clad - Automatic Differentiation with Clang

Violeta Ilieva and Vassil Vassilev
Princeton University and CERN

Automatic Differentiation

Differentiation is the process of finding a derivative, which measures how a function changes as its input changes. Derivatives can be evaluated with machine precision accuracy through a method called **automatic differentiation**. Unlike other methods for differentiation, including numerical and symbolic, automatic differentiation yields exact derivatives even of complicated functions at relatively low processing and storage costs.

Chain Rule

Automatic differentiation calculates derivatives by combining the values of basic operations that have known derivatives, which are reached by employing the **chain rule** multiple times on the input function. We chose to apply the chain rule in the forward mode, which means that we propagate derivatives of intermediate variables along with the control flow of the original function. Here is a short example:

$$\frac{d(x+y\cdot z)}{dy} = \frac{d(x)}{dy} + \frac{d(y\cdot z)}{dy} = \frac{d(y)}{dy}\cdot z + y\cdot\frac{d(z)}{dy} = z$$

Source Code Transformation

We implemented automatic differentiation using **source code transformation**, which consists of explicitly building a new source code through a compiler-like process that includes parsing the original program, constructing an internal representation and performing global analysis. It takes into account the context of a particular computation, leading to identification of global dependencies and flexibility in applying derivative rules.

Example

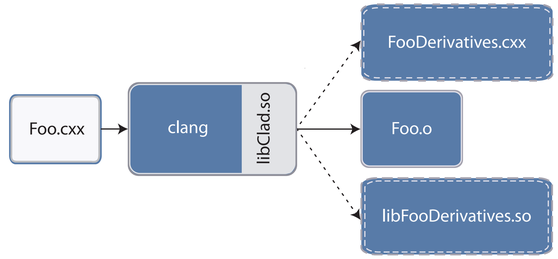
```
#include "Differentiator.h"

float func() { return 12.0; }
namespace custom_derivatives { float func() {return 42.0;} }
float foo() (int x, int y) { return std::sin(custom_derivatives::func())*x+y; }
int foo_derived_x();
int main() { diff(foo, 0).execute(); /*or fwd decl works too*/
             foo_derived_x(int x, int y); return 0; }
```

What is clad?

clad is a C++ plugin for clang that implements **automatic differentiation** of user-defined functions by employing the **chain rule** in forward mode, coupled with **source code transformation** and AST constant fold.

Architecture

Given the sample C/C++ file Foo.cxx shown at the bottom left corner, we can find the derivative of foo by invoking a clang instance that has been extended with our plugin clad. There are 3 possible outcomes and currently clad gives the middle one. That is, the output is an object file that contains the generated derivatives. We are almost done with implementing the creation of a .cxx file and are working towards composing a dynamic library comprised of the derivatives.



Usage

Once a user has defined a function in a C/C++ file, they can invoke its derivative right away in the source code, relying on forward declaration. Before running the file however, they have to invoke clad in one of the two ways shown below. This results in generating either plain derivatives, or folded derivatives, i.e. functions in which the constant computations have been optimized away, improving readability. Sample command-line invocations and derivative source code is shown below.

Derivatives

```
$ clang -cc1 -x c++ -std=c++11 -load libAutoDiff.dylib
-plugin clad -plugin-arg-clad -fprint-derived-fn
-plugin-arg-clad -fprint-derived-fn-ast
/src/tools/autodiff/test/Foo.c
```

```
// input is sin(custom_derivatives::func()) * x + y

float foo_derived_x(int x, int y) {
  return cos(custom_derivatives::func()) * 0 * x
       + sin(custom_derivatives::func()) * 1 + 0;
}
```

Folded Derivatives

```
$ clang -cc1 -x c++ -std=c++11 -load libAutoDiff.dylib
-plugin clad -plugin-arg-clad -fprint-folded-fn
-plugin-arg-clad -fprint-folded-fn-ast
/src/tools/autodiff/test/Foo.c
```

```
// input is sin(custom_derivatives::func()) * x + y
// note that we are not using func(), but custom_derivatives::func()

float foo_derived_x(int x, int y) {
  return sin(custom_derivatives::func());
}
```

Features

Overload resolution.
clad is able to make a difference between the overloaded functions below, resolve correctly the one that was specified to be derived, and carry out the operation, if the function's declaration appears in the custom namespace.

```
int overloaded(int x) {
  printf("I was called!");
  return x*x;
}
int overloaded(float x) {
  return x;
}
```

Library of derivatives of essential functions.
For convenience, we provide a library of predefined derivatives, that use either templates, like *sin*, or overloading, like *cos*, shown to the right. It contains the declarations of all differentiable functions and can be expanded by users. For example, *printf* from above will not be differentiated recursively and the derivative of the second overloaded function will be 3 instead of 1.

```
namespace custom_derivatives {
  template<typename R, typename A> R sin(A x)  { return (R)::std::cos(x); }

  float cos(float arg ) { return -std::sin(arg); }
  double cos(double arg ) { return -std::sin(arg); }
  long double cos(long double arg ) { return -std::sin(arg); }
  double cos(Integral arg ) { return -std::sin(arg); }

  int my_custom_derivative_x(float x)  { return 3; }
}
```

Control flow management.
We are able to handle control flow issues by creating additional variables to avoid ambiguity. For example, the following block

```
if (x < 0)  rslt = rslt * x;
else  rslt = x * x;
return rslt;
```

will yield the following derivative:

```
if (x < 0)  {
   rslt = rslt * x; rslt_derived_x = (rslt_derived_x * x + rslt * 1);
}
else  {
   rslt = x * x; rslt_derived_x = (1 * x + x * 1);
}
return rslt_derived_x;
```

Under the Hood

1. Collect diff invocations through clang's RecursveASTVisitor.

```
diff(foo, 0).execute();
```

2. Parse their arguments.
   - functions to derive - foo
   - independent variables - x.
3. Create a new function that mirrors the input and whose name signifies that it is a derivative with respect to a specific independent variable.

```
int foo(int x, int y) {}
int foo_derived_x(int x, int y) {}
```

4. Visit the different components of the user function definition, then clone and transform them through the compiler according to the chain rule. Here is an overview of how *DeclRefExpr* nodes are handled:

```
NodeContext DerivativeBuilder::VisitDeclRefExpr(DeclRefExpr* DRE) {
  DeclRefExpr* cloneDRE = m_NodeCloner->Clone(DRE);
  if (/* the variable was already declared */)
    return /* the name of its derivative */;
  if ( /* this is an independent varible */) {
    llvm::APInt one(m_Context.getIntWidth(m_Context.IntTy), 1);
    IntegerLiteral* constant1 = IntegerLiteral::Create(m_Context, one,
                                          m_Context.IntTy, SourceLocation());
    return NodeContext(constant1);
  }
  /* this is a dependent variable, so create constant0 */
  return NodeContext(constant0);
}
```

Future

clad will be embedded in the next release of the CERN ROOT Framework. We will also expand the different types of AST nodes that the tool can handle properly, extend the built-in derivative library, expand the template support, and include macros as well as functor objects. Further future work will include making clad pluggable in any compiler and enabling it to offload computations to the GPU by using OpenCL.

Acknowledgements

We would like to thank Lorenzo Moneta and Alexander Penev who provided valuable comments, ideas, and assistance. The presented work was facilitated by Google Summer of Code 2013.

Contact Information

- **Web**: https://code.google.com/p/clang-auto-differentiation-plugin
- **Email**: vilieva@princeton.edu, vvasilev@cern.ch
- **Phone**: +1.609.216.8913