. The programmatic AST synthesis might look challenging and laborious to implement, but yields several advantages. Having access to such a high-level code representation means being able to follow the high-level semantics of the algorithm and it facilitates domain-specific optimizations. This can be used to automatically generate code (re-targeting C++) for hardware acceleration with corresponding scheduling. Clang supports an entire family of languages, such as C, C++, CUDA and OpenMP. For the most part, the AST for these languages is shared and thus the addition of CUDA support focuses only on the different constructs.

The CUDA programming model provides a separation between the device (GPU) and host (CPU) using language attributes such as *__global__*, *__host__* and *__device__*. A typical function that is to be differentiated can become a device executable function and its execution can be scheduled via kernels (functions that are marked with the *__global__* attribute) for parallel computation. In order to maintain this support for the gradient function, it is sufficient for Clad to transfer the relevant attributes to the generated code and make the Clad-based run-time data structures compatible with CUDA by adding the relevant attributes.

```
#include "clad/Differentiator/Differentiator.h"
#define N 512
using arrtype = double[1];
// A device function to compute a gaussian
__device__ __host__ double gauss(double x,
    double p, double sigma) {
  double t = - (x - p) * (x - p) /
             (2 * sigma * sigma);
  return pow(2 * M_PI, -1/2.0) *
         pow(sigma, -0.5) * exp(t);
}
// Tells Clad to create a gradient
auto gauss_g = clad::gradient(gauss, "x, p");

// Forward declare the generated function so
// that the compiler can put it on the device.
void gauss_grad_0_1(double x, double p,
    double sigma, clad::array_ref<double> _d_x,
    clad::array_ref<double> _d_p)
    __attribute__((device))
    __attribute__((host));
```

```
// The compute kernel including scheduling
__global__ void compute(double* x,
    double* p, double sigma, arrtype dx,
    arrtype dp) {
  int i = blockIdx.x * blockDim.x +
          threadIdx.x;
  // Runs `N` different compute units,
  // each unit computes the gradient wrt
  // each parameter set
  if (i < N)
    gauss_grad_0_1(x[i], p[i], sigma,
                   dx[i], dp[i]);
}

int main() {
  // CUDA device memory allocations...
  compute<<<N/256+1, 256>>>(dev_x, dev_p,
                            dev_sigma,
                            dx, dp);
  // Utilize the results...
}
```

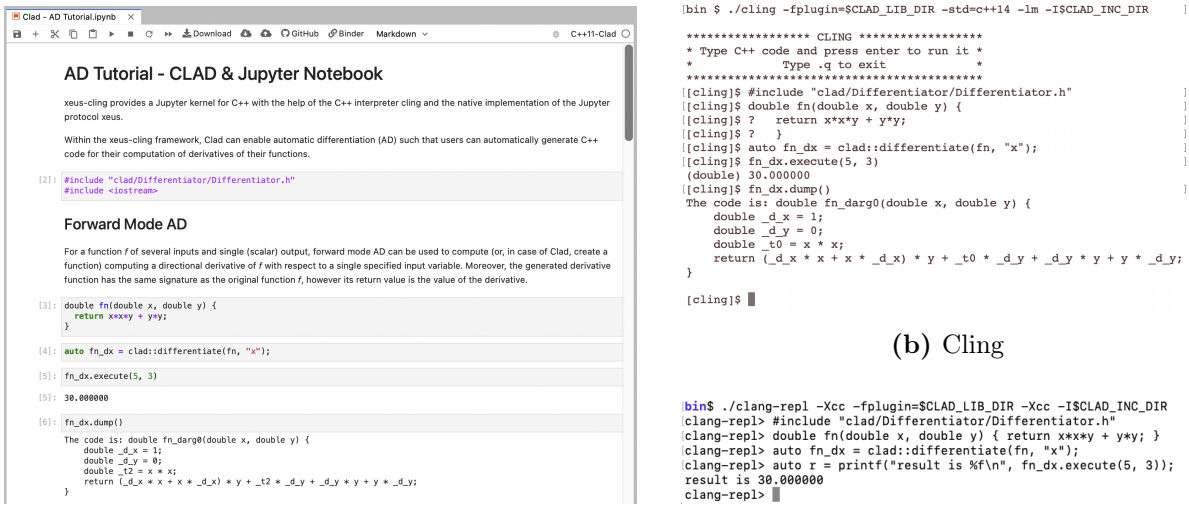
Listing 1: Clad-CUDA support example of a function and generated gradient

