GPU Acceleration of Automatic Differentiation in C++ with Clad

Ioana Ifrim, Princeton University

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 and Numerical differentiation (the method of finite differences) and has applications ranging form 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)

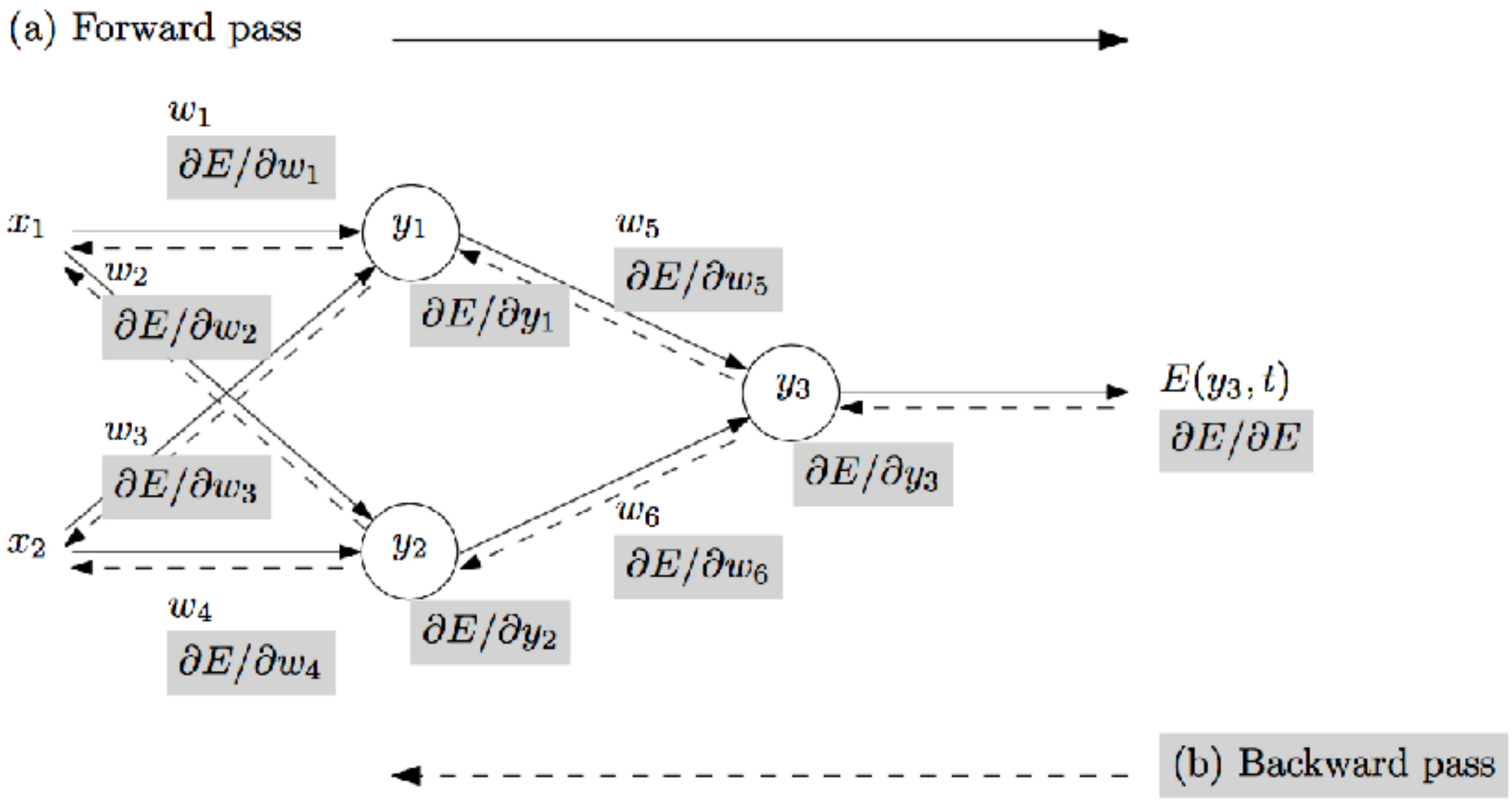
The range of automatic differentiation (AD) application problems are defined by their high computational requirements and thus can greatly benefit from parallel implementations on graphics processing units (GPUs).

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.

Differentiation Methods

Case Study : ML Application

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

$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}$$

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

Case Study : ML Application

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

$$\frac{d}{dx}\left(f(x)+g(x)\right) \rightsquigarrow \frac{d}{dx}f(x)+\frac{d}{dx}g(x)$$

$$\frac{d}{dx}\left(f(x)\,g(x)\right) \rightsquigarrow \left(\frac{d}{dx}f(x)\right)g(x)+f(x)\left(\frac{d}{dx}g(x)\right)$$

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

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

$$\begin{aligned}
\nabla_{\beta}\mathcal{L} =& p\nabla_{\beta} \log a(h) + (1 - p)\nabla_{\beta} \log(1 - a(h)) \\
=& p\ diag\left(\frac{1}{a(h_i)}\right) \nabla_{\beta} a(h) + (1 - p)\ diag\left(\frac{1}{1 - a(h_i)}\right) \nabla_{\beta}(1 - a(h))) \\
=& p\ diag\left(\frac{1}{a(h_i)}\right) \nabla_{h} a(h) \nabla_{\beta} h(X, \beta) \\
& - (1 - p)\ diag\left(\frac{1}{1 - a(h_i)}\right) \nabla_{h}(1 - a(h)) \nabla_{\beta} h(X, \beta)
\end{aligned}$$

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.

