Automatic Differentiation in C++ and CUDA using Clad

Ioana Ifrim, Princeton University
compiler-research.org

This project was supported by National Science Foundation under Grant OAC-1931408

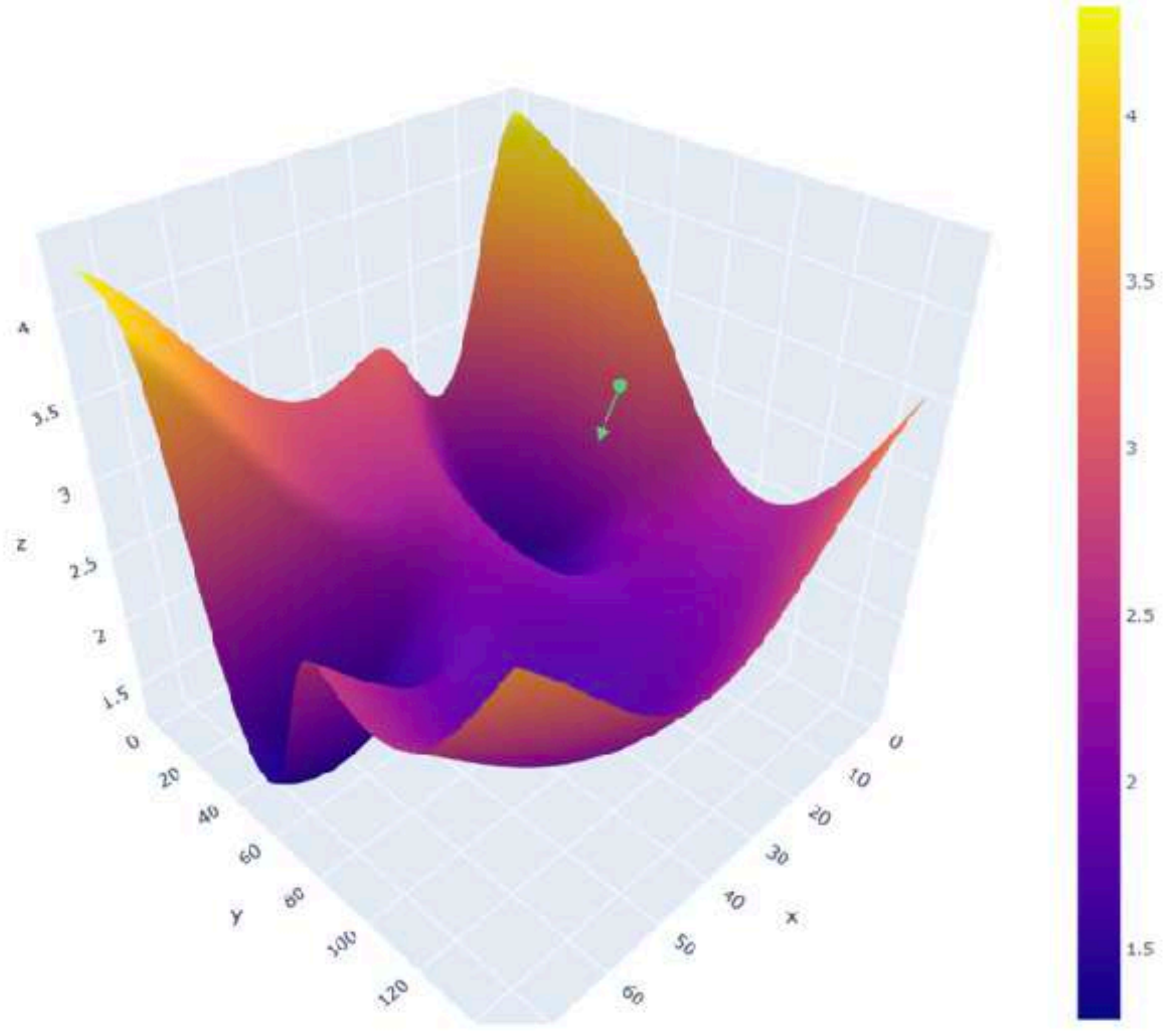
Content

Motivation

In mathematics and computer algebra, automatic differentiation (AD) is defined as a set of techniques used for numerically evaluating the derivative of a function specified by a computer program.

Automatic differentiation is an alternative technique to Symbolic differentiation or Numerical differentiation (the method of finite differences) and powers gradient-based optimisation algorithms used in applications such as Deep Learning, Robotics, High Energy Physics, etc.

