GPU Acceleration of Automatic Differentiation in C++ with Clad

Ioana Ifrim, Princeton University
Content

1. Motivation

2. Automatic Differentiation and applications - ML Case Study

3. Clad.AD Plugin for Clang

4. C++ Compilation Pipeline

5. Clang Compilation Pipeline. Clad

6. GPU Accelerated AD

7. Clad & CUDA as a Service

8. Summary

9. Future Steps
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 and Numerical differentiation (the method of finite differences) and has applications ranging from the Machine Learning areal of domains to High Energy Physics.

The aim of Clad is to provide automatic differentiation for C/C++ which works without code modification (including legacy code).

The range of automatic differentiation (AD) application problems are defined by their high computational requirements and thus can greatly benefit from parallel implementations on graphics processing units (GPUs).
Case Study: ML Application

In machine learning, we use gradient descent to update the parameters of our chosen model. A set of training inputs $x_i$ are fed forward into the model generating corresponding activations $y_i$. We define an error $E$ as the difference computed between the data target output $t$ and the model output $y_3$. The error adjoint is propagated backward, resulting in the gradient with respect to the weights:

$$\nabla_{w_i} E = \left( \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_6} \right)$$

This procedure is the central player of the gradient-descent optimisation algorithm. The required gradient is obtained by the backward propagation of the susceptibility of the objective value at the output, using the chain rule to compute partial derivatives of the objective wrt each of the weights. In this way, the resulting algorithm can be interpreted as transforming the network evaluation function composed with the objective function under reverse mode AD (generalisation of the back-propagation procedure).
Case Study : ML Application

Manual Differentiation

It was historically the case that Machine Learning researchers would dedicate considerable amounts of time to the process of manual derivation of analytical derivatives which in turn were used in gradient descent procedures.
Numerical Differentiation

Numerical differentiation is the finite difference approximation of derivatives using values of the original function evaluated at some sample points with $e_i$ being $i$-th unit vector and $h > 0$ is a small step size

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h}$$

The introduction of round off errors bring forth issues of consistency, convergence, and stability of the numerical solution.

Moreover, the $O(n)$ complexity of numerical differentiation for a gradient in $n$ dimensions is the main obstacle to its usefulness in machine learning, where $n$ can be as large as millions or billions in state-of-the-art deep learning models.
Symbolic Differentiation

Symbolic differentiation is the automatic manipulation of expressions for obtaining derivative expressions and it is carried out by applying transformations representing rules of differentiation such as

\[
\frac{d}{dx} (f(x) + g(x)) \rightarrow \frac{d}{dx} f(x) + \frac{d}{dx} g(x)
\]

\[
\frac{d}{dx} (f(x) g(x)) \rightarrow \left( \frac{d}{dx} f(x) \right) g(x) + f(x) \left( \frac{d}{dx} g(x) \right)
\]

Symbolic differentiation can easily produce exponentially large symbolic expressions which take correspondingly long times to evaluate. This problem is known as expression swell. Moreover, it may require transcribing result back into code.
Case Study: ML Application

Automatic Differentiation

Automatic generation of a C++ program able to compute the derivative of a given function

The procedure involves applying the chain rule of differential calculus throughout the semantics of the original program

Example Usage: Fitting a logistic regression model by minimising the binary cross-entropy loss of the logistic output

\[
\mathcal{L} \circ \sigma(X, y, \beta) = p \log \sigma(X\beta) + (1 - p) \log(1 - \sigma(X\beta))
\]

\[
\nabla_\beta \mathcal{L} = p \nabla_\beta \log a(h) + (1 - p) \nabla_\beta \log(1 - a(h))
\]

\[
= p \ diag \left( \frac{1}{a(h_i)} \right) \nabla_\beta a(h) + (1 - p) \ diag \left( \frac{1}{1 - a(h_i)} \right) \nabla_\beta (1 - a(h))
\]

\[
= p \ diag \left( \frac{1}{a(h_i)} \right) \nabla_h a(h) \nabla_\beta h(X, \beta)
\]

\[- (1 - p) \ diag \left( \frac{1}{1 - a(h_i)} \right) \nabla_h (1 - a(h)) \nabla_\beta h(X, \beta)\]
Case Study: ML Application

Automatic Differentiation - Forward Mode

In forward mode auto differentiation, we start from the left-most node and move forward along to the right-most node in the computational graph – a forward pass.

We calculate elementary derivatives using the expressions and leveraging the chain rule to obtain the intermediate derivatives at each step, obtaining the desired derivative with respect to the first variable. A forward pass is needed for each desired derivative – one derivative with respect to each of the n input parameters.

\[ h(X, \beta), \nabla_{\beta} h(X, \beta) \rightarrow a(h), \nabla_{h} a(h), \rightarrow \log a(h), \nabla_{a} \log a(h) \]

Derivative function created by the forward-mode AD is guaranteed to have at most a constant factor (around 2-3) more arithmetical operations compared to the original function.

