GSoC 2021 Final Report

Utilize second order derivatives from Clad in ROOT

Student: Baidyanath Kundu **Mentors:** Vassil Vassilev, Ioana Ifrim **Organisation:** CERN-HSF

Acknowledgement

I want to say a huge thanks to my mentors Vassil Vassilev and Ioana Ifrim for guiding we throughout the project. They were better mentors than I had hoped for. I am also thankful to my fellow students Garima Singh and Parth Arora for pointing out some issues in my code and thus help me make it better. Last but not the least I would like to thank the members of Cpplang slack for helping me improve my knowledge on template meta programming.

Overview

What is ROOT and TFormula?

ROOT is a framework for data processing, born at CERN, at the heart of the research on high-energy physics. Every day, thousands of physicists use ROOT applications to analyze their data or to perform simulations. ROOT has a clang-based C++ interpreter Cling and integrates with Clad to enable flexible automatic differentiation facility.

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

What is clad?

In mathematics and computer algebra, automatic differentiation (AD) is a set of techniques to numerically evaluate 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). Clad is an AD tool based on Clang which provides the necessary facilities for code transformation.

Project Description

The project aims to add second order derivative support in TFormula using clad::hessian. The PR that added support for gradients in ROOT is taken as a reference and can be accessed here.

Optionally, if time permits, this project will also convert the gradient function into CUDA device kernels.

Objectives

The original objectives of the project were to enable `clad::hessian` support in ROOT:

- Add array differentiation support for arrays named `p` in forward mode
- Add array differentiation support for arrays named `p` in hessian mode
- Add hessian mode support to ROOT
- (Optional) Add CUDA gradient support to ROOT

However, clad needed support for array differentiation, and it is a natural extension of this project, so I took it on. As a result, the amended objectives were:

- Add array differentiation support in forward mode

- Add array differentiation support in reverse mode
- Add array differentiation support in hessian mode
- Update clad interface in ROOT
- Add support for hessian mode in ROOT
- (Optional) Add CUDA gradient support to ROOT

Contributions

The major PRs I contributed in the GSoC period are mentioned here. All of them have been merged.

Clad

PR	Description
#244	Add array differentiation support in forward mode
#237	Add array differentiation support in reverse and hessian mode
#284	Add changes necessary to integrate clad into root
#286	Add support for llvm 13
#291	Add tests for ROOT interface

To see all the PRs contributed to clad by me checkout this link.

ROOT

PR	Description
#8848	Bump clad version to v0.9.
#8371	Initial support of hessian calculation in TFormula using clad

Results

Add array differentiation support in forward mode

Users can now utilise forward mode to derive functions with respect to a specific array element:

```
double f1(double *arr) {
    return arr[0] * arr[0] + arr[1];
}

// Differentiate f w.r.t. arr[0]
auto f1_dx = clad::differentiate(f1, "arr[0]");
// Produces:
// double f_darg0_0(double *arr) {
//     return 1 * arr[0] + arr[0] * 1 + 0;
// }

double arr[2] = {3, 4};

// Execute the generated derivative
```

```
double darr0 = f1_dx.execute(arr);
// Results in: darr0 = 6
```

The Plan:

The basic concept was that the array components may be thought of as separate variables grouped together. Forward mode differentiates the source function with respect to a single variable, we just needed to activate differentiation of the requested element (by multiplying by `1`) and prevent differentiation of the rest (by multiplying them with `0`).

As an example consider the source function to be:

```
double f2(double* arr) {
  return arr[0] * arr[1];
}
```

The derivative produced with respect to `arr[1]` is:

```
double f2_darg0_1(double *arr) {
    // Differentiation with respect to arr[1] is arr[0] and it is enabled by
multiplying with 1
    // Differentiation with respect to arr[0] is arr[1] and it is disabled by
multiplying with 0
    return 0 * arr[1] + arr[0] * 1;
}
```

Challenges Faced:

Since this was very close to the original objective of adding array differentiation support for arrays named `p` it went mostly according to the proposed plan. A slight challenge that I faced was differentiating array subscript expressions which had expressions as their subscript. To fix this we check if subscript expression is equal to the index that the user has requested and set the dx variable accordingly.

So for a function like this:

```
double f3(double* arr) {
  int x = 1;
  return arr[x]; // The subscript used is an expression ("x")
}
```

The derivative w.r.t. `arr[1]` produced would be:

```
clad::differentiate(f3, "arr[1]"); // To differentiate f w.r.t. to arr[1]

double f3_darg0_1(double *arr) {
    int _d_x = 0;
    int x = 1;
    return (x == 1); // The subscript expression ("x") is checked against
                     // the requested index of arr (which is "1" in this case
}
```

Add array differentiation support in reverse mode

This enables reverse mode to differentiate w.r.t. to arrays:

```
double g1(double *arr) {
    return arr[0] * arr[0] + arr[1];
}

// Differentiate f w.r.t each of the elements in arr
auto g1_darr = clad::gradient(g1);
// Produces:
// void g1_grad(double *arr, clad::array_ref<double> _d_arr) {
//     double _t0;
//     double _t1;
//     _t1 = arr[0];
//     _t0 = arr[0];
//     double g1_return = _t1 * _t0 + arr[1];
//     goto _label0;
//   _label0:
//     {
//         double _r0 = 1 * _t0;
//         _d_arr[0] += _r0;
//         double _r1 = _t1 * 1;
//         _d_arr[0] += _r1;
//         _d_arr[1] += 1;
//     }
// }


double arr[2] = {3, 4};

// The array needs to be of the same size as arr
double darr[2] = {0, 0};
clad::array_ref<double> darr_ref(darr, 2); // here 2 is the size of darr

// Execute the generated gradient
g1_darr.execute(arr, darr_ref);
// Results in: darr = {6, 1}
```

The Plan:

Because it was an unexpected objective, the PR to add array differentiation to reverse mode was largely unplanned. Along with that, there had been a lot of debate about it, but no one could agree on which technique would be the best to adopt. As a result, I had to first construct a prototype so that we could decide which path to follow, and I immediately realised that it implied some significant design challenges.