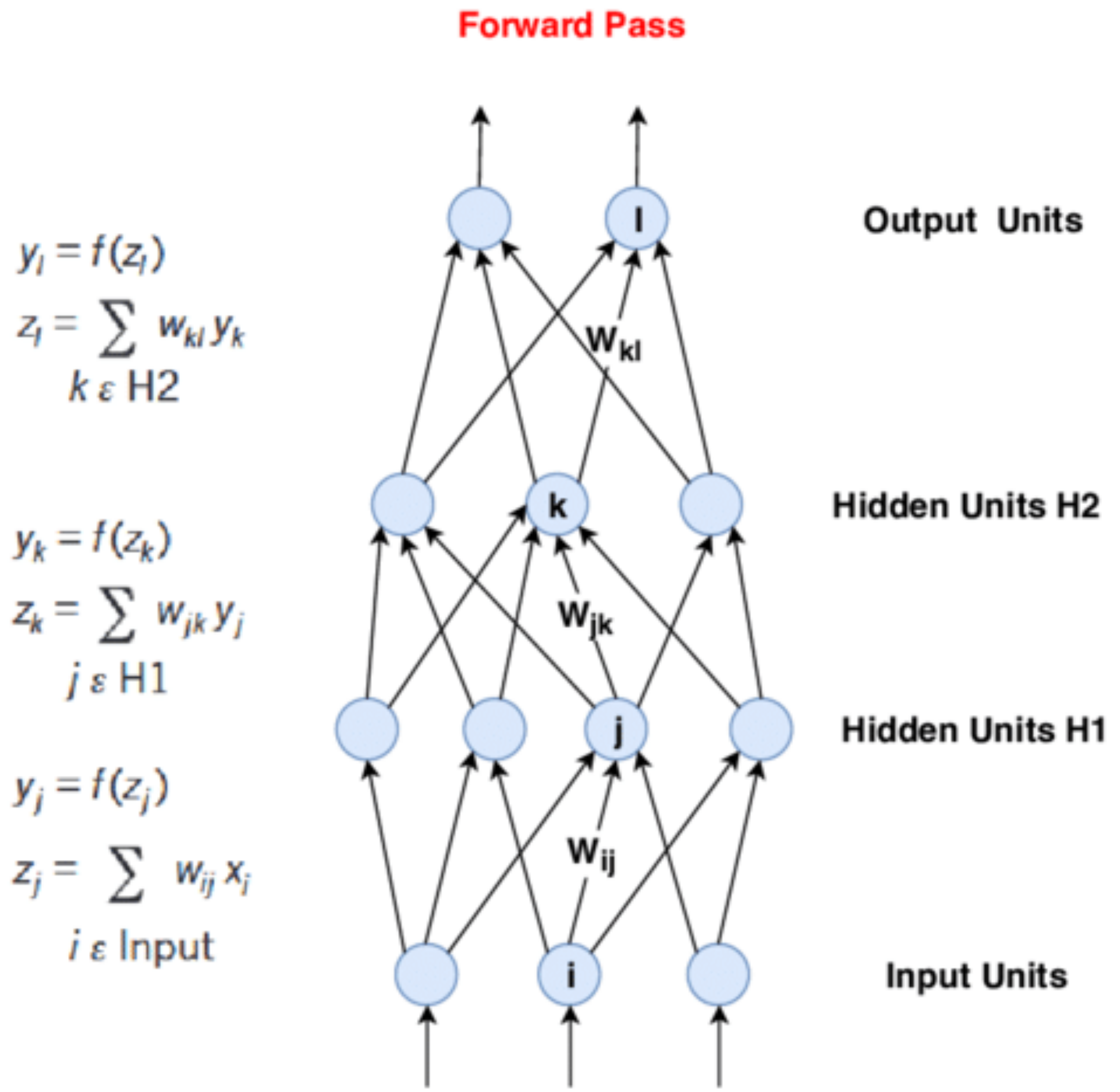
$$h(X,\beta), \nabla_{\beta} h(X,\beta) \rightarrow a(h), \nabla_{h} a(h), \rightarrow \log a(h), \nabla_{a} \log a(h)$$

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.
`clad::differentiate(f, ARGS)` takes 2 arguments:

1. `f` is a pointer to a function or a method to be differentiated
2. `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"` or `"y"`)

Generated derivative function has the same signature as the original function `f`, however its return value is the value of the derivative.



Deep Multi Layer Neural Network Forward Pass

Case Study : ML Application

Automatic Differentiation - Reverse Mode

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

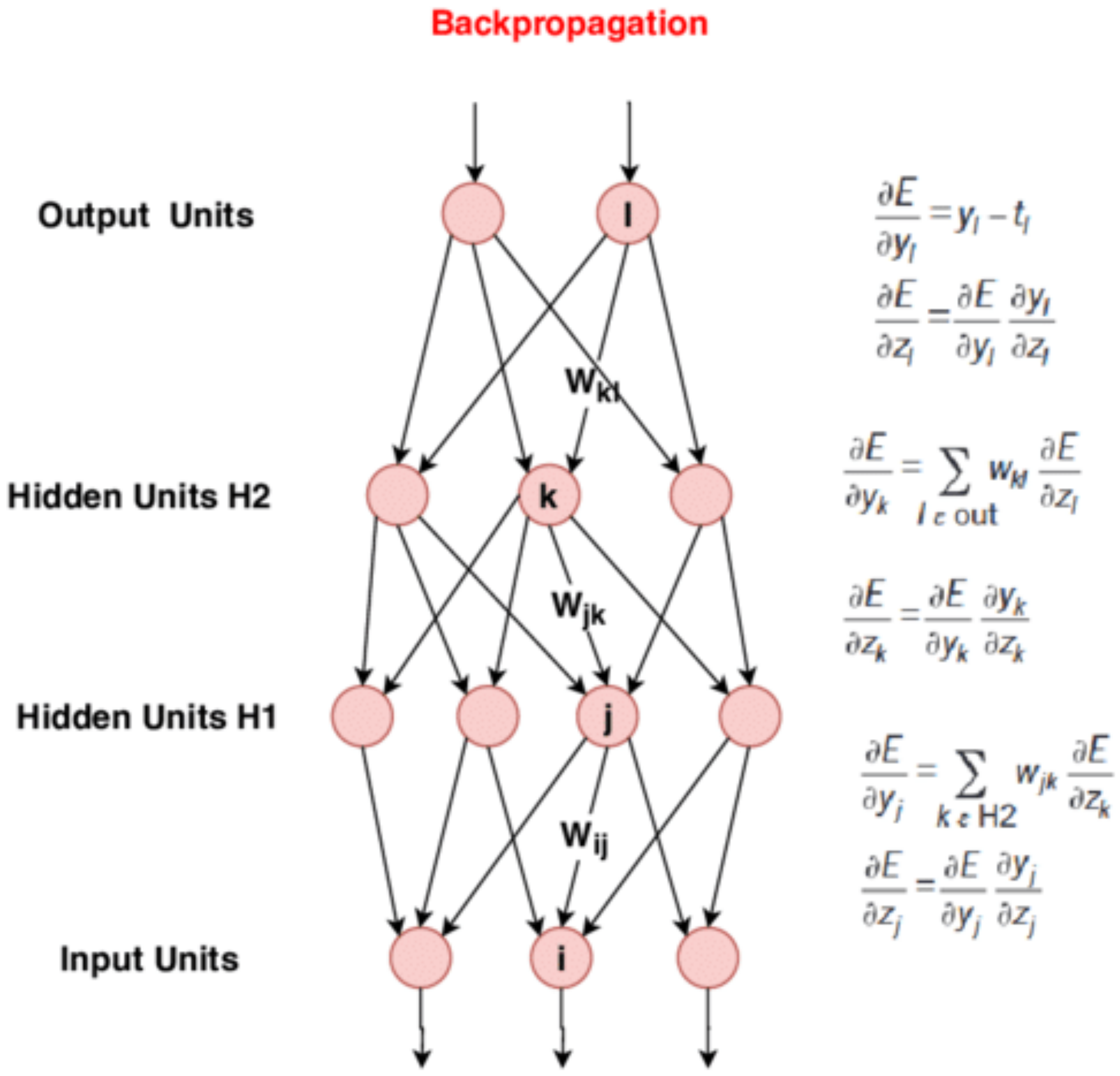
Suppose that we are interested in calculating the log derivative with respect to a particular activation, we employ the chain rule

$$h(X, \beta) \rightarrow a(h) \rightarrow \log a(h)$$

$$\nabla \log a(h) \rightarrow \nabla a(h) \rightarrow \nabla h(X, \beta)$$

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
auto f_grad = clad::gradient(f);
double result1[2] = {};
f_grad.execute(x, y, result1);
std::cout << "dx: " << result1[0] << ' ' << "dy: " << result1[1] << std::endl;