The aim of Clad is to provide automatic differentiation for C/C++ which works without code modification



Deep Learning use-case : Gradient Descent = gradient of the cost function with respect to the neural network parameters

AD Approaches

Classification

Implementation approaches in AD can be classified based on the amount of work done at compile time. Thus, we can identify several approaches: Domain Specific Languages (DSL), Tracing / Taping and Source Transformation

By keeping all the intricate knowledge of the original source code, source transformation approaches enable optimisation

Domain Specific Languages (DSL)	Tracing / Taping	Source Transformation
source code transformation is performed on a data flow graph (computation graph)	the compute graph is constructed as the program is executed, the execution is recorded, transformed and compiled "just-in-time"	the compute graph is constructed before compilation and then transformed and compiled
requires both the code to be rewritten and the DSL to provide support for all the operations in the original code	typically uses operator overloading (special floating point type); replaces all elementary operations by the overloaded	typically uses a custom parser to build code representation and produce the transformed code
tailored implementation	easy to implement	difficult to implement (especially for C++)
the speed of this approach is correlated with the similarity factor between the DSL and the original code	inefficient, needs code modification	efficient as many computations and optimisations are done ahead of time
Theano, TensorFlow, PyTorch	C++ : ADEPT, Python: JAX	Tapenade, Enzyme, Clad

Clad: An approach to source transformation AD

Clad uses the source transformation approach by statically analysing the original code to produce a gradient function in the source code language

- mitigates the difficulties related to custom C++ parsers
- having full access to the Clang compiler's internals means that Clad is able to follow the high-level semantics of algorithms and can perform domain-specific optimisations
- it can automatically generate code (re-targeting C++) on accelerator hardware with appropriate scheduling
- has a direct connection to compiler diagnostics engine and thus can produce precise and expressive diagnostics positioned at desired source locations

```
#include "clad/Differentiator/Differentiator.h"
#include <iostream>

double f(double x, double y) { return x * y; }

int main() {

  auto f_dx = clad::differentiate(f, “x");

  // derivative of ‘f' - (x, y) = (3, 4)
  std::cout << f_dx.execute(3, 4) << std::endl;
  // prints: 4

  f_dx.dump(); // prints:
  /*
     double f_darg0(double x, double y) {
         double _d_x = 1;
         double _d_y = 0;
         return _d_x * y + x * _d_y;
     }
   */
}
```

Clad Example

Clad.AD Plugin for Clang

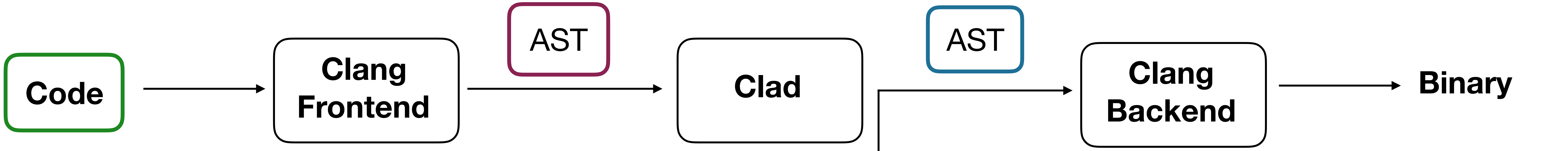
Clad is a compiler plugin extending Clang able to produce derivatives in both forward and reverse mode:

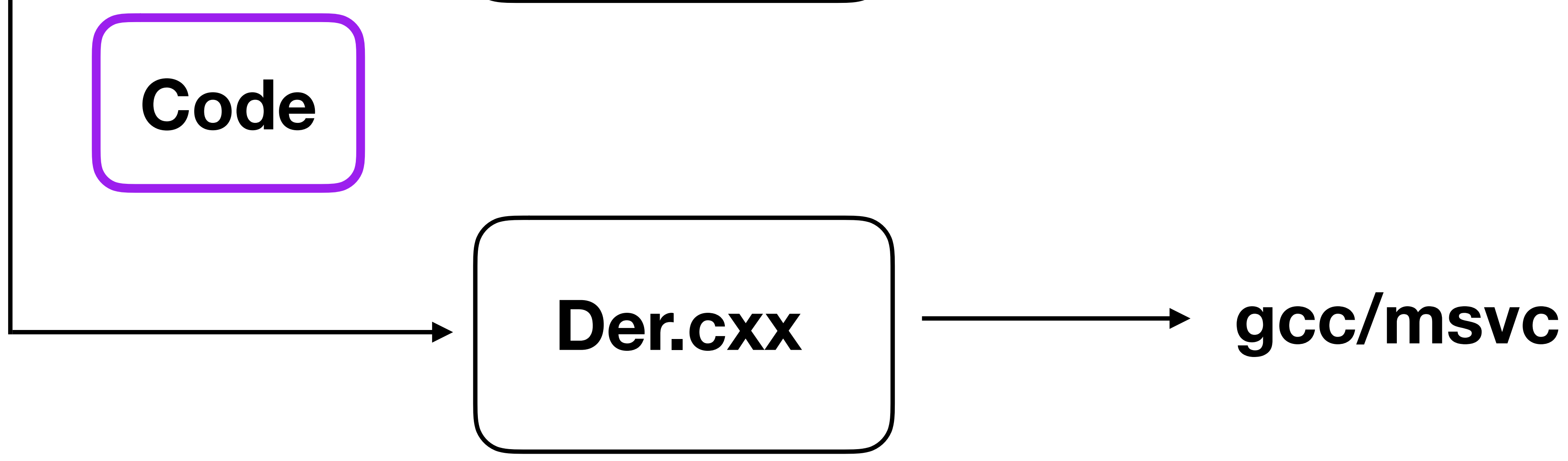
- Requires no code modification for computing derivatives of existing codebase
- Features both reverse mode AD (backpropagation) and forward mode AD
- Computes derivatives of functions, member functions, functors and lambda expressions
- Supports a large subset of C++ including if statements, for, while loops
- Provides direct functions for the computation of Hessian and Jacobian matrices
- Supports array differentiation, that is, it can differentiate either with respect to whole arrays or particular indices of the array
- Features numerical differentiation support, to be used where automatic differentiation is not feasible

