High Performance Tape–Free Adjoint AD for C++11
Introducing dco/map, a cross–platform, accelerator ready AAD tool
Jacques du Toit (NAG), Johannes Lotz (RWTH), Klaus Leppkes and Uwe Naumann

Making Adjoint of C++ Codes
The C++ language is so complex that no AD compilers can handle it. To get an adjoint, we must write it by hand or use an operator overloading AAD tool. Handwritten adjoints are tedious to write and maintain, while operator overloading tools have a tape which must be managed. The tape is particularly problematic on accelerators since each thread (or potentially each AVX lane) needs its own. On GPUs this is just impractical.

Meta Adjoint Programming
dco/map is a tape–free operator overloading AAD tool. Since it uses operator overloading it can handle many C++11 codes. It uses meta programming to instantiate sections of the adjoint code at compile time, re–using program data on the stack. This technique is called meta adjoint programming and creates highly efficient adjoint codes.

Key Features of dco/map
• Efficient adjoints through C++11 meta adjoint programming
• First and second order tangents
• First order adjoints (second order in development)
• Thread safe by design: high performance array and scalar types for shared input data
• Unified code: a single source code can be run passively, in tangent mode or in adjoint mode
• Support for user defined custom “tape”
• Mixed precision adjoint types in development
• Supports CUDA and OpenMP (AVX51 SIMD in development)
• Supports Windows (VS2015 and higher) and Linux (gcc 4.6.0 and higher)
• Existing source will need refactoring to fit with dco/map programming model

The tool is already being used in pre–production XVA projects.

An In–House CVA Prototype
NAG has produced an in–house prototype CVA code demonstrating how to combine dco/map and CUDA to accelerate CVA applications. The code uses the G2++ model with wrong way risk and the netting set is a portfolio of swaps. Setup, calibration and final aggregation is on the CPU using tape based AD, while the Monte Carlo uses dco/map and can be run on CPU or GPU with minor code changes.

Code Samples from the CVA Prototype

```cpp
#include <dco_map/dco_map.h>

// Global functions

template<class Base> class CurveInterpolator {
    /* ... snip ... */
};

template<class Active> class G2InterestRateModel {
    /* ... snip ... */
};

template<class Active> class Functions {
    /* ... snip ... */
};

template<class Active> class G2InterestRateModel {
    /* ... snip ... */
};

// Bond price in the G2++ model: V() is a class member function
FunDecl void zeroCouponBondPrice(const Base &t, const Base &T, 
    const G2InterestRateModel<Base>::Functions<Active> sigmaCurve, 
    const dco_map::connector_t<Active> a, b, rho, x0, y0; 
    const CurveInterpolator<Base>::Functions<Active> sigmaCurve, 
    const dco_map::connector_t<Active> a, b, rho, x0, y0; 
    const dco_map::array_t<Active>d_yt, d_a1_yt; 
    const dco_map::array_t<Active,dco_map::reduction_push> curveValues;
    const G2InterestRateModel<Base>::Functions<Active> g2model(...);
    const auto rho = g2model.rho; const auto corr = sqrt(1-rho*rho);
    const auto dW1 = z1; const auto dW2 = rho*z1 + corr*z2;
    const auto z1 = Z(path,time,0);
    // Read the random numbers and correlate them
    FunDecl void makeSamplePaths(...)
```

Results for CVA Prototype
The table below gives runtimes (in ms) and memory use (in MB) for the whole (heterogeneous) application:

<table>
<thead>
<tr>
<th></th>
<th>gcc</th>
<th>clang</th>
<th>VS2015</th>
<th>nvc: total</th>
<th>nvc: GPU</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>passive</td>
<td>1,650</td>
<td>2,070</td>
<td>900</td>
<td>184</td>
<td>15</td>
<td>68</td>
</tr>
<tr>
<td>tape</td>
<td>5,024</td>
<td>5,370</td>
<td>24,590</td>
<td>370</td>
<td>117</td>
<td>176</td>
</tr>
</tbody>
</table>

Results for Local Volatility Monte Carlo Kernel
The table below gives runtimes (in ms) and memory use (in MB) of a local volatility FX basket pricer. There are 10 assets and the basket is priced with Monte Carlo using 10,000 sample paths and 360 time steps. The tape based implementation features an optimised checkpointing strategy and uses virtually the same amount of memory as a handwritten adjoint. Nevertheless the dco/map runtime is substantially less due to the meta adjoint programming.

<table>
<thead>
<tr>
<th></th>
<th>clang</th>
<th>gcc</th>
<th>VS2015</th>
<th>nvc: total</th>
<th>nvc: GPU</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>passive</td>
<td>1,461</td>
<td>1,406</td>
<td>18</td>
<td>430</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tape</td>
<td>3,031</td>
<td>3,025</td>
<td>83</td>
<td>827</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>clang</th>
<th>gcc</th>
<th>VS2015</th>
<th>nvc: total</th>
<th>nvc: GPU</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>passive</td>
<td>1,172</td>
<td>1,510</td>
<td>1,421</td>
<td>430</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tape</td>
<td>4,384</td>
<td>10,241</td>
<td>11,671</td>
<td>827</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To arrange access to dco/map please email support@nag.co.uk