auto f_dx_dy = clad::gradient(f, "x, y"); // same effect as before

auto f_dy_dx = clad::gradient(f, "y, x");
double result2[2] = {};
f_dy_dx.execute(x, y, result2);
// note that the derivatives are mapped to the "result" indices in the same order as they were specified in the argument:
std::cout << "dy: " << result2[0] << ' ' << "dx: " << result2[1] << std::endl;
```



$$\frac{\partial E}{\partial y_l} = y_l - t_l$$

$$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l} \frac{\partial y_l}{\partial z_l}$$

$$\frac{\partial E}{\partial y_k} = \sum_{l \in out} w_{kl} \frac{\partial E}{\partial z_l}$$

$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_k}$$

$$\frac{\partial E}{\partial y_j} = \sum_{k \in H2} w_{jk} \frac{\partial E}{\partial z_k}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j}$$

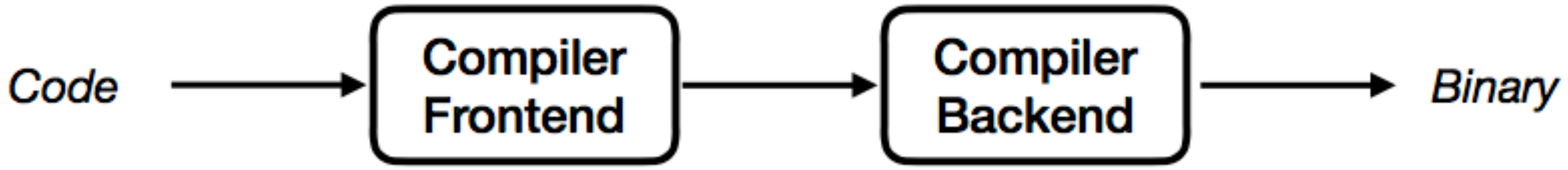
Deep Multi Layer Neural Network Reverse Pass

Clad.AD Plugin for Clang

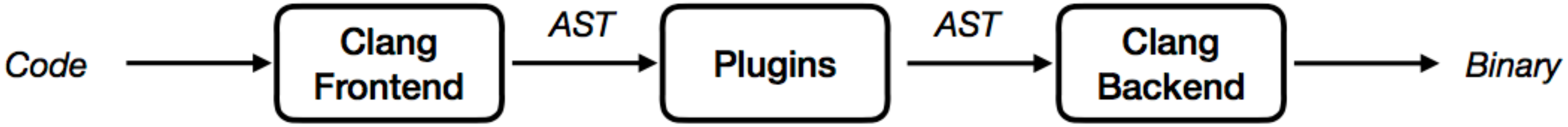
Clad is a compiler plugin extending Clang able to produce derivatives in both forward and reverse mode:

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

Typical C++ Compilation Pipeline



Code → **Compiler Frontend** → **Compiler Backend** → *Binary*



Code → **Clang Frontend** —*AST*→ **Plugins** —*AST*→ **Clang Backend** → *Binary*

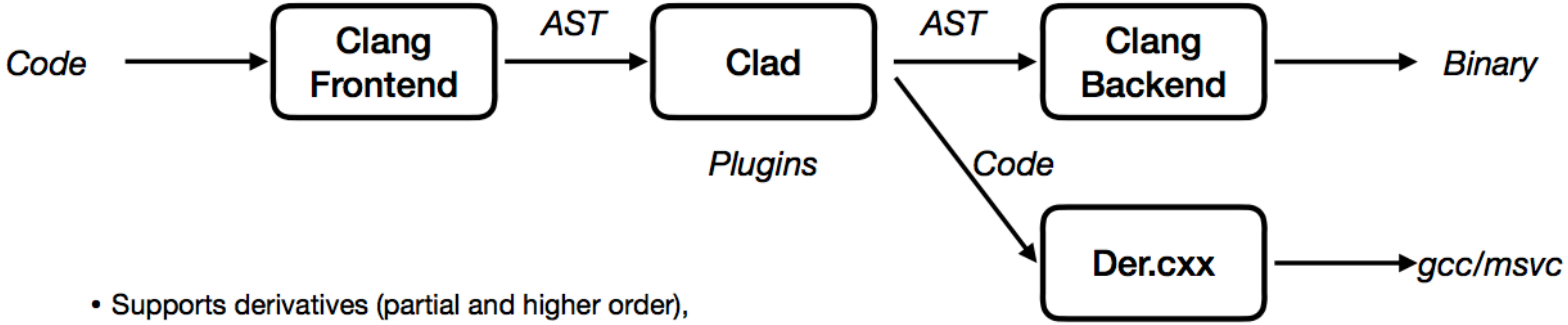
V. Vassilev, L. Moneta *Automatic Differentiation in C++ with clad. Integration in ROOT* *94th ROOT PPP Meeting*

Clang Compilation Pipeline. Clad