https://clad.readthedocs.io / https://github.com/vgvassilev/clad

Clang Compilation Pipeline. Clad

Clad is a Clang Plugin transforming the AST of the supported languages : C++, CUDA, C, ObjC





Clad can:

- Produce the **AST** and pipe it to the backend
- Decompile that AST into code - which is code that you can compile with any other compilation pipeline (gcc/ msvc/ etc), then use it by plugging it in your library

```cpp
double f(double x) {
  return x * x;
}
```

```
FunctionDecl f 'double (double)'
|-ParmVarDecl x 'double'
`-CompoundStmt
  `-ReturnStmt
    `-BinaryOperator 'double' '*'
      |-ImplicitCastExpr 'double' <LValueToRValue>
      | `-DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
      `-ImplicitCastExpr 'double' <LValueToRValue>
        `-DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
```

```
FunctionDecl 0x7f7f801dbff8 <<invalid sloc>> <invalid sloc> f_darg0 'double (double)'
|-ParmVarDecl 0x7f7f801dc090 <<invalid sloc>> <invalid sloc> used x 'double'
`-CompoundStmt 0x7f7f801dc3d0 <<invalid sloc>>
  |-DeclStmt 0x7f7f801dc190 <<invalid sloc>>
  | `-VarDecl 0x7f7f801dc118 <<invalid sloc>> <invalid sloc> used _d_x 'double' cinit
  |   `-ImplicitCastExpr 0x7f7f801dc178 <<invalid sloc>> 'double' <IntegralToFloating>
  |     `-IntegerLiteral 0x7f7f801dc0f8 <<invalid sloc>> 'int' 1
  `-ReturnStmt 0x7f7f801dc398 <<invalid sloc>>
    `-BinaryOperator 0x7f7f801dc318 <<invalid sloc>> 'double' '+'
      |-BinaryOperator 0x7f7f801dc298 <<invalid sloc>, T.cpp:3:32> 'double' '*'
      | |-ImplicitCastExpr 0x7f7f801dc268 <<invalid sloc>> 'double' <LValueToRValue>
      | | `-DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var
0x7f7f801dc118 '_d_x' 'double'
      | `-ImplicitCastExpr 0x7f7f801dc280 <col:32> 'double' <LValueToRValue>
      |   `-DeclRefExpr 0x7f7f801dc208 <col:32> 'double' lvalue ParmVar 0x7f7f801dc090
'x' 'double'
      `-BinaryOperator 0x7f7f801dc2f0 <col:30, <invalid sloc>> 'double' '*'
        |-ImplicitCastExpr 0x7f7f801dc2c0 <col:30> 'double' <LValueToRValue>
        | `-DeclRefExpr 0x7f7f801dc1d0 <col:30> 'double' lvalue ParmVar 0x7f7f801dc090
'x' 'double'
        `-ImplicitCastExpr 0x7f7f801dc2d8 <<invalid sloc>> 'double' <LValueToRValue>
          `-DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var
0x7f7f801dc118 '_d_x' 'double'
```