Challenges Faced:

To understand the challenges, note the gradient's signature before the addition of array differentiation:

```
// Source Function
double g2(double x, double y) {
  . . .
}

// Call clad gradient on g2
```

```
clad::gradient(g2);

// Gradient function generated by clad
void g2_grad(double x, double y, double* _result) { // the _result variable stores the derivative w.r.t. both x and y
  . . .
}
```

The initial plan was to split `_result` into one output variable for each input variable, such that the signature would look like:

```
void g2_grad(double x, double* _d_x, double y, double *_d_y) { // the _d_x variable stores the derivative w.r.t. x and similarly _d_y is for y
  . . .
}
```

Despite being the most user-friendly, this signature has a flaw. Users can specify the input variable from which they wish to derive the function, and execute requires the signature of the derived function to function. Clad is divided into two parts: the user interface, with which the user interacts to call clad to derive a function, and the plugin, which reads the arguments given by the user to clad::gradient (or the other clad functions). Because `execute` is part of the user interface, it has to know the resultant function signature before clad knows which variables the user wishes to derive against. As a result, clad requires that the resultant gradient function have the same signature regardless of the variables supplied. We solved this problem by modifying the function signature to relocate all the output variables to the end of the function and introducing an overload when the user just wants a subset of the input variables.

So if the user calls:

```
// Source Function
double g3(double x, double y, double z) {
  . . .
}

clad::gradient(g3, "y");
```

clad would produce:

```
void g3_grad_1(double x, double y, double z, double *_d_y) { // The actual gradient that is produced
  . . .
}

void g3_grad_1(double x, double y, double z, double *_d_y, double *_d_0, double *_d_1)
{ // The overload created to interface with "execute"
  f_grad_1(x, y, _d_y);
}
```

When the user calls `execute`, clad pads the user arguments with `nullptr`, and invokes the gradient overload.

Another issue that needed to be addressed was the size of the array needed for differentiating call expressions. To address this, we created a new class called `clad::array_ref` to hold the diff, which accepts an array pointer and its size. This approach also allows the user to submit variable-sized arrays, making it rather versatile. If the user just gives a pointer, it thinks it is a single variable with a size of `1`.

So finally the signature of the produced gradient is:

```
// Source function
double g4(double *x, double y) {
  . . .
}

// Call to clad gradient on g4
clad::gradient(g4);

// The produced gradient
double g4_grad(double *x, double y, clad::array_ref<double> _d_x,
clad::array_ref<double> _d_y) {
  . . .
}
```

Add array differentiation support in hessian mode

This added support for generating hessians with functions that take array inputs:

```
double h1(double *x) {
    return x[0] * x[0] + x[1] * x[1];
}

// Generate hessian w.r.t. to all elements in x
auto h_hess = clad::hessian(h1, "x[0:1]");

double x[2] = {2, 3};

// Hessian produces a matrix so the size required is the square
// of number of variables we are differentiating against
// Since we are passing a single array of size 2 the hessian matrix
// generated is of size 4
double hess[4] = {0};
clad::array_ref<double> hess_ref(hess, 4);

// Executes the generated hessian
h_hess.execute(x, hess_ref);
// Results in: hess = {2, 0, 0, 2}
```

The Plan:

The original plan was to try to find the size of the `p` array by counting the indexes used in the source function but the modified objective meant that the size arrays could no longer be calculated by hessian.

Challenges Faced:

The difficulty we encountered was determining the size of the input arrays. The hessian function returns a flattened matrix, and using a `clad::array_ref` for the hessian matrix does not provide the different array sizes. To address this, `clad::hessian` was modified to take the array size via its args parameter. As an example, consider the following:

```
double h2(double *x, double *y) {
  return x[0] * y[0] + x[1] * y[1] + x[2] * x[3]
}

// The size of x in h2 function is 3 and x[0:2]
// means differentiate indexes 0 through 2 for x. \
// Same for y
clad::hessian(h2, "x[0:2], y[0:2]");
```

Using the index representation we were able to get the size of the arrays that we need to differentiate.

Update clad interface in ROOT

The Plan:

This was a straightforward objective. Because the signature of the produced gradient has changed, the ROOT interface with clad had to be modified to match the changes

Challenges Faced:

One minor issue we discovered was that executing code through the ROOT interpreter is slow, so we had to avoid it wherever feasible. Initially, the interpreter was used to construct and delete the `clad::array_ref` object needed to provide to the produced gradient. Because this is called every time the user wants to employ the produced gradient, it significantly slowed things down. To get around this, one solution was to expose the `clad::array_ref` header to the TFormula class, which incurred extra complexity. Instead, we used a struct with the same data members as `clad::array` ref, and because the objects are provided as void pointers and explicitly cast to the type of the real argument in the trampoline function (basically a `reinterpret_cast`), we were able to reduce the additional cost by using this technique.

Add support for hessian mode in ROOT

The Plan:

This mostly duplicates the code that supports `clad::gradient` to support `clad::hessian` in ROOT.

(Optional) Add CUDA gradient support to ROOT

I couldn't complete this requirement as a lot of time went into adding array differentiation support for reverse mode. I will put up a PR soon.

Conclusion

I was able to complete my mandatory goals and learnt a lot about automatic differentiation, compilers and template metaprogramming. I had a great time and would like to continue contributing more to the clad, ROOT and the CERN-HSF community in general.

For an in-depth demonstration of how to utilise array differentiation in clad in various modes check this link.