\texttt{clad::differentiate()} takes 2 arguments:
1. \texttt{f} is a pointer to a function or a method to be differentiated
2. \texttt{ARGS} is either:
   - a single numerical literal indicating an index of independent variable (e.g., 0 for \(x\), 1 for \(y\))
   - a string literal with the name of independent variable (as stated in the definition of \(f\), e.g., \(x^t\) or \(y^t\))

Generated derivative function has the same signature as the original function \(f\), however its return value is the value of the derivative.
Automatic Differentiation - Reverse Mode

In reverse mode autodiff, we start from the outer-most node

Suppose that we are interested in calculating the log derivative with respect to a particular activation, we employ the chain rule

\[ h(X, \beta) \rightarrow a(h) \rightarrow \log a(h) \]

\[ \nabla \log a(h) \rightarrow \nabla a(h) \rightarrow \nabla h(X, \beta) \]

Machine learning tasks involve a large number of feature space parameters who are to be tuned, thus reverse mode AD fits perfectly the task of calculating the derivatives of the cost function wrt model parameters

```cpp
1 auto f_grad = clad::gradient(f);
2 double result1[2] = {0};
3 f_grad.execute(x, y, result1);
4 std::cout << "dx: " << result1[0] << ", " << dy: " << result1[1] << std::endl;
5 auto f_dx_dy = clad::gradient(f, "x, y"); // same effect as before
6 auto f_dy_dx = clad::gradient(f, "y, x");
7 double result2[2] = {0};
8 f_dy_dx.execute(x, y, result2);
9 // note that the derivatives are mapped to the "result" indices in the same order as they were specified in the argument:
10 std::cout << "dy: " << result2[0] << ", " << dx: " << result2[1] << std::endl;
```

Deep Multi Layer Neural Network Reverse Pass

Case Study: ML Application

Deep Multi Layer Neural Network Reverse Pass
Clad is a compiler plugin extending Clang able to produce derivatives in both forward and reverse mode:

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

V. Vassilev, L. Moneta
Automatic Differentiation in C++ with Clad, Integration in ROOT
94th ROOT.PP Meeting
Clang Compilation Pipeline. Clad

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

double f_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
Considering our Machine Learning case study, the goal is to compute the gradient of the cost function with respect to a transformation parameter vector $x$.

From an AD perspective, this can be done either by invoking the forward mode derivative once for every dimension in the parameters space or by a single pass of the reverse mode derivative.

These passes are bounded by access to and computations performed on the transformation parameters, hence this process is an excellent candidate for acceleration through GPU support implementation.

Tasks featuring heavy computations increase their time consumption proportional with the data sets magnitude. These applications can thus profit from the usage of threads and in this sense GPU acceleration brings a new layer of optimisation and a proportional speed up.
The CUDA support for Clad includes extensions that allow one to execute functions on the GPU using many threads in parallel.

Function attributes cloning has been introduced for __device__ __host__ to be carried forward in the Clad gradient definition

Custom derivatives were extended to include __device__ __host__ declaration as well as previous dependencies on the standard library functionalities not supported by CUDA, were reimplemented. (falling back on Thrust, the template library for CUDA based on the Standard Template Library (STL) has proved to be an issue in the context of Clang compilation)

Clad uses Tape Records for the execution that is replayed such that the gradient is produced in one pass - this also required extensions for the CUDA context and removal of dependencies on the standard library.
GPU Accelerated AD

- Benchmark showcases how using CUDA can influence the overall AD performance in computation of a gauss gradient with different dimensions

- GPU : Tesla P100-PCIE-16GB
- CPU : Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz

More benchmarks coming soon *
GPU Accelerated AD

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

#define M 10485760

typedef void (*)(double *x, double *p, double *sigma, int dim, double *result);

__device__ __host__ 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]);
    return std::sqrt(2*M_PI/dim*2.0) * std::sqrt(pow(t, -0.5)) * std::exp(t);
}

__device__ __host__ void gaus_grad(double *x, double *p, double *sigma, int dim, double *result);

auto gaus = clad::gradient(gaus);

__device__ func p_pow2 = sun_grad;

globfunc compute(func op, double *d_x, int n, double *result) {
    int i = threadIdx.x;
    if (i < n)
        (op)(d_x[i], d_x[i] * 2.0, &result[i]);
}

int main(void) {
    __double__ __host__ double x;
    x = (double*)malloc(N*sizeof(double));
    for (int i = 0; i < N; i++) {
        x[i] = 2.0;
    }
    func h_pow2;
    cudaMalloc((void**)&h_pow2, N*sizeof(double));
    cudaMemcpy(h_pow2, f_pow2, N*sizeof(func));
    cudaDeviceSynchronize();

    cudaMalloc((void**)&h_result, N*sizeof(double));
    __double__ __host__ double result = (double*)malloc(N*sizeof(double));
    cudaMemcpy(result, d_result, N*sizeof(double));
    cudaDeviceSynchronize();
    cudaMemcpy(result, d_result, N*sizeof(double), cudaMemcpyDeviceToHost);
}
```
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.
Summary

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

- Scheduling still requires a certain degree of user input which we aim to further automate.

- Clad can now handle a hybrid GPU/CPU setup, where the generation is currently done on the CPU, while the execution can be parallelised on GPUs.

- Challenges in terms of:
  - CUDA version Clang & Cling compatibility can be observed in implementation choices (e.g. not using Thrust (C++ template library for CUDA based on the Standard Template Library (STL)) due to compatibility issues with Clang) (fixed)
  - Passing the gradient function by pointer when compiling with Clang (fixed) / Cling (wip)
Future Steps

- Full support of arrays

- Enable AD support for second order derivatives for HEP analysis through ROOT (data analysis software package) via Clad: GSoC Project

- In the interactive case - provide an API which synthesises the CUDA kernel automatically and alleviates the user’s need to deal with memory optimisation

- Currently the scheduling procedure requires a certain degree of user input to make it suitable for a hybrid CPU/GPU setup. A future 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
Thank you!