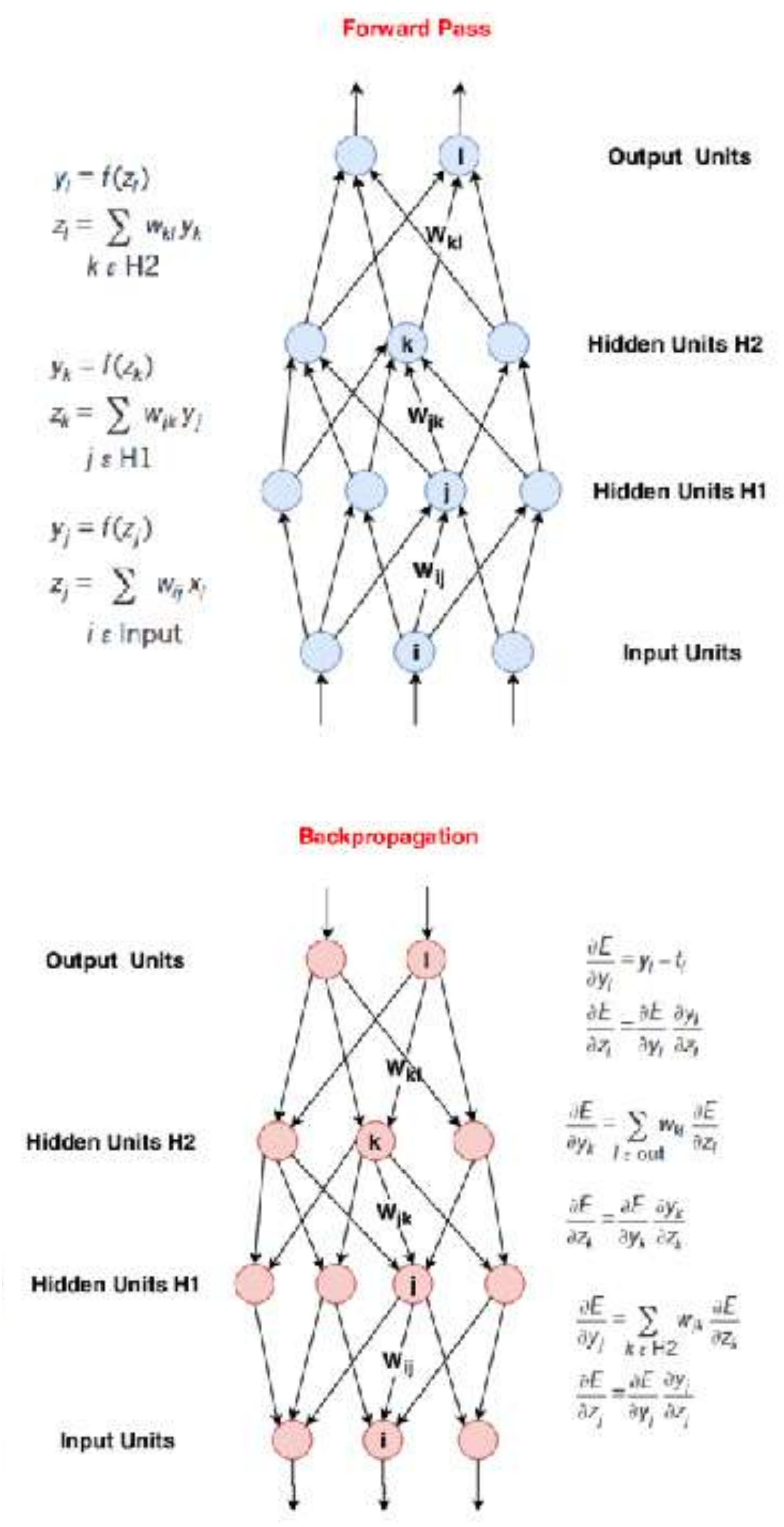
```cpp
double f_darg0(double x) {
  double _d_x = 1;
  return _d_x * x + x * _d_x;
}
```

Clad Features Showcase

Forward Mode

```
double f(double x, double y) {
  return x * y;
}

int main() {

  auto f_dx = clad::differentiate(f, “x");

  f_dx.dump();

  /* prints:
  double f_darg0(double x,
                 double y) {
      double _d_x = 1;
      double _d_y = 0;
      return _d_x * y + x * _d_y;
  } */
}
```

The independent parameter can be specified either using the parameter name or the parameter index; `d_fn_1.execute` returns the computed derivative.



Deep Multi Layer Neural Network Forward/Reverse Pass

Reverse Mode

```
double fn(double x, double y) {
  return x*x + y*y;
}

int main() {

  auto d_fn_2 = clad::gradient(fn, "x, y");

  d_fn_2.dump();

  /* prints:
  void fn_grad(double x, double y, clad::array_ref<double> _d_x,
               clad::array_ref<double> _d_y) {
      double _t2 = x, t3 = x, _t4 = y, _t5 = y;
      double fn_return = _t3 * _t2 + _t5 * _t4;
      goto _label0;
    _label0: {
          double _r0 = 1 * _t2;
          * _d_x += _r0;
          double _r1 = _t3 * 1;
          * _d_x += _r1;
          double _r2 = 1 * _t4;
          * _d_y += _r2;
          double _r3 = _t5 * 1;
          * _d_y += _r3;
    }
  } */
}
```

If no parameter is specified, then the function is differentiated w.r.t all the parameters

Clad Features Showcase

Hessian

```
#include “clad/Differentiator/Differentiator.h"
#include <iostream>

double kinetic_energy(double mass, double velocity) {
  return mass * velocity * velocity * 0.5;
}


int main() {

  // Can manually specify independent arguments
  auto hessian = clad::hessian(kinetic_energy, "mass, velocity”);

  // Creates an empty matrix to store the Hessian in
  // 2 independent variables require 4 elements (2^2=4)
  double matrix[4];

  // Substitutes these values into the Hessian function
  // pipes the result into the matrix variable.
  hessian.execute(10, 2, matrix);

  std::cout<<"Hessian matrix:\n";
  for (int i=0; i<2; ++i) {
    for (int j=0; j<2; ++j) {
      std::cout<<matrix[i*2 + j]<<" ";
    }
    std::cout<<“\n";
  }
}
```