In essence, Clad operates on the same footing as the compiler's template instantiation logic. As Clad *visits* the AST, each node is cloned and differentiated. The differentiation rules are implemented in separate classes depending on the accumulation mode. In Listing 1, Clad differentiates a multidimensional normal distribution implemented as a device function. In this particular example, Clad generates the *gauss_grad_0_1* gradient executable in a CUDA environment. The generated gradient for the device function can be called from *compute* in the same way the original function was, to take care of the level of parallelism.

Currently, preserving the parallel properties of the generated gradient relies on the user. In general, preserving parallel properties after AD transformation is still an ongoing research topic, however some interesting results can be achieved automatically based on the gradient content. We are working to automatically preserve the semantics of the generated code as in some cases the parallel reads from memory of the original function become parallel writes to memory in the produced gradient. In addition, the parallelism scheme of the original function might not be optimal for the gradient function. We are working to make use of the information available in Clad to find a greater degree of parallelism for any produced code.

4. Integration and Results

Interactive prototyping and exchanges for collaborative work is the backbone of research. Being a Clang plugin, Clad can easily integrate with the LLVM-based ecosystem. For example, Clad integrates with Jupyter notebooks, the C++ interactive interpreters Clang-Repl and Cling.



(a) Jupyter **(b)** Cling **(c)** Clang-Repl

Figure 1: Clad Integrated in Interactive Environments

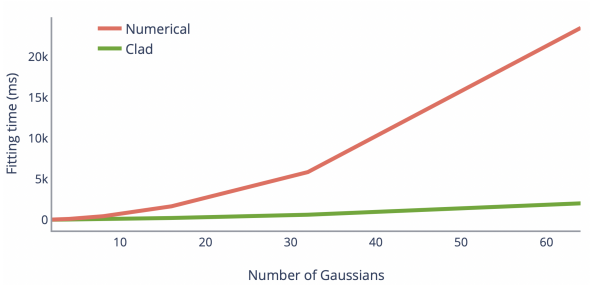
Figure 1a demonstrates that Clad can be used in a Jupyter notebook via xeus-cling. Xeus-cling enables interactive C++ for Jupyter notebooks. Figure 1b shows that Clad works with the interactive C++ interpreter Cling. Figure 1c demonstrates that Clad works in a terminal setup with Clang-Repl available in LLVM (since version 13).

AD with Clad is integrated in ROOT to provide an efficient way to calculate the gradient required for minimization or fitting with a function. This facility is available in ROOT's *TFormula* class via the *GradientPar* and the *HessianPar* interfaces. The functionality can be used within the *TF1* fitting interfaces as well, shown in figure 2a. Replacing the numerical gradients yield promising results. In Figure 2b we compare ROOT's implementation of numerical differentiation for generating gradients, and AD-produced gradients with Clad. As expected by

the AD theory, gradients produced by reverse mode AD scale better as the time complexity of the computation is independent on the number of inputs. The better performance of Clad AD approach versus the numerical differentiation, via the central finite difference method, has been previously studied [22]. Further performance improvements can be achieved by moving the second order derivatives (the Hessian matrix) to use Clad. Currently, they are computed numerically as ROOT's minimizer must be adapted to use externally provided Hessians.

```
auto f1 = new TF1("f1", "gaus");
auto h1 = new TH1D("h1", "h1", 1000, -5, 5);
double p1[] = {1, 0, 1.5}, p2[] = {100, 1, 3}
f1->SetParameters(p1); f1->SetParameters(p2);
h1->FillRandom("f1", 100000);
// Enable Clad in TFormula.
f1->GetFormula()->GenerateGradientPar();
auto r2 = h1->Fit(f1, "S G Q N"); // clad
// ...
```

(a) Example use of Clad in TH1.