```
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'
```


Code

Clang Frontend

AST

Clad

Plugins

AST

Clang Backend

Binary

Code

Der.cxx

gcc/msvc

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

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

V. Vassilev Clad — Automatic Differentiation for C++ Using Clang *Mini-Workshop: Differentiable Programming for High-Performance, Data-Intensive Computations*

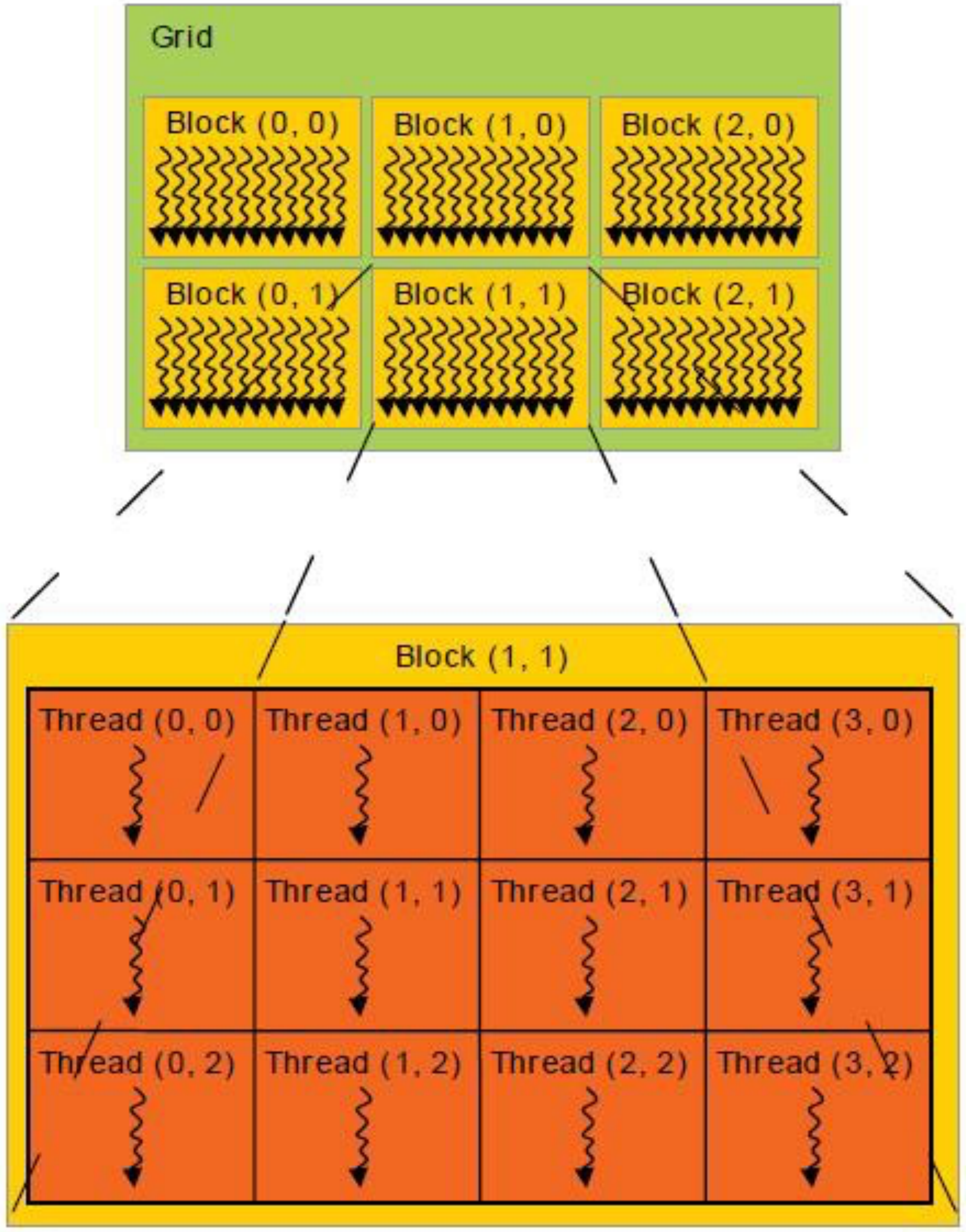
GPU Accelerated AD

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.



Cuda Thread

GPU Accelerated AD

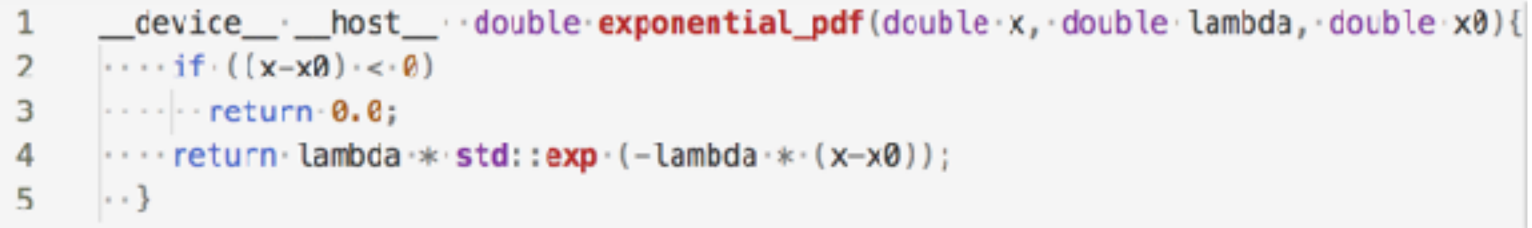
The CUDA support for Clad includes extensions that allow one to execute functions on the GPU using many threads in parallel.

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.

Original Function

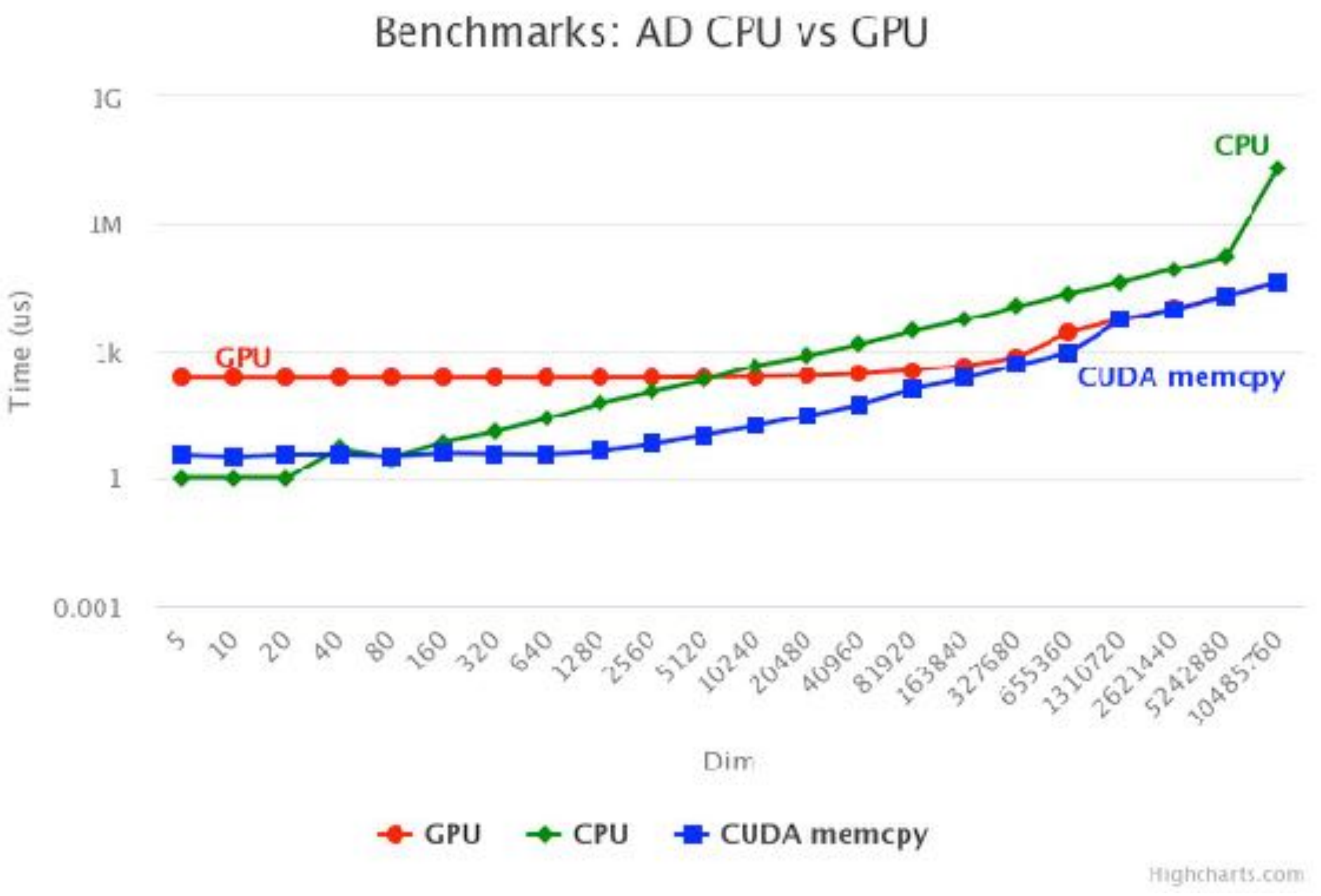


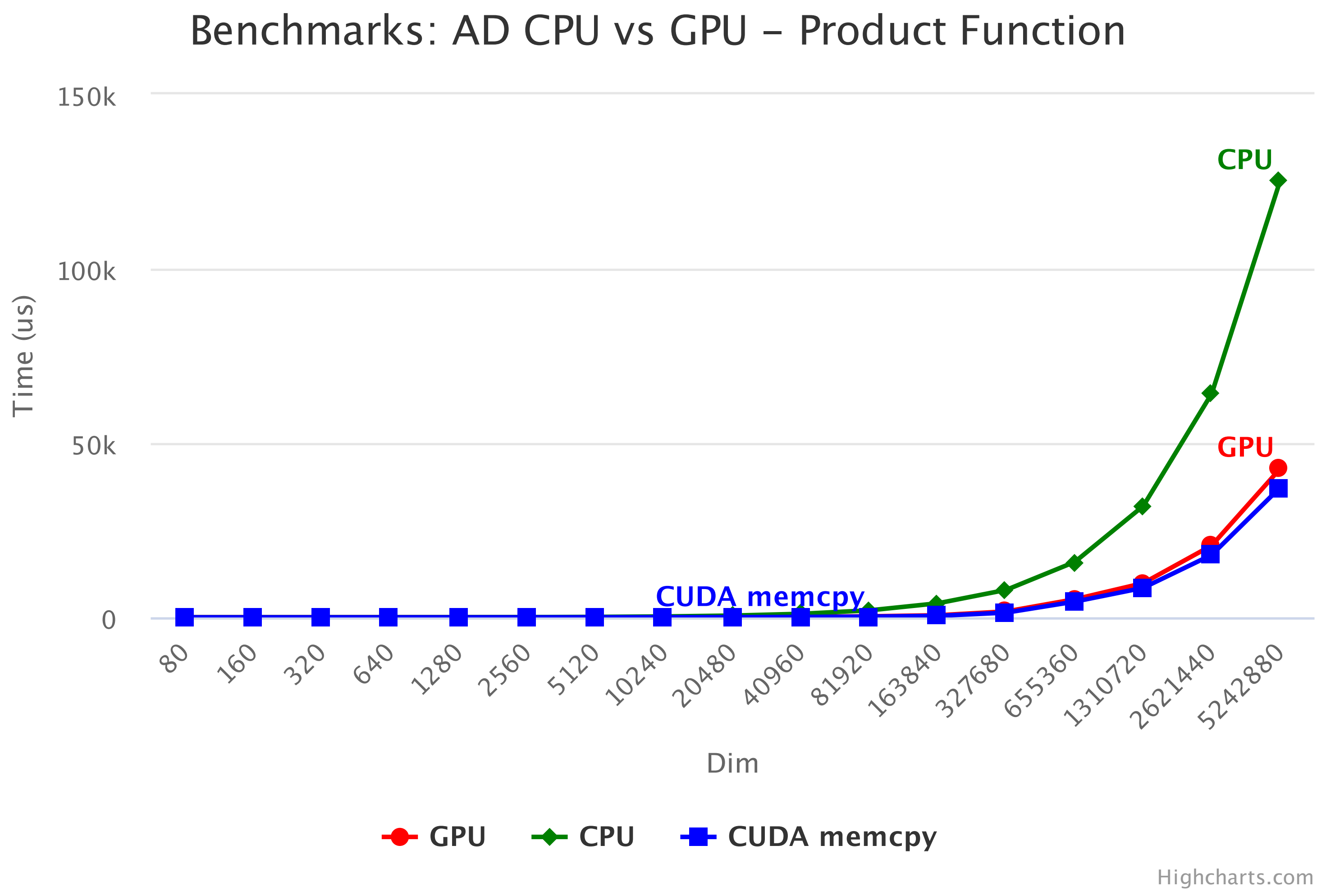
```
__device__ __host__  double exponential_pdf(double x, double lambda, double x0){
    if ((x-x0) < 0)
      return 0.0;
    return lambda * std::exp (-lambda * (x-x0));
  }
```

Clad Gradient