Jacobian

```
#include “clad/Differentiator/Differentiator.h"
#include <iostream>

void fn(double i, double j, double *res) {
  res[0] = i*i;
  res[1] = j*j;
  res[2] = i*j;
}

int main() {


  auto d_fn = clad::jacobian(fn);
  double res[3] = {0, 0, 0};

  // Creates a matrix to store the Jacobian in
  // It will store in this case 6 derivatives
  double matrix[6] = {0, 0, 0, 0, 0, 0};

  // Substitutes these values into the Jacobian function
  // pipes the result into the derivatives variable.
  d_fn.execute(3, 5, res, matrix);

  std::cout<<"Jacobian matrix:\n";
  for (int i=0; i<3; ++i) {
    for (int j=0; j<2; ++j) {
      std::cout<<matrix[i*2 + j]<<" ";
    }
    std::cout<<“\n";
  }
}
```

Both support differentiating w.r.t multiple parameters. Moreover, in both cases, the array which will store the computed Hessian or Jacobian matrix should be passed as the last argument to the call to CladFunction::execute.

Newly Supported C++ Constructs

Functors

- functor objects are stateful
- can be used to create configurable algorithms
- calls to functor objects are often inlined by compilers - better performance

```
#include "clad/Differentiator/Differentiator.h"

// A class type with user-defined call operator
class Equation {
  double m_x, m_y;

  public:
  Equation(double x, double y) : m_x(x), m_y(y) {}
  double operator()(double i, double j) {
    return m_x*i*j + m_y*i*j;
  }
  void setX(double x) {
    m_x = x;
  }
};
```

```
Equation E(3,5);

// differentiate `E` wrt parameter `i`
// `E` is saved in the `CladFunction` object `d_E`

auto d_E = clad::differentiate(E, "i");

// differentiate `E` wrt parameter `i`
// `E` is saved in the `CladFunction` object `d_E_ptr`
auto d_E_ptr = clad::differentiate(&E, "i");
```

Differentiating functor objects in Clad
(GSoC 2021- Parth Arora)

Newly Supported C++ Constructs

Lambda Expressions

- defining an anonymous function object (a *closure*) at the location where it's invoked or passed as an argument to a function

```
#include "clad/Differentiator/Differentiator.h"

auto momentum = [](double mass, double velocity)
{
  return mass * velocity;
};
```

```
//both ways are equivalent
auto d_momentum = clad::differentiate(&momentum, “velocity”);
auto d_momentumRef = clad::differentiate(momentum, “velocity");

//compute derivatives wrt ‘velocity’ when (mass, velocity) = (5,7)
std::cout<<d_momentum.execute(5, 7)<< “\n”;

auto d_momentumGrad = clad::gradient(&momentum);
double d_mass=0, d_velocity=0;

//compute derivatives wrt ‘mass’ and ‘velocity’
//given (mass, velocity) = (5,7)

d_momentumGrad.execute(5, 7, &d_mass, &d_velocity);
std::cout<<d_mass<<“ “<<d_velocity<< “\n”;
```

Differentiating functor objects in Clad (GSoC 2021- Parth Arora)

Clad New CUDA Support

```
__device__ __host__ double gauss(double* x, double* p,
                                  double sigma, int dim) {
    double t = 0;
    for (int i = 0; i< dim; i++)
        t += (x[i] - p[i]) * (x[i] - p[i]);
    t = -t / (2*sigma*sigma);
    return std::pow(2*M_PI, -dim/2.0) * std::pow(sigma, -0.5) * std::exp(t);
}
```

```
auto gauss_g = clad::gradient(gauss);
```

```
void gauss_grad(double* x, double* p, double sigma, int dim,
           clad::array_ref<double> _d_x, clad::array_ref<double> _d_p,
           clad::array_ref<double> _d_sigma, clad::array_ref<double> _d_dim)
           __attribute__((device)) __attribute__((host)) {
    double _d_t = 0;
    unsigned long _t2;
    int _d_i = 0;
    clad::tape<double> _t3 = {};
    clad::tape<int> _t4 = {};
    ..............................
    for (; _t2; _t2--) {
        double _r_d0 = _d_t;
        _d_t += _r_d0;
        double _r0 = _r_d0 * clad::pop(_t3);
        _d_x[clad::pop(_t4)] += _r0;
        _d_p[clad::pop(_t5)] += -_r0;
        double _r1 = clad::pop(_t6) * _r_d0;
        _d_x[clad::pop(_t7)] += _r1;
        _d_p[clad::pop(_t8)] += -_r1;
        _d_t -= _r_d0;
    }
}
```

