

Add support for differentiating with respect to user-defined types.

Progress Update

Parth Arora

Mentors: Vassil Vassilev, David Lange



Topics to be discussed

- Current Progress
- Design Changes
- Current Challenges
- Next Goals

Current Progress

- Added support for differentiating switch statements in the reverse mode.
- Added support for differentiating scalar types with respect to basic user-defined types using the forward mode differentiation.
- Added support for differentiating scalar types with respect to basic user-defined types using the reverse mode differentiation.

Switch Statement support in the reverse mode.

```
double fn(double i, double j) {
    int count = 0;
    double a = 0;

    switch (count) {
        case 0: a += i; break;
        case 2: a += 4 * i; break;
        default: a += 10 * i;
    }
    return a;
}

auto d_fn = clad::gradient(fn);
double res[2] = {};
d_fn.execute(3, 5, res, res+1);
```

Differentiating scalar type wrt user-defined types in the forward mode

```
class ComplexNumber {
public:
    double real = 0, im = 0;
    ComplexNumber(double p_real, double p_im) : real(p_real), im(p_im) {}
};

class __clad_double_wrt_ComplexNumber;
class __clad_ComplexNumber_wrt_ComplexNumber;

double fn(ComplexNumber c, double i) { }

auto d_fn = clad::differentiate(fn, "c");
ComplexNumber c(3, 5);
auto res = static_cast<__clad_double_wrt_ComplexNumber*>(d_fn.execute(c, 7));
std::cout<<"Differentiation of fn(c, 7) wrt c.real: "<<res->real<<"\n";
std::cout<<"Differentiation of fn(c, 7) wrt c.im: "<<res->im<<"\n";
```

Differentiating scalar types wrt to user-defined types in the reverse mode

```
class ComplexNumber {
public:
    double real = 0, im = 0;
    ComplexNumber(double p_real, double p_im) : real(p_real), im(p_im) {}
};

class __clad_double_wrt_ComplexNumber;

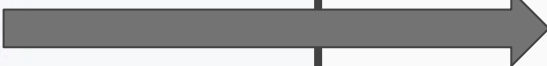

double fn(ComplexNumber c, double i) { }

auto d_fn = clad::gradient(fn);
ComplexNumber c(3, 5);
double d_i = 0;
__clad_double_wrt_ComplexNumber d_c;
d_fn.execute(c, 11, &d_c, &d_i);
std::cout<<"Derivative of fn(c, 11) wrt c.real: "<<d_c.real<<"\n";
std::cout<<"Derivative of fn(c, 11) wrt c.im: "<<d_c.im<<"\n";
```

Switch Statement support

Switch Statement support in the reverse mode

- Basic idea used is, if we can keep track of which *switch case* was selected and which *break* statement was hit in the forward pass of the switch statement, then, in the reverse pass, we can execute derived statements of the switch statement body that were executed in the forward pass.
- The information of which *switch case* was selected and which *break* statement was hit is further used by an auxiliary switch statement to jump the execution to the correct point in the reverse pass.

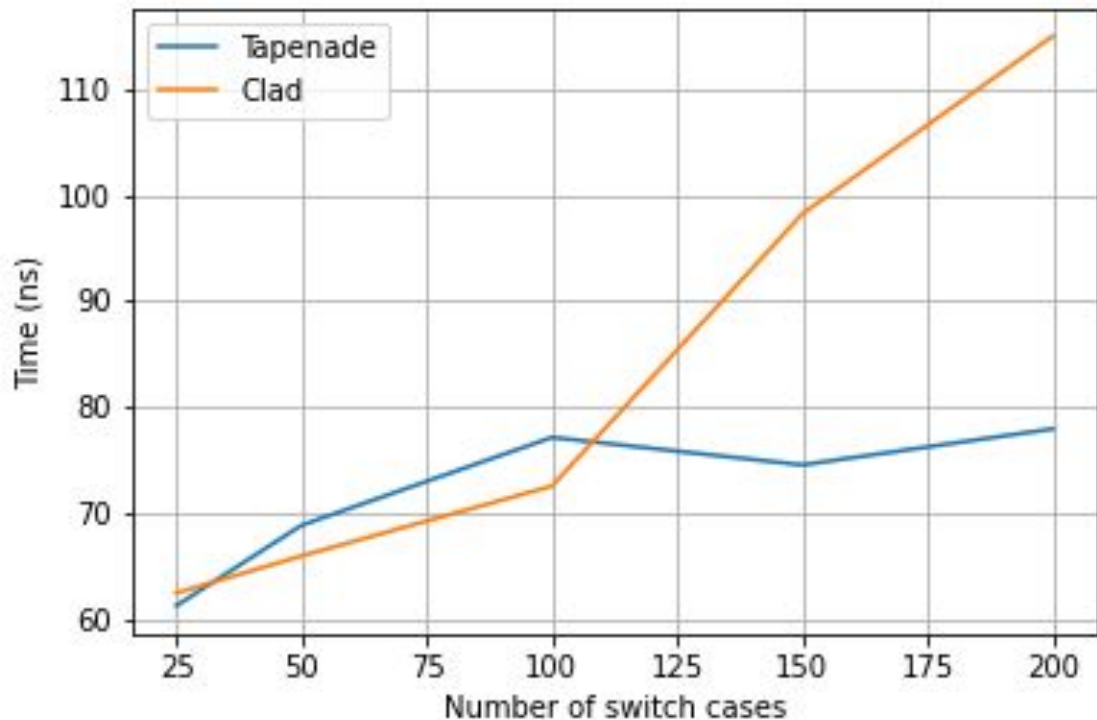

```
switch (count) {  
  case 0:   
    res += i;  
    break;   
  case 1:  
    res += j;  
    break;  
}
```

