Clad — Automatic Differentiation for C++ Using Clang

Vassil Vassilev, Princeton University
Motivation

Provide automatic differentiation for C/C++ that works without code modification (including legacy code)
Clang Compilation Pipeline. Clad

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

- Supports derivatives (partial and higher order), gradients, hessians and jacobians.
- Provides low-level derivative access primitives
- Allows embedding in frameworks

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

```cpp
// clang -Xclang -add-plugin -Xclang clad -Xclang -load
// -Xclang libclad.so ...
// Necessary for clad to work include
#include "clad/Differentiator/Differentiator.h"

double pow2(double x) { return x * x; }

double pow2_darg0(double);

int main() {
  auto dfdx = clad::differentiate(pow2, 0);

  // Function execution can happen in 3 ways:
  // 1) Using CladFunction::execute method.
  double res = cladPow2.execute(1);

  // 2) Using the function pointer.
  auto dfdxFnPtr = cladPow2.getFunctionPtr();
  res = cladPow2FnPtr(2);

  // 3) Using direct function access through fwd declaration.
  res = pow2_darg0(3);
  return 0;
}
```

The body will be generated by clad.

Result via the clad function-like wrapper object

Result via function pointer call

Result via a forward declaration
Clad. Usage. CUDA

Work in progress by Ioana Ifrim

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

typedef double(*func) (double);

__device__ __host__ double pow2(double x) {
    return x * x;
}

__device__ __host__
double pow2_darg0(double x);

__device__ __host__
double pow2_darg0(double x);

__device__ func p_pow2 = pow2_darg0;

__global__ func p_pow2 = pow2_darg0;

int main(void) {
    double x = 5.0;
    func h_pow2;
    double * d_x;
    double result;
    double * d_result, * h_result;
    cudaMalloc(&d_x, sizeof(double));
    cudaMemcpy(d_x, &x, sizeof(double), cudaMemcpyHostToDevice);
    h_result = &result;
    clad::differentiate(pow2, "x");
    cudaMemcpyFromSymbol(&h_pow2, p_pow2, sizeof(func));
    compute<<<1,1>>>(h_pow2, d_x, d_result);
    cudaMemcpy(h_result, d_result, sizeof(double), cudaMemcpyDeviceToHost);
    printf("Result: %f\n", result);
}
```

Clad will provide a definition

Static pointer to the device function

Copy device function pointer of differentiated function to host

Memory allocation

Device Sync
People

Violeta Ilieva
Initial prototype, Forward Mode
GSoC

Vassil Vassilev
Conception, Mentoring, Bugs, Integration, Infrastructure, US NSF

Martin Vassilev
Forward Mode,CodeGen GSoC

Alexander Penev
Conception, CMake, Demos

Aleksandr Efremov
Reverse Mode, US NSF

Jack Qui
Hessians GSoC

Roman Shakhov
Jacobians GSoC

Oksana Shadura
Infrastructure, Co-mentoring, US NSF

Pratyush Das
Infrastructure

Garima Singh
FP error estimation, Bugs IRIS-HEP Fellow

Ioana Ifrim
CUDA AD, US NSF
Integration of Clad in Cling and ROOT

Some domains benefit from supporting derivatives of user code. Interpretative languages such as python can provide that easily.

C++ has **cling** — an interactive, llvm-based C++ interpreter.

**ROOT** is a data analysis software package used to process data in the field of high-energy physics.
Use of Clad on the fly

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.

The demo shows cling use clad as a plugin to produce a derivative on the fly
Clad as a Service

• A service capable of running AD on a given code at program’s runtime

• Runs embedded in your framework code with your favorite compiler
# Derivatives as a service.