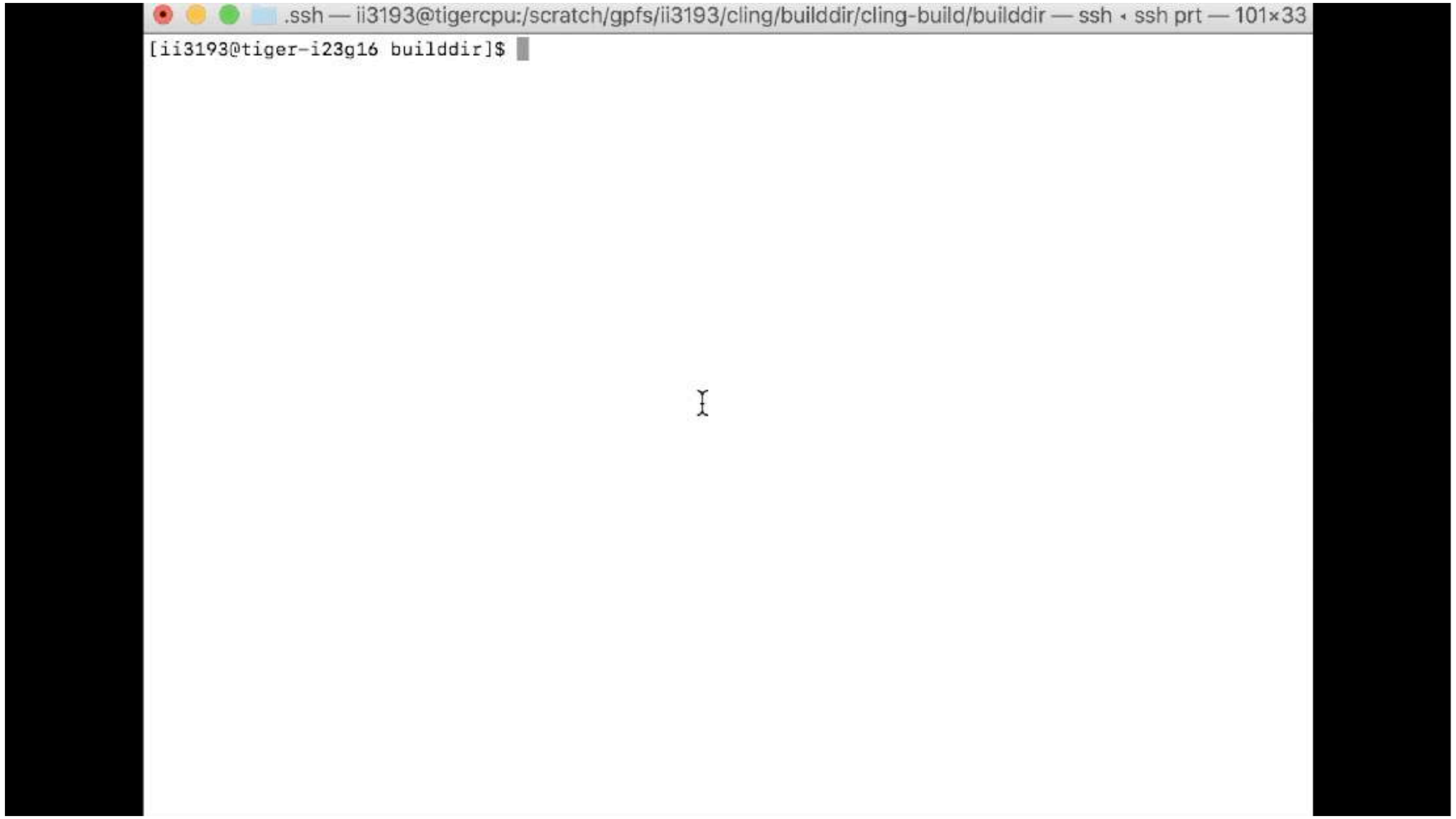
Clad can compute the gradient of host/ device functions

CUDA computation kernels can now call Clad defined derivatives

Currently working on:

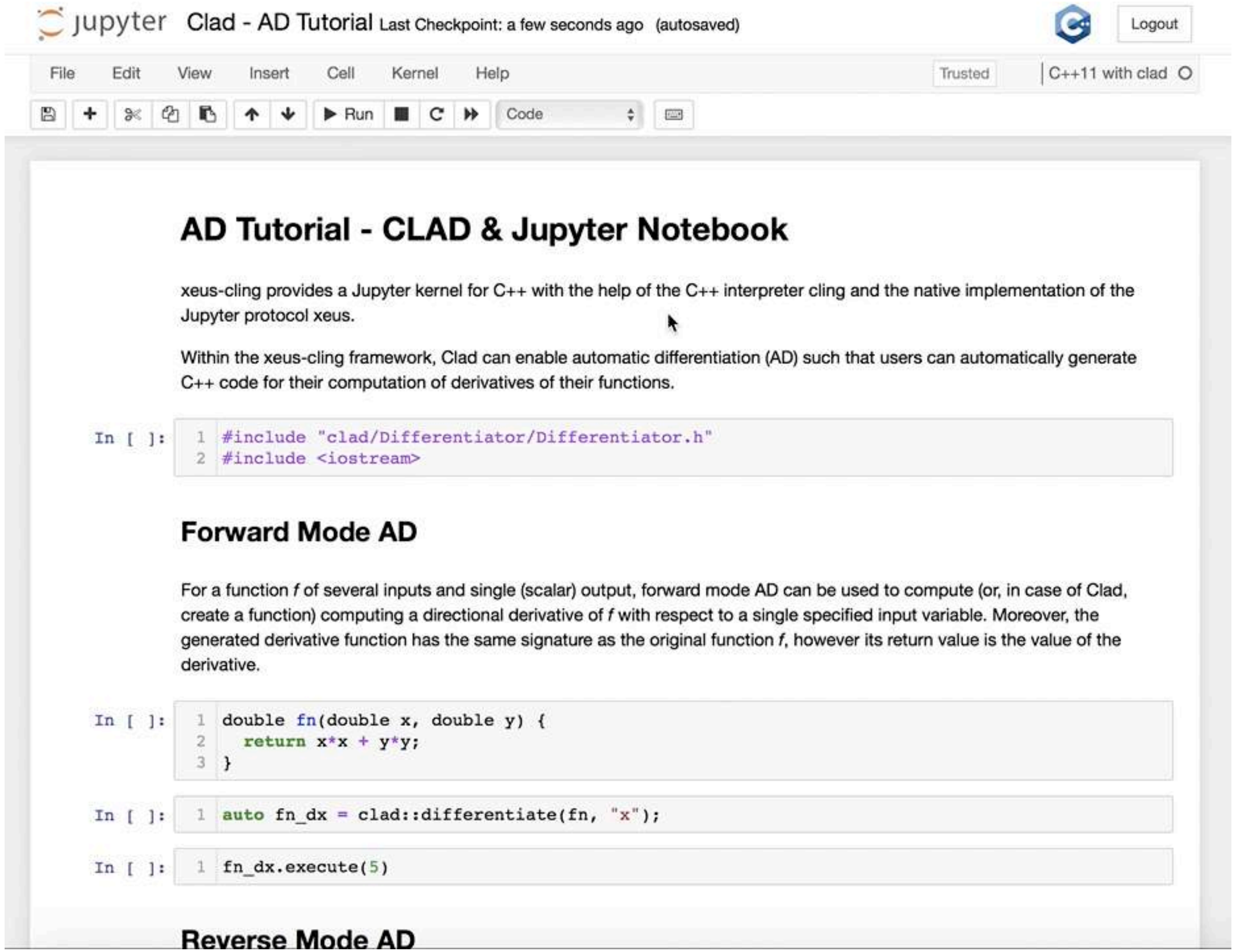
- **enabling automatic offloading of gradient computations to GPU**
- **differentiating CUDA kernels**

Clad & CUDA as a Service



The demo shows cling usage of clad as a plugin to produce a derivative on the fly and send it to a CUDA kernel for execution

Clad as a Service



Usage of CLAD within the Jupyter Notebook with the help of “xeus-cling” (a Jupyter kernel for C++ based on the C++ interpreter cling)

Clad integration in ROOT

ROOT is a data analysis software package used to process data in the field of high-energy physics.

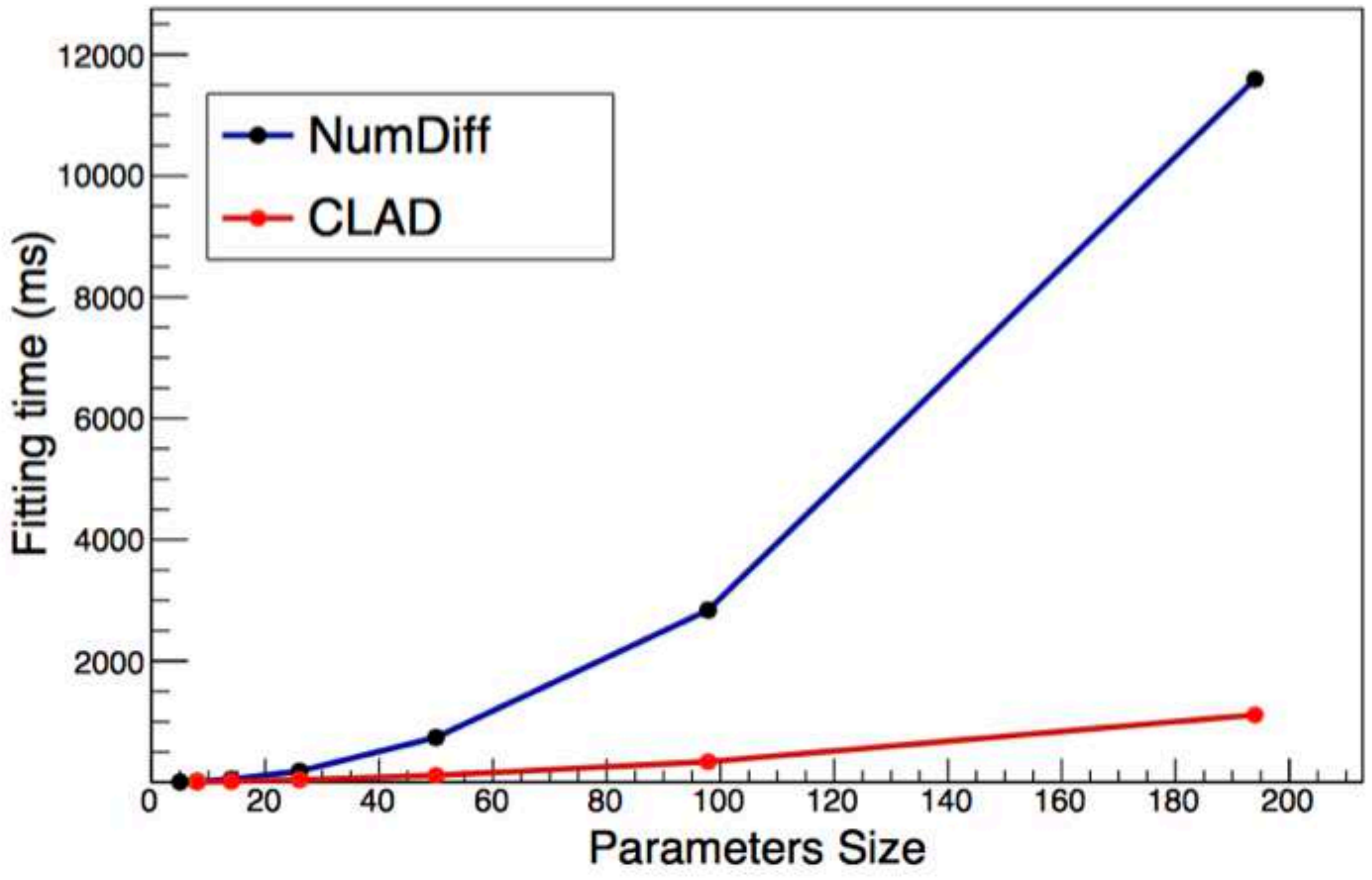
Clad has replaced numerical gradient calculations for formula based functions.