*If (cond == 0)
break;*

A case label is associated with each break statement. The case label is used to jump the execution to the correct point if this break statement was hit in the forward pass

```
switch(branch) {
  case 10: // if 10th break statement was hit in the forward pass
    ...
    ...
    if (cond == 4) // case statement with value 4 was selected in the forward pass
      break;
  case 9: // if 9th break statement was hit in the forward pass
    ...
    ...
    if (cond == 3) // case statement with value 3 was selected in the forward pass
      break;
  // .... and so on ...
  // ...
  // ...
}
```

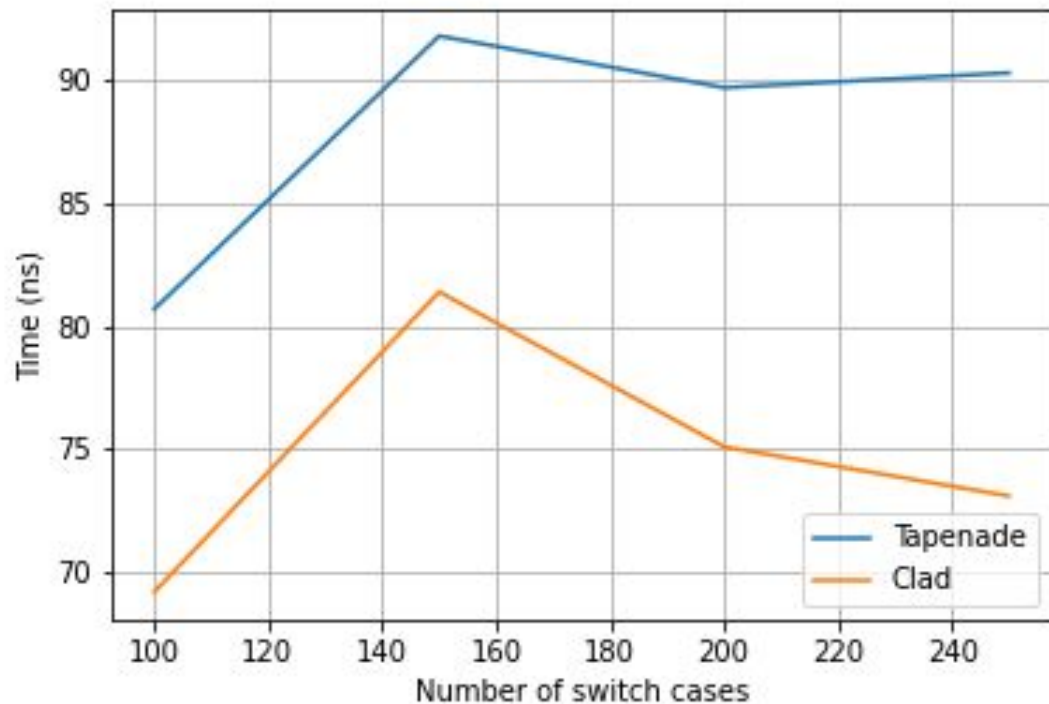
Benchmarks



Probability of
break statement
per case: 80%

Compiler: GNU g++
Optimization: -O3

Benchmarks



Probability of
break
statement per
case: 30%

Compiler: GNU g++
Optimization: -O3

Benchmarks

- Runtime of Tapenade's implementation generally increases with number of case statements, but is not as much affected by number of break statements.
- Runtime of Clad's implementation increases with number of break statements, but is not as much affected by number of case statements.

