

Algorithmic Differentiation: Compile Time vs Runtime

There are two approaches to adjoint algorithmic differentiation: compile time (which includes hand written adjoints) and runtime.

- Compile time approach: primal code is passed through an AD compiler, producing source code implementing the adjoint. This is compiled and linked with normal platform tools.
- Runtime approach: primal code is executed through a tool which builds an execution graph at runtime and then computes the adjoint from this graph.

These two approaches have different strengths:

- AD Compiler: can produce extremely efficient adjoints, but only simple input languages are understood, typically only a subset of C. Production C++ source codes are just too complex. Primal and adjoint codes must be kept in sync.
- Runtime tool: can handle production C++ codes, however there is an inevitable runtime penalty (execution time and memory) from building up the graph and interpreting it.

Many organisations prefer runtime AD tools such as dco/c++ due to their flexibility, increased developer productivity and performance: the runtime overhead of dco/c++ is typically small.

C++11 and a Meta-Program AD Compiler

C++11 introduces the keyword `auto`. For suitably formatted blocks of **straight-line code** (code with no control flow, e.g., `if`, `for`, etc.) it is possible to use `auto` to construct an execution graph at compile time. A **meta-program can convert this into an adjoint** which can be as efficient as that produced by an AD compiler. All this is done in a **single pass by the platform compiler over the code**, i.e., it is completely transparent to the user.

The target block of straight-line code must observe certain constraints:

- each active input must be “labelled” using dco/c++ API
- each intermediate variable must be of type `const auto`
- each output must be assigned only once

This idea has been implemented in a new **experimental** dco/c++ type `dco::ntr` (no tape reversal).

Example Code

This code snippet illustrates usage of the new type on a function `foo`:

```
template<typename FP, bool COMPUTE_PASSIVELY>
void foo(const FP &x1, const FP &x2, double a, FP &y) {
    // Active inputs are 'labelled'
    const auto tx1 = dco::label<COMPUTE_PASSIVELY,0>(x0);
    const auto tx2 = dco::label<COMPUTE_PASSIVELY,1>(x1);
    const auto t1 = tx1 * sqrt(tx2);
    const auto t2 = tx2 / tx1;
    const auto t3 = t2 * exp(-0.5 / a * t1 * sin(t2)) - tx1;
    y = t3 + tx1 * t1;
}
// Function can be called with 'normal' passive types
foo<double, true>(x1, x2, a, y);
// Passive computation with the new dco type
foo<dco::ntr, true>(x1, x2, a, y);
// Active computation: adjoint of y propagated
// to adjoints of x1 and x2
dco::derivative(y) = 1.0;
foo<dco::ntr, false>(x1, x2, a, y);
double a1_x1 = dco::derivative(x1);
double a1_x2 = dco::derivative(x2);
```

Benefits of the New dco/c++ Type

The type makes it much easier to produce “hand written adjoints”:

- It handles the **tedious, error-prone differentiation and adjoint propagation** of blocks of straight-line code
- Changes to these blocks are straightforward: **adjoints are always in sync**
- Users only focus on the **overall data flow reversal** and dco/c++’s tape can be used for this (if appropriate)
- Complexities of C++ types are automatically handled

Test Code 1: Euler Stepping of a Single Local Vol Path

The new type was tested on an Euler scheme for a single sample path in a local volatility model with the volatility surface expressed as a cubic spline. **None of the compilers struggled with this fairly typical finance code**. The runtime was compared to that of a hand-written adjoint (all times were scaled by the primal runtime). Note the dco/c++ tape is currently not supported in CUDA, whereas the new type is.

Test Code 2: Spherical Harmonic Function

To test the robustness of the new type we applied it to a spherical harmonic function from computational geometry. The code has 4 inputs, 1 output, is 80 lines long and has 330 edges (and numerous sub-trees)

in its binary Directed Acyclic Graph. Producing an efficient metaprogram-instantiated adjoint **entails a phenomenal amount of analysis by the compilers** and they struggled. All the compilers produced the correct answer, however only clang optimized the meta-program output fully (Linux nvcc 7.0 failed to compile due to a compiler bug). More work is needed to optimize the meta-program for such large blocks of straight-line code, and this process is already underway.

Relative Runtimes (Primal vs Adjoint) for Test Code 1

| Compiler | Primal Runtime | dco/c++ | Hand-written Adjoint | New dco/c++ type |
|--------------------|--|---------|----------------------|------------------|
| Linux | | | | |
| gcc 4.7.2 | 1 | 7.42x | 2.11x | 2.40x |
| clang 3.4 | 1 | 8.42x | 2.07x | 2.17x |
| icc 15.0.2 | 1 | 8.15x | 2.25x | 2.37x |
| nvcc 7.0 | 1 | – | 2.84x | 2.58x |
| Windows | | | | |
| icl 15.0.1 | 1 | 8.60x | 2.47x | 2.57x |
| Visual Studio 2013 | Not C++11 compliant: compilation fails | | | |
| nvcc 7.0 | Not C++11 compliant: compilation fails | | | |

Relative Runtimes (Primal vs Adjoint) for Test Code 2

| Compiler | Primal Runtime | dco/c++ | Hand-written Adjoint | New dco/c++ type |
|--------------------|--|---------|----------------------|------------------|
| Linux | | | | |
| clang 3.4 | 1 | 10.52x | 2.35x | 2.73x |
| gcc 4.7.2 | 1 | 17.73x | 1.85x | 8.00x |
| icc 15.0.2 | 1 | 17.21x | 1.76x | 11.58 |
| nvcc 7.0 | Compilation fails due to compiler bug | | | |
| Windows | | | | |
| icl 15.0.1 | 1 | 19.93x | 1.45x | 12.79x |
| Visual Studio 2013 | Not C++11 compliant: compilation fails | | | |
| nvcc 7.0 | Not C++11 compliant: compilation fails | | | |