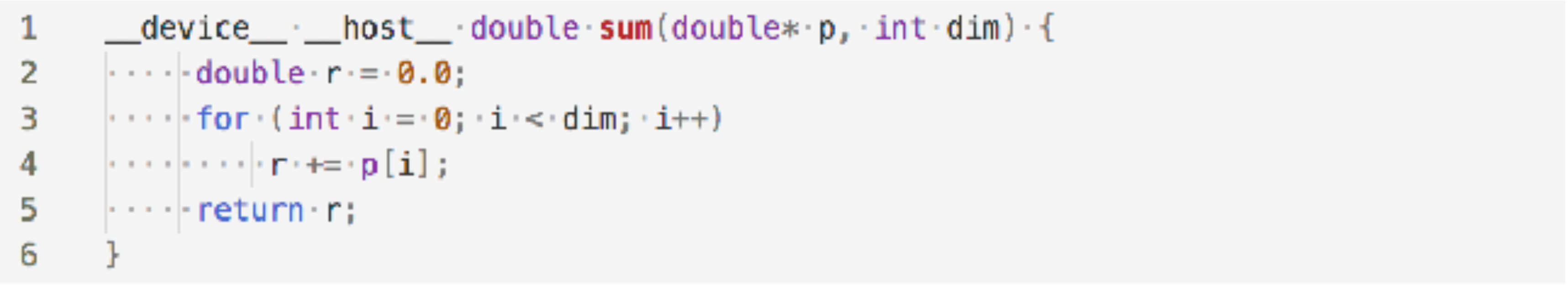
```
void exponential_pdf_grad(double x, double lambda, double x0, double *_result) 
__attribute__((device)) __attribute__((host)) {
    bool _cond0;
    double _t0;
    double _t1;
    double _t2;
    double _t3;
    double _t4;
    _cond0 = (x - x0) < 0;
    if (_cond0) {
        double exponential_pdf_return = 0.;
        goto _label0;
    }
    _t1 = lambda;
    _t3 = -lambda;
    _t2 = (x - x0);
    _t4 = _t3 * _t2;
    _t0 = std::exp(_t4);
    double exponential_pdf_return = _t1 * _t0;
    goto _label1;
  _label1:
    {
        double _r0 = 1 * _t0;
        _result[1UL] += _r0;
        double _r1 = _t1 * 1;
        double _r2 = _r1 * custom_derivatives::exp_darg0(_t4);
        double _r3 = _r2 * _t2;
        _result[1UL] += -_r3;
        double _r4 = _t3 * _r2;
        _result[0UL] += _r4;
        _result[2UL] += -_r4;
    }
    if (_cond0)
      _label0:
        ;
}
```

GPU Accelerated AD



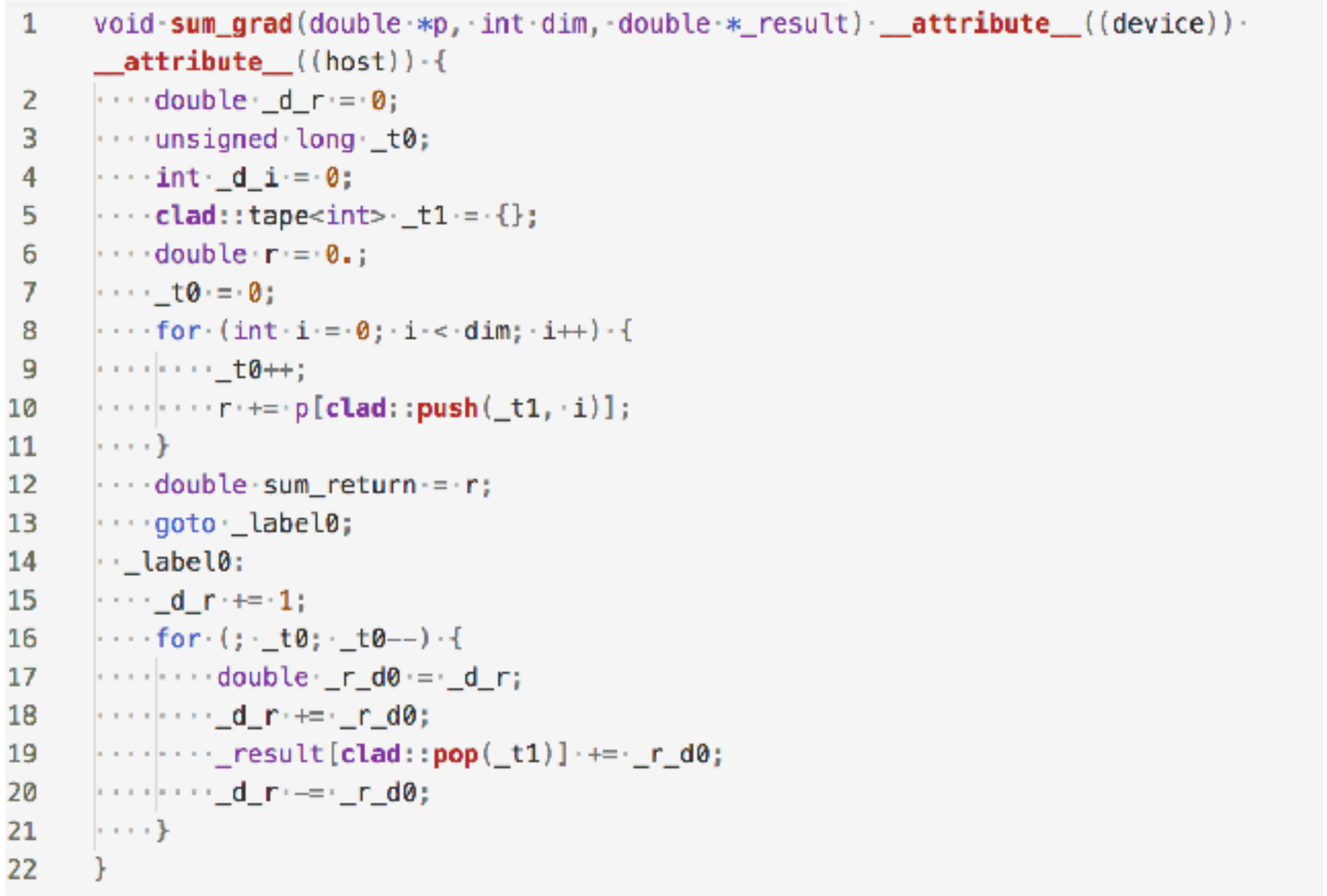


Original Function



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

Clad Gradient

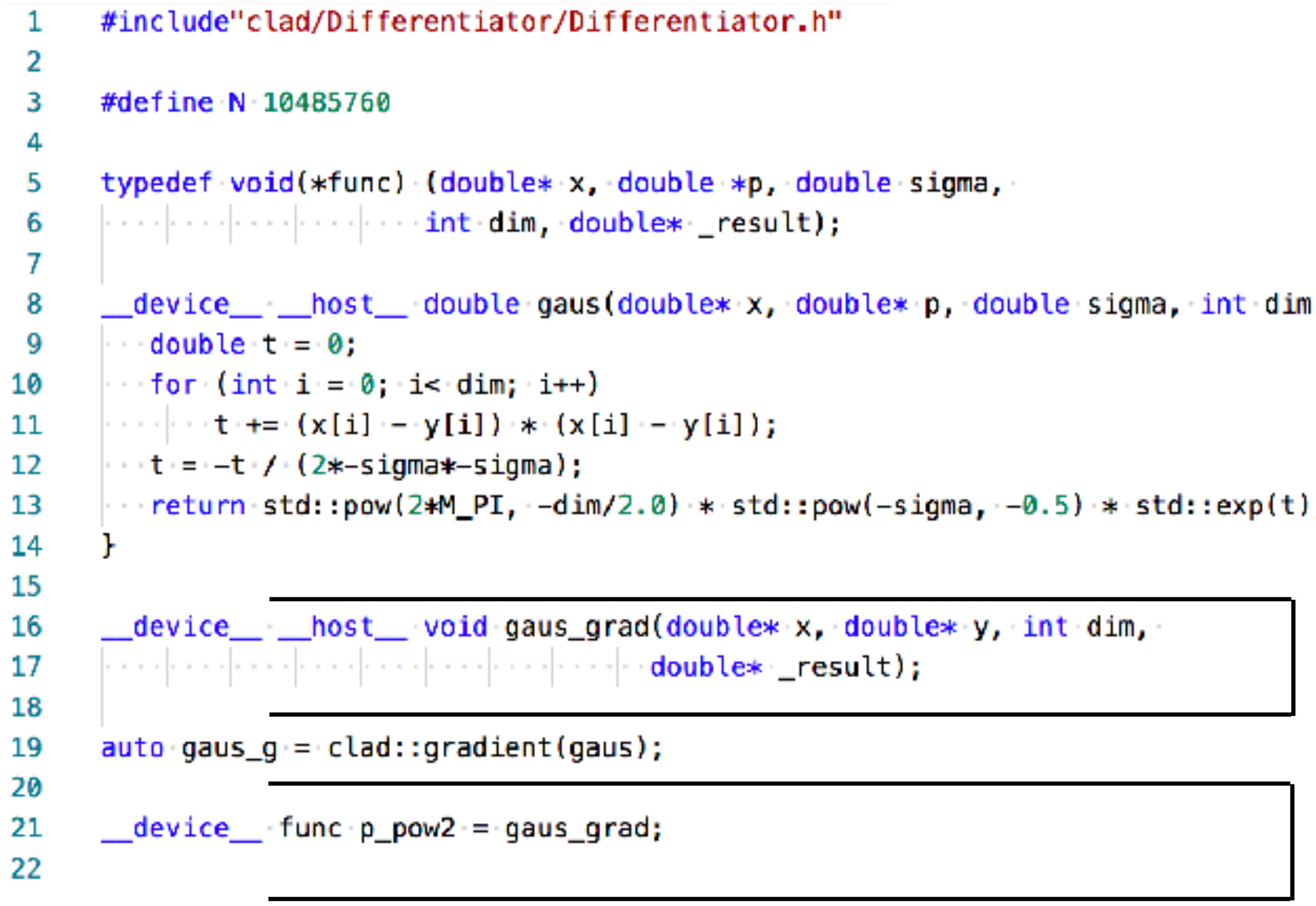