Derived Types

Derived Types

- What should be type of a variable that stores derivative of a *double* variable with respect to a *ComplexNumber* variable?

```
class ComplexNumber {  
    double real, im;  
}
```

- What should be the type of a variable that stores derivative of a *ComplexNumber* variable with respect to a *ComplexNumber* variable?

Derived Types

```
class __clad_double_wrt_ComplexNumber {  
    double real = 0, im = 0;  
};
```

```
class __clad_ComplexNumber_wrt_ComplexNumber {  
    __clad_double_wrt_ComplexNumber real, im;  
};
```


How to interpret these derived types?

```
__clad_double_wrt_ComplexNumber d_i; // represents derivative of a double i wrt a
                                     // ComplexNumber c
d_i.real; // derivative of i wrt c.real
d_i.im;   // derivative of i wrt c.im
```

```
__clad_ComplexNumber_wrt_ComplexNumber d_c; // represents derivative of c1 wrt c2
d_c.real.real; // derivative of c1.real wrt c2.real
d_c.real.im;   // derivative of c1.real wrt c2.im
d_c.im.real;   // derivative of c1.im wrt c2.real
d_c.im.im;     // derivative of c1.im wrt c2.im
```

Derived Types

- What should be the type of a variable that stores derivative of a *double* variable with respect to a *ComplexNumberPair* variable?

```
class ComplexNumberPair {  
    ComplexNumber c1, c2;  
};
```

- What should be the type of a variable that stores derivative of a *ComplexNumber* variable with respect to a *ComplexNumberPair* variable?

Derived Types

```
class __clad_double_wrt_ComplexNumberPair {  
    __clad_double_wrt_ComplexNumber c1, c2;  
};
```

Reusing the derived types infrastructure instead of creating all the nested fields everytime

```
class __clad_ComplexNumber_wrt_ComplexNumberPair {  
    __clad_ComplexNumber_wrt_ComplexNumber real, im;  
}
```

How to interpret these derived types?

```
__clad_double_wrt_ComplexNumberPair d_i; // Represents derivative of a double i
                                         // wrt a ComplexNumberPair CP
d_i.c1.real; // derivative of i wrt CP.c1.real
d_i.c1.im;   // derivative of i wrt CP.c1.im
```

```
__clad_ComplexNumber_wrt_ComplexNumber d_c; // Represents derivative of a
                                              // ComplexNumber c wrt a
                                              // ComplexNumberPair CP
d_c.real.c1.real; // derivative of c.real wrt CP.c1.real
d_c.im.c2.real;   // derivative of c.im wrt CP.c2.real
```

Derived Types

- Derived types are used to store derivative of a variable of type T1 with respect to a variable of type T2.
- Derived types can be easily and efficiently generated algorithmically.
- Derived types contains data members of other derived types. This infrastructure reuse makes derived types easier to create, use and understand.

Derived Types

- Users need to forward-declare each derived type that is required by clad.

```
class __clad_double_wrt_ComplexNumber;  
class __clad_ComplexNumber_wrt_ComplexNumber;
```

- Clad will automatically generate body of the derived types.
- Users will directly use variables of derived types to access the derivatives. Thus, derived types need to be visible to the users.


Differentiating scalar types with respect to user-defined types in the forward mode

- If we are differentiating a function in the forward mode with respect to a variable of user-defined type containing n member variables, then it is equivalent to differentiating the function n times in the forward mode. One direct advantage here is that computationally expensive operations are computed only once instead on n times.

Differentiating a *double* variable wrt a *ComplexNumber* variable.

```
res = u + v; // res, u and v are of double type
```


```
_d_res = dAdd(_d_u, _d_v);
```



```
__clad_double_wrt_ComplexNumber dAdd(__clad_double_wrt_ComplexNumber d_u,  
                                       __clad_double_wrt_ComplexNumber d_v) {  
    __clad_double_wrt_ComplexNumber d_res;  
    d_res.real = d_u.real + d_v.real;  
    d_res.im = d_u.im + d_v.im;  
    return d_res;  
}
```