```cpp
#include <cling/Interpreter/Interpreter.h>
#include <cling/Interpreter/Value.h>

// Derivatives as a service.

void gimme_pow2dx(cling::Interpreter& interp) {
  // Definitions of declarations injected also into cling.
  interp.declare("double pow2(double x) { return x*x; }");
  interp.declare("#include <clad/Differentiator/Differentiator.h>");
  interp.declare("auto dfdx = clad::differentiate(pow2, 0); ");

  cling::Value res; // Will hold the evaluation result.
  interp.process("dfdx.getFunctionPtr()");, &res);

  using func_t = double(double);
  func_t* pFunc = res.getAs<func_t*>();
  printf("dfdx at 1 = %f\n", pFunc(1));
}

int main(int argc, const char* const* argv) {
  std::vector<const char*> argvExt(argv, argv+argc);
  argvExt.push_back("-fplugin=etc/cling/plugins/lib/clad.dylib");
  // Create cling. LLVMDIR is provided as -D during compilation.
  cling::Interpreter interp(argvExt.size(), &argvExt[0], LLVMDIR);
  gimme_pow2dx(interp);
  return 0;
}
```

Result from running the clad-demo binary:

```
./clad-demo
dfx at 1 = 2.000000
```
Benchmarks
double* Numerical(double* p, int dim, double eps = 1e-8) {
    double* result = new double[dim]{}
    for (int i = 0; i < dim; i++) {
        double pi = p[i];
        p[i] = pi + eps;
        double v1 = sum(p, dim);
        p[i] = pi - eps;
        double v2 = sum(p, dim);
        result[i] = (v1 - v2)/(2 * eps);
        p[i] = pi;
    }
    return result;
}

double sum(double* p, int dim) {
    double r = 0.0;
    for (int i = 0; i < dim; i++)
        r += p[i];
    return r;
}

double Clad(double* p, int dim) {
    auto result = new double[dim]{}
    auto sum_grad = clad::gradient(sum, "p");
    sum_grad.execute(p, dim, result);
    return result;
}

Efremov, Clad: the automatic differentiation plugin for Clang, verified by Ioana Ifrim in 2021!
General Benchmarks

```cpp
double* Numerical(double* p, int dim, double eps = 1e-8) {
    double* result = new double[dim]{};
    for (int i = 0; i < dim; i++) {
        double pi = p[i];
        p[i] = pi + eps;
        double v1 = sum(p, dim);
        p[i] = pi - eps;
        double v2 = sum(p, dim);
        result[i] = (v1 - v2)/(2 * eps);
        p[i] = pi;
    }
    return result;
}
```

```cpp
double Clad(double* p, int dim) {
    auto result = new double[dim]{};
    auto sum_grad = clad::gradient(sum, "p");
    sum_grad.execute(p, dim, result);
    return result;
}
```

```cpp
double gaus(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);
}
```

**AD using Clad**

**Numerical differentiation based on the central differences**

**Efremov, Clad: the automatic differentiation plugin for Clang, verified by Ioana Ifrim in 2021!**

**900x speedup**
Integration in ROOT
Integration in ROOT TF1

• Use Clad to generate gradients for formula based functions

```cpp
TF1* f1 = new TF1("f1", ":[a]*x^2+[b]*x+[c]";
// tell TFormula to generate gradient function
f1->GetFormula()->GenerateGradientPar();
// compute gradient using CLAD
std::vector<double> gradient(3);
f1->GradientPar(x.data(), gradient.data());
```

• When Clad is not available use numerical differentiation
Clad in ROOT

TFormula is a ROOT class which bridges compiled and interpreted code.

Only 10x speedup, the optimizer needs the hessian.

Work by Lorenzo Moneta
Future

- Clad works for the cases we tested, however, there are a lot of unexplored codes.
- We need a gradual adoption to allow improving the implementation if necessary.
- Work on standardization of AD in C++ [https://wg21.link/P2072](https://wg21.link/P2072)
- Full support of arrays
- Develop error estimation framework — at compile time and at runtime
- Retarget code on GPGPU — OpenCL and/or CUDA
- Support functor objects
- Make the derivation process explicitly configurable
- Integration with Enzyme
Thank you.

https://github.com/vgvassilev/clad
void sum_grad_0(double *p, int dim, double *_result) {
    double _d_r = 0;
    unsigned long _t0;
    int _d_i = 0;
    clad::tape<int> _t1 = {};
    double r = 0.;
    _t0 = 0;
    for (int i = 0; i < dim; i++) {
        _t0++;
        r += p[clad::push(_t1, i)];
    }
    double sum_return = r;
    goto _label0;
_label0:
    double _d_r = 1;
    for (; _t0; _t0--) {
        double _r_d0 = _d_r;
        _d_r += _r_d0;
        _result[clad::pop(_t1)] += _r_d0;
        _d_r -= _r_d0;
    }
}