```
void sum_grad(double *p, int dim, double *_result) __attribute__((device)) 
__attribute__((host)) {
    double _d_r = 0;
    unsigned long _t0;
    int _d_i = 0;
    clad::tape<int> _t1 = {};
    double r = 0.;
    _t0 = 0;
    for (int i = 0; i < dim; i++) {
        _t0++;
        r += p[clad::push(_t1, i)];
    }
    double sum_return = r;
    goto _label0;
  _label0:
    _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;
    }
}
```

- Benchmark showcases how using CUDA can influence the overall AD performance in computation of a gauss gradient (left) / Product Function (right) with different dimensions

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

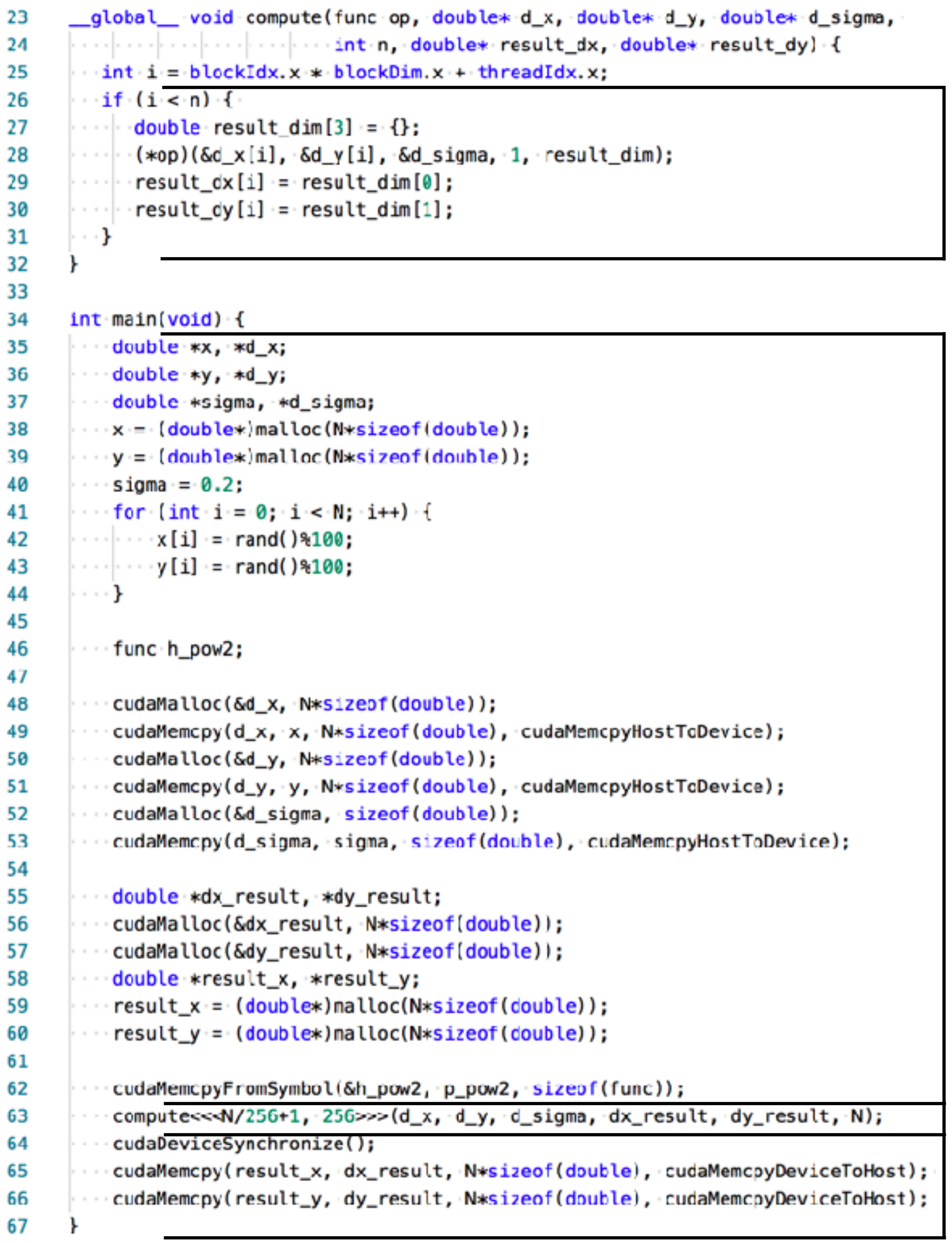
GPU Accelerated AD



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

#define N 10485760

typedef void(*func) (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] - y[i]) * (x[i] - y[i]);
  t = -t / (2*-sigma*-sigma);
  return std::pow(2*M_PI, -dim/2.0) * std::pow(-sigma, -0.5) * std::exp(t);
}

__device__ __host__ void gaus_grad(double* x, double* y, int dim,
                                   double* _result);

auto gaus_g = clad::gradient(gaus);

__device__ func p_pow2 = gaus_grad;