(b) Scaling of many gaussian fits.

Figure 2: Comparison of fitting time using gradients produced by Clad (in green) and numerically (red) of sums of Gaussians

We will work with the RooFit development team to integrate Clad into RooFit [31]. RooFit processes more computationally-intense operations and the benefit from automatically generated gradients will have a significant performance impact [32]. Clad could be also used to differentiate the new batch-based CUDA backend in RooFit or re-target C++ gradients for GPU execution.

5. Conclusion and Future Work

In this paper, we introduced automatic differentiation and motivated its advantages for tasks oriented towards gradient-based optimization. We demonstrated the basic usage of Clad and described the advantages provided by it being a compiler-assisted tool. We outlined the advancements in the area of CUDA in Clad. We demonstrated the ease of use via simple use cases of Clad and we showed how the tool integrates with other interactive systems such as Jupyter, Cling and Clang-Repl. We provided some performance results in using AD in ROOT's histogram fitting logic where we compared against the numerical differentiation approach.

We aim to further advance the GPU support of Clad, and to preserve better the parallel algorithm properties. One of the immediate next steps is to be able to differentiate the GPU kernel code automatically along with the already supported device function differentiation. The planned automatic kernel dispatch will utilize the information available in Clad to provide the optimal scheme for parallel computation for the produced gradient code. We plan to implement more advanced program analyses based on the Clang static analyzer infrastructure to allow more efficient code generation. We are working on extending the coverage of C++ and CUDA language features such as support of polymorphism (virtual functions) and differentiation with respect to aggregate types (for example class member variables). We work on a Clad-based backend for a statistical modelling tool with RooFit in order to provide efficient means to compute gradients.

6. Acknowledgments

This project is supported by National Science Foundation under Grant OAC-1931408. Some of the code contributions were facilitated by the 2021 Google Summer of Code program.

References

1. Verma A. An introduction to automatic differentiation. Current Science 2000 :804–7
2. Qin J, Liang S, Li X, and Wang J. Development of the adjoint model of a canopy radiative transfer model for sensitivity study and inversion of leaf area index. IEEE Transactions on Geoscience and Remote Sensing 2008; 46:2028–37
3. Ramos A. Automatic differentiation for error analysis. arXiv preprint arXiv:2012.11183 2020
4. Bruch D vom. Real-time data processing with GPUs in high energy physics. Journal of Instrumentation 2020; 15:C06010
5. Chen Z, Di Pilato A, Pantaleo F, and Rovere M. GPU-based Clustering Algorithm for the CMS High Granularity Calorimeter. *EPJ Web of Conferences*. Vol. 245. EDP Sciences. 2020 :05005
6. Hageboeck S. What the new RooFit can do for your analysis. arXiv preprint arXiv:2012.02746 2020
7. Xeus-cling project on Github. 2021. Available from: https://github.com/QuantStack/xeus-cling
8. Vasilev V, Canal P, Naumann A, and Russo P. Cling – The New Interactive Interpreter for ROOT 6. Journal of Physics: Conference Series 2012 Dec; 396:052071. DOI: 10.1088/1742-6596/396/5/052071. Available from: https://doi.org/10.1088/1742-6596/396/5/052071
9. Vassilev V. Cling Transitions to LLVM's Clang-Repl. Blog post. https://root.cern/blog/cling-in-llvm/. 2022
10. Brun R and Rademakers F. ROOT — An object oriented data analysis framework. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 1997; 389. New Computing Techniques in Physics Research V:81–6. DOI: https://doi.org/10.1016/S0168-9002(97)00048-X. Available from: http://www.sciencedirect.com/science/article/pii/S016890029700048X
11. Griewank A and Walther A. Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM, 2008
12. Kakade SM and Lee JD. Provably correct automatic sub-differentiation for qualified programs. Advances in neural information processing systems 2018; 31
13. Griewank A, Juedes D, and Utke J. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. ACM Transactions on Mathematical Software (TOMS) 1996; 22:131–67
14. Bell BM. CppAD: A package for differentiation of C++ algorithms. 2008 06-01. http://www.coin-or.org/CppAD 2021
15. Hogan RJ. Fast reverse-mode automatic differentiation using expression templates in C++. ACM Transactions on Mathematical Software (TOMS) 2014; 40:1–16
16. Hascoet L and Pascual V. The Tapenade automatic differentiation tool: principles, model, and specification. ACM Transactions on Mathematical Software (TOMS) 2013; 39:1–43
17. Frostig R, Johnson MJ, and Leary C. Compiling machine learning programs via high-level tracing. Systems for Machine Learning 2018 :23–4
18. Paszke A, Johnson D, Duvenaud D, Vytiniotis D, Radul A, Johnson M, Ragan-Kelley J, and Maclaurin D. Getting to the point. index sets and parallelism-preserving autodiff for pointful array programming. arXiv preprint arXiv:2104.05372 2021
19. RFC: C++ Gradients (TensorFlow Community Proposal). 2021. Available from: https://github.com/tensorflow/community/pull/335