Differentiating a *double* variable wrt a *ComplexNumber* variable.

```
res = u*v;  
_d_res = dMultiply(u, _d_u, v, _d_v);
```



```
__clad_double_wrt_ComplexNumber dMultiply(double u,  
                                           __clad_double_wrt_ComplexNumber d_u,  
                                           double v,  
                                           __clad_double_wrt_ComplexNumber d_v) {  
    __clad_double_wrt_ComplexNumber d_res;  
    d_res.real = d_u.real*v + u*d_v.real;  
    d_res.im = d_u.im*v + u*d_v.im;  
    return d_res;  
}
```

Differentiating scalar types with respect to user-defined types in the forward mode

- For each derived type, clad creates functions *dAdd*, *dSub*, *dMultiply* and *dDivide*. These functions contains rules how to produce resultant derivative when original variables are added, subtracted, multiplied and divide respectively.
- These arithmetic derived functions are used by other derived types arithmetic function as well. For example, *dAdd* function for *ComplexNumberPair* type internally used *dAdd* function for *ComplexNumber* type.

InitialiseSeeds Function

- For derived types such as, `__clad_T_wrt_T`, where T is any user-defined type, clad also creates a member function associated with these derived types, *InitialiseSeeds*.

```
void __clad_ComplexNumber_wrt_ComplexNumber::InitialiseSeeds() {  
    real.real = 1;  
    im.im = 1;  
}
```

```
double fn(ComplexNumber c, double i) {  
    double res = 0;  
    res += c.real;  
    res -= i;  
    res *= i;  
    res /= c.im;  
    return res;  
}
```

In the forward mode,
This function will get differentiated as



```
void *fn_darg0(ComplexNumber c, double i) {
    __clad_ComplexNumber_wrt_ComplexNumber _d_c;
    _d_c.InitialiseSeeds();
    __clad_double_wrt_ComplexNumber _d_i;
    __clad_double_wrt_ComplexNumber _d_res;
    double res;
    _d_res = dAdd(_d_res, _d_c.real);
    res += c.real;
    _d_res = dSub(_d_res, _d_i);
    res -= i;
    _d_res = dMultiply(res, _d_res, i, _d_i);
    res *= i;
    double &t0 = c.im;
    _d_res = dDivide(res, _d_res, t0, _d_c.im);
    res /= t0;
    __clad_double_wrt_ComplexNumber *_t1 = new
        __clad_double_wrt_ComplexNumber(_d_res);
    return _t1;
}
```

Differentiating scalar types with respect to user-defined types in the reverse mode.

- Differentiating a function with respect to a variable of a user-defined type that contain n members in the reverse mode is effectively same as differentiating the function with respect to a variable of a scalar type.

Current limitations of differentiating scalar types with respect to user-defined types

- User-defined types should only contain scalar fields. Thus, nested aggregate types are not supported.
- Operator overloads and member functions are not supported

Design Changes

Major Design Changes

- Now `clad::differentiate`, returns a void pointer. Users have to use `static_cast` to cast the result to correct type before using it.
- Now `clad::gradient`, takes derived arguments of type `clad::array_ref<void>` instead of `clad::array_ref<FnReturnType>`.

Major Design changes

```
void *fn_darg0(ComplexNumber c, double i) {  
    ...  
    ...  
}
```

```
void fn_grad(ComplexNumber c, double i, clad::array_ref<void> _d_c,  
             clad::array_ref<void> _d_i) {  
    ...  
    ...  
}
```

Major Design Changes

- Using advance metaprogramming techniques we can avoid returning a pointer and instead return by value in *clad::differentiate*, thus maintaining the current design.

```
__clad_double_wrt_ComplexNumber fn_darg0(ComplexNumber c, double i) {  
    ...  
    ...  
}
```

Major design changes

- Metaprogramming technique will have added restriction that users will only be able to specify independent parameter using parameter index.
- No way to specify differentiate with respect to *arr[3]* using the metaprogramming technique.

Current Challenges

Current Challenges

- Design a more sophisticated syntax for naming derived types. Current syntax cannot handle namespaces, nested class and template notation.
- For namespaces, we can use:

```
namespace A {  
  namespace B {  
    class __clad_double_wrt_ComplexNumber;  
  }  
}
```

Current challenges

- We need to fix a scalar type in which derivatives should be computed.
- *long* and *long double* can be used interchangeably, but *__clad_double_wrt_ComplexNumber* and *__clad_long_double_wrt_ComplexNumber* cannot be used interchangeably.

Next Goals

Next goals

- Refactor all the code and prepare a PR.
- Add support for differentiating calls to overloaded operators and member functions.
- Add support for differentiating with respect to user-defined types containing data members of user-defined types (nested aggregate types).