```

body generated by Clad

device function pointer



```
__global__ void compute(func op, double* d_x, double* d_y, double* d_sigma,
                        int n, double* result_dx, double* result_dy) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {
    double result_dim[3] = {};
    (*op)(&d_x[i], &d_y[i], &d_sigma, 1, result_dim);
    result_dx[i] = result_dim[0];
    result_dy[i] = result_dim[1];
  }
}

int main(void) {
   double *x, *d_x;
   double *y, *d_y;
   double *sigma, *d_sigma;
   x = (double*)malloc(N*sizeof(double));
   y = (double*)malloc(N*sizeof(double));
   sigma = 0.2;
   for (int i = 0; i < N; i++) {
       x[i] = rand()%100;
       y[i] = rand()%100;
   }

   func h_pow2;

   cudaMalloc(&d_x, N*sizeof(double));
   cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);
   cudaMalloc(&d_y, N*sizeof(double));
   cudaMemcpy(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice);
   cudaMalloc(&d_sigma, sizeof(double));
   cudaMemcpy(d_sigma, sigma, sizeof(double), cudaMemcpyHostToDevice);

   double *dx_result, *dy_result;
   cudaMalloc(&dx_result, N*sizeof(double));
   cudaMalloc(&dy_result, N*sizeof(double));
   double *result_x, *result_y;
   result_x = (double*)malloc(N*sizeof(double));
   result_y = (double*)malloc(N*sizeof(double));

   cudaMemcpyFromSymbol(&h_pow2, p_pow2, sizeof(func));
   compute<<<N/256+1, 256>>>(d_x, d_y, d_sigma, dx_result, dy_result, N);
   cudaDeviceSynchronize();
   cudaMemcpy(result_x, dx_result, N*sizeof(double), cudaMemcpyDeviceToHost);
   cudaMemcpy(result_y, dy_result, N*sizeof(double), cudaMemcpyDeviceToHost);
}
```

parallel function

serial code

parallel code

serial code

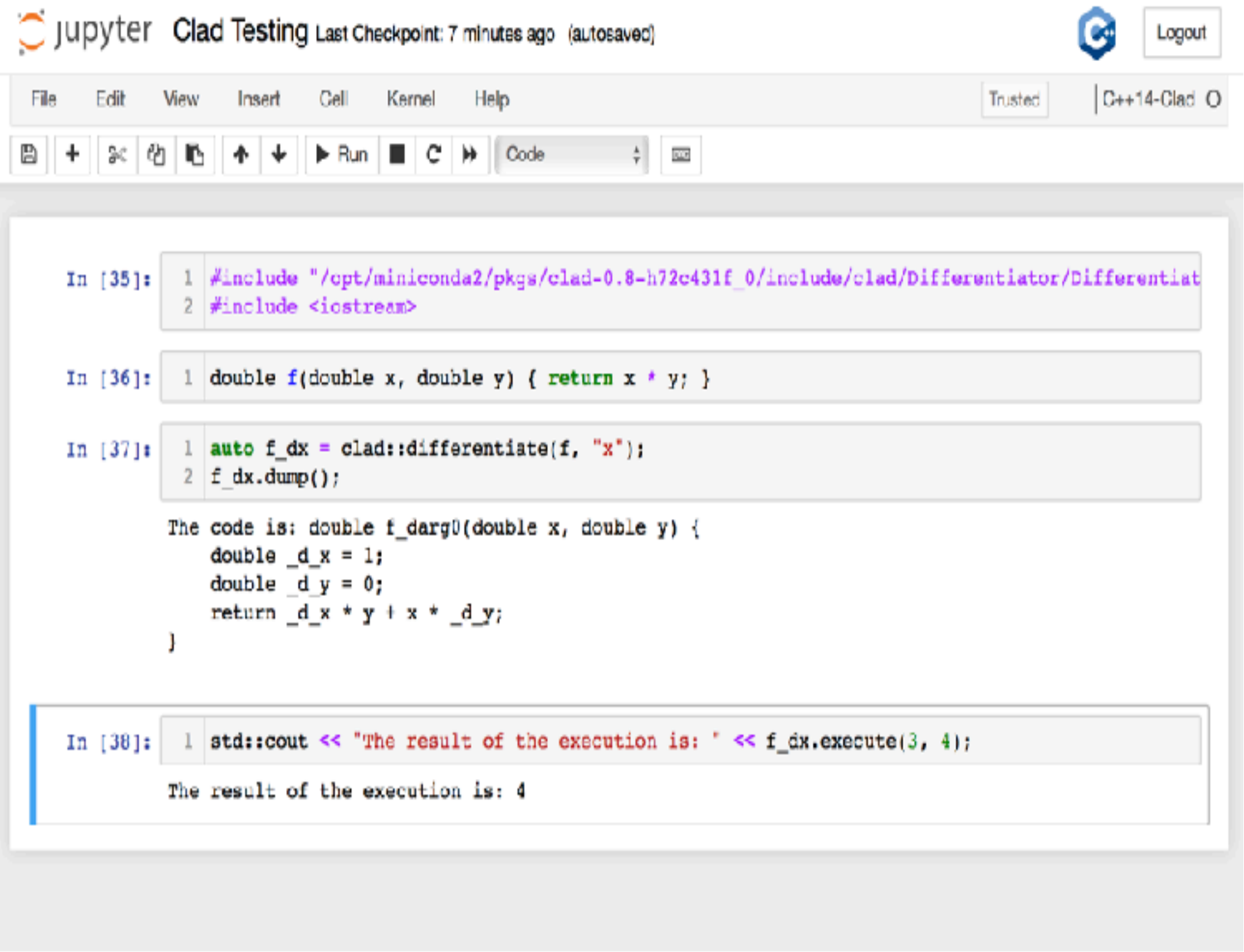
Clad & CUDA as a Service

```
[ii3193@tiger-i23g16 builddir]$
```