20. Heinrich L and Kagan M. Differentiable Matrix Elements with MadJax. arXiv preprint arXiv:2203.00057 2022
21. Ai X, Allaire C, Calace N, Czirkos A, Ene I, Elsing M, Farkas R, Gagnon LG, Garg R, Gessinger P, et al. A common tracking software project. arXiv preprint arXiv:2106.13593 2021
22. Vassilev V, Efremov A, and Shadura O. Automatic Differentiation in ROOT. *EPJ Web of Conferences*. Vol. 245. EDP Sciences. 2020 :02015
23. De Castro P and Dorigo T. INFERNO: Inference-aware neural optimisation. Computer Physics Communications 2019; 244:170–9
24. Simpson N and Heinrich L. neos: End-to-End-Optimised Summary Statistics for High Energy Physics. arXiv preprint arXiv:2203.05570 2022
25. Baydin AG, NYU KC, Feickert M, Gray L, Heinrich L, NYU AH, Neubauer AMVM, Pearkes J, Simpson N, Smith N, et al. Differentiable programming in high-energy physics. Submitted as a Snowmass LOI 2020
26. Bischof CH, Roh L, and Mauer-Oats AJ. ADIC: an extensible automatic differentiation tool for ANSI-C. Software: Practice and Experience 1997; 27:1427–56
27. Moses WS and Churavy V. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. *Advances in Neural Information Processing Systems 33*. 2020
28. Vassilev V, Vassilev M, Penev A, Moneta L, and Ilieva V. Clad — Automatic Differentiation Using Clang and LLVM. Journal of Physics: Conference Series 2015; 608. http://stacks.iop.org/1742-6596/608/i=1/a=012055:012055
29. Saeta B and Shabalin D. Swift for TensorFlow: A portable, flexible platform for deep learning. Proceedings of Machine Learning and Systems 2021; 3
30. JuliaDiff – Differentiation tools in Julia. https://juliadiff.org/. 2022
31. Verkerke W and Kirkby DP. The RooFit toolkit for data modeling. eConf 2003; C0303241:MOLT007. arXiv: physics/0306116 [physics]
32. Bos EP, Burgard CD, Croft VA, Hageboeck S, Moneta L, Pelupessy I, Attema JJ, and Verkerke Wr. Faster RooFitting: Automated parallel calculation of collaborative statistical models. *EPJ Web of Conferences*. Vol. 245. EDP Sciences. 2020 :06027