The Clad gradient is then used to compute the gradient of the objective function (χ^2 or negative log-likelihood function) when fitting

$$\chi^2 = \sum_{i=1}^{N} \frac{\left(Y_i - f(x_i, \mathbf{p})\right)^2}{\sigma_i^2}$$

Thus, ROOT fitting class computes $\nabla_{\mathbf{p}}(\chi^2)$ from $\nabla_{\mathbf{p}}(f(x, \mathbf{p}))$ obtained using Clad

Clad VS Numerical Differentiation of objective function



* current implementation still requires one numerical gradient call for second derivatives (when seeding) - higher speedups will be possible when introducing second derivatives computation using Clad

Clad Hessian Mode in ROOT (GSoC 2021- Baidyanath Kundu)

Summary

- Clad uses the source transformation approach by statically analysing the original code to produce a gradient function in the source code language

- Clad can produce the **AST** and pipe it to the backend as well as decompile that **AST** into code. Moreover, one can compile the produced code with any other preferred compilation pipeline (gcc/ msvc/etc), then plug it in one's library and use it

- Continuous effort is put into expanding the support subset of C++, such as support for differentiating *continue* and *break* statements

- The new CUDA support means generated Clad derivatives are now supported for computations on CUDA kernels thus allowing for further optimisation

- The performance results in ROOT show good improvement, however work is ongoing on a set of general benchmarks

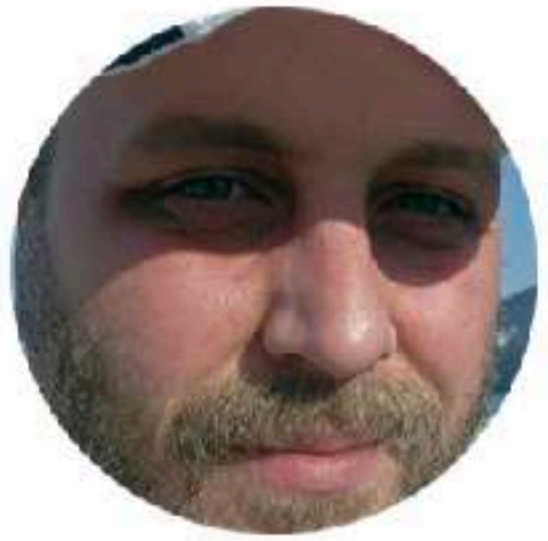
- Currently the scheduling procedure requires a certain degree of user input to make it suitable for a hybrid CPU/GPU setup. Our current aim is to fully automate this last step for complete CUDA integration, where the full toolchain process needs to be formalised with both scheduling optimisation and global memory constraints in mind

People

Violeta Ilieva
Initial prototype, Forward Mode
GSoC

Vassil Vassilev
Conception, Mentoring, Bugs, Integration, Infrastructure

Martin Vassilev
Forward Mode, CodeGen
GSoC



Alexander Penev
Conception, CMake, Demos

Aleksandr Efremov
Reverse Mode

Jack Qui
Hessians
GSoC

Roman Shakhov
Jacobians
GSoC

Oksana Shadura
Infrastructure, Co-mentoring

Pratyush Das
Infrastructure

Garima Singh
FP error estimation, Bugs
IRIS-HEP Fellow

Ioana Ifrim
CUDA AD

Thank you!