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

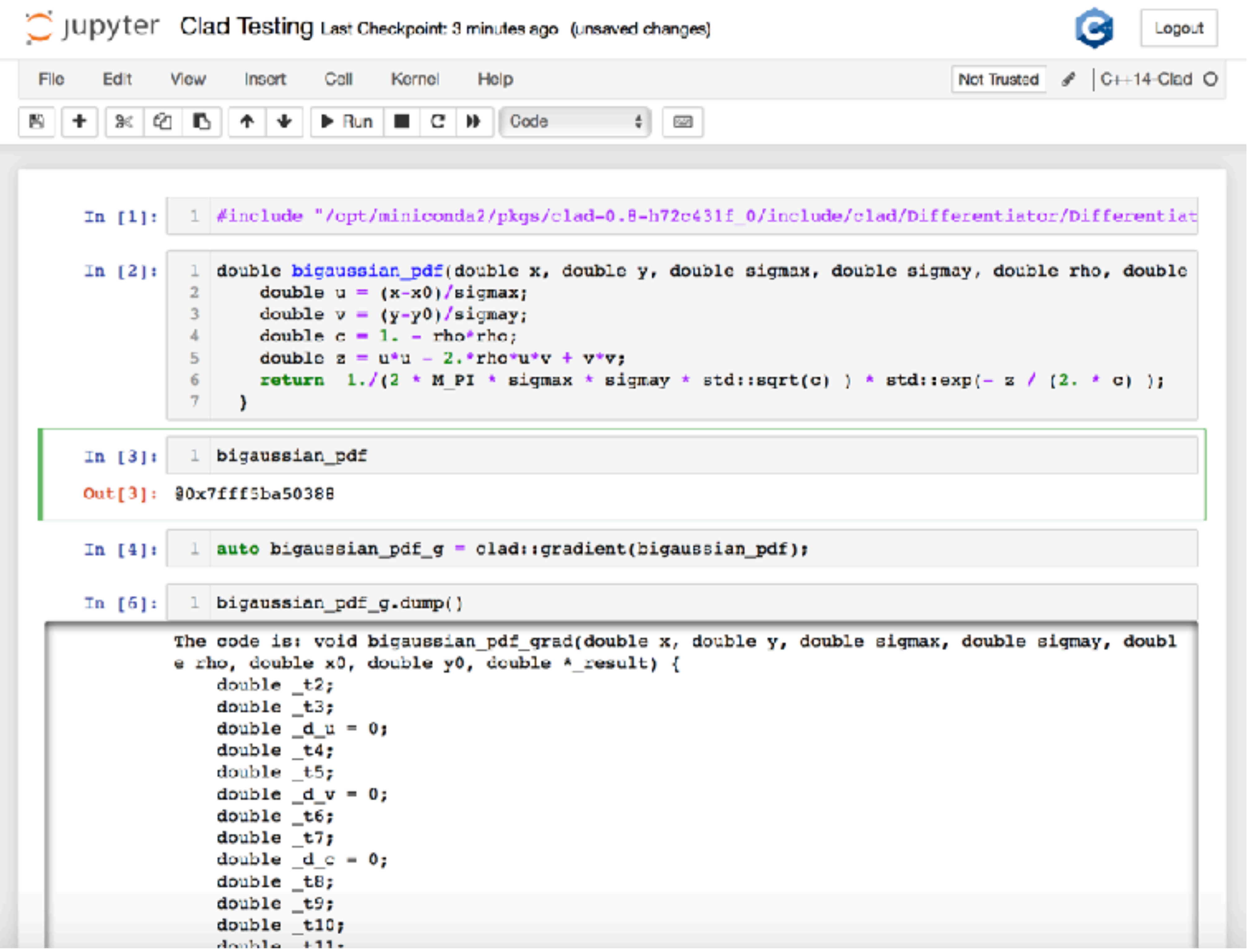
Clad & CUDA as a Service

Forward Mode



```
In [35]: #include "/opt/miniconda2/pkgs/clad-0.8-h72c431f_0/include/clad/Differentiator/Differentiat
         #include <iostream>

In [36]: double f(double x, double y) { return x * y; }

In [37]: auto f_dx = clad::differentiate(f, "x");
         f_dx.dump();

         The code is: double f_darg0(double x, double y) {
             double _d_x = 1;
             double _d_y = 0;
             return _d_x * y + x * _d_y;
         }

In [38]: std::cout << "The result of the execution is: " << f_dx.execute(3, 4);

         The result of the execution is: 4
```

Reverse Mode



```
In [1]: #include "/opt/miniconda2/pkgs/clad-0.8-h72c431f_0/include/clad/Differentiator/Differentiat

In [2]: double bigaussian_pdf(double x, double y, double sigmax, double sigmay, double rho, double
            double u = (x-x0)/sigmax;
            double v = (y-y0)/sigmay;
            double c = 1. - rho*rho;
            double z = u*u - 2.*rho*u*v + v*v;
            return  1./(2 * M_PI * sigmax * sigmay * std::sqrt(c) ) * std::exp(- z / (2. * c) );
          }

In [3]: bigaussian_pdf
Out[3]: @0x7fff5ba50388

In [4]: auto bigaussian_pdf_g = clad::gradient(bigaussian_pdf);

In [6]: bigaussian_pdf_g.dump()

        The code is: void bigaussian_pdf_grad(double x, double y, double sigmax, double sigmay, doubl
        e rho, double x0, double y0, double *_result) {
            double _t2;
            double _t3;
            double _d_u = 0;
            double _t4;
            double _t5;
            double _d_v = 0;
            double _t6;
            double _t7;
            double _d_c = 0;
            double _t8;
            double _t9;
            double _t10;
            double _t11;
```

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

Summary

- The generated Clad derivatives are now supported for computations on CUDA kernels thus allowing for further optimisation
- Given that scheduling still requires a certain degree of user input, we work on further automising this
- 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 :
    - Forward mode support implemented by Baidyanath Kundu with reverse mode in progress
- Enable AD support for second order derivatives for HEP analysis through ROOT (data analysis software package) via Clad :
    - In progress as Baidyanath Kundu GSoC Project
- 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

Thank you!