Compile-Time Adjoints via Operator Overloading in C++: Applications to CPU and GPU

Jacques du Toit

Numerical Algorithms Group

Acknowledgements

This is joint work with STCE group at RWTH Aachen.

Adjoint Model

$$F: \underset{\uparrow}{\mathbf{x}} \xrightarrow{f} \underset{\uparrow}{\mathbf{x}_1} \xrightarrow{g} \underset{\uparrow}{\mathbf{x}_2} \xrightarrow{h} \underset{\uparrow}{\mathbf{y}}$$
$$\mathbb{R}^n \qquad \mathbb{R}^{m_1} \qquad \mathbb{R}^{m_2} \qquad \mathbb{R}^m$$

Jacobian vector product is

$$\nabla F \cdot \mathbf{z} = \left[\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right]\mathbf{z} = \underset{\uparrow}{\left[\frac{\partial h}{\partial \mathbf{x}_2}\right]}\left(\underset{\uparrow}{\left[\frac{\partial g}{\partial \mathbf{x}_1}\right]}\left(\underset{\uparrow}{\left[\frac{\partial f}{\partial \mathbf{x}}\right]}\ \underset{\uparrow}{\mathbf{z}}\right)\right)$$
$$\mathbb{R}^{m_2 \times m} \qquad \mathbb{R}^{m_2 \times m_1} \qquad \mathbb{R}^{m_1 \times n} \qquad \mathbb{R}^n$$

where $\mathbf{z} \in \mathbb{R}^n$. Get whole Jacobian in n evaluations (bumping).

Adjoint Model

$$F: \underset{\uparrow}{\mathbf{x}} \xrightarrow{f} \underset{\uparrow}{\mathbf{x}_1} \xrightarrow{g} \underset{\uparrow}{\mathbf{x}_2} \xrightarrow{h} \underset{\uparrow}{\mathbf{y}}$$
$$\mathbb{R}^n \qquad \mathbb{R}^{m_1} \qquad \mathbb{R}^{m_2} \qquad \mathbb{R}^m$$

Matrix adjoint is transpose. *Adjoint model* $F_{(1)}$ of F is

$$F_{(1)}(\mathbf{x}, \mathbf{z}) = \nabla F(\mathbf{x})^T \, \mathbf{z}$$

where $\mathbf{z} \in \mathbb{R}^m$. Implementing this model gives

$$\underset{\uparrow}{\left[\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right]^T} \mathbf{z} = \underset{\uparrow}{\left[\frac{\partial f}{\partial \mathbf{x}}\right]^T} \left(\underset{\uparrow}{\left[\frac{\partial g}{\partial \mathbf{x}_1}\right]^T} \left(\underset{\uparrow}{\left[\frac{\partial h}{\partial \mathbf{x}_2}\right]^T} \underset{\uparrow}{\mathbf{z}} \right) \right)$$
$$\mathbb{R}^{n \times m_1} \qquad \mathbb{R}^{m_1 \times m_2} \qquad \mathbb{R}^{m_2 \times m} \qquad \mathbb{R}^m$$

- Entire Jacobian in m calls of model *regardless of* n!
- $F_{(1)}$ requires at most 5 times more flops than F
- **BUT** natural order of calculation is *backwards*!

Computing Adjoints

- Useful definition of adjoint: adjoint $x_{(1)}$ of x is

$$x_{(1)} = \frac{dt}{dx}$$

for some auxiliary scalar variable t.

- So for some sequence of mappings

$$\mathbf{x} \mapsto \mathbf{x_1} \mapsto \mathbf{x_2} \mapsto y$$

we have

$$\begin{aligned}\mathbf{x_{(1)}} = \frac{\partial t}{\partial \mathbf{x}} &= \frac{\partial \mathbf{x_1}}{\partial \mathbf{x}} \frac{\partial t}{\partial \mathbf{x_1}} \\ &= \frac{\partial \mathbf{x_1}}{\partial \mathbf{x}} \frac{\partial \mathbf{x_2}}{\partial \mathbf{x_1}} \frac{\partial t}{\partial \mathbf{x_2}} \\ &= \frac{\partial \mathbf{x_1}}{\partial \mathbf{x}} \frac{\partial \mathbf{x_2}}{\partial \mathbf{x_1}} \frac{\partial y}{\partial \mathbf{x_2}} \frac{\partial t}{\partial y} = \frac{\partial y}{\partial \mathbf{x}} y_{(1)}\end{aligned}$$

Computing Adjoints

Consider a function

```
double f(double x, double y, double z)
{
  double a = 4.7*x*y - z;
  double b = a*a*z - x;
  double c = x*sin(b) + 3.1*a;
  return c;
}
```

Suppose we want an adjoint version of this

```
double f1(double x, double &x1, double y, double &y1,
                    double z, double &z1, double c1)
```

We can write this by hand

Computing Adjoints

```
double f1(double x, double &x1, double y, double &y1,
                    double z, double &z1, double c1)
{
  double a = 4.7*x*y - z;
  double b = a*a*z - x;
  double c = x*sin(b) + 3.1*a;
  double x1 += sin(b)*c1;          // dc_dx * c1
  double b1 += x*cos(b)*c1;        // dc_db * c1
  double a1 += 3.1*c1;             // dc_da * c1

  a1 += 2*a*z*b1;                  // db_da * b1
  z1 += a*a*b1;                    // db_dz * b1
  x1 += -1*b1;                     // db_dx * b1

  x1 += 4.7*y*a1;                  // da_dx * a1
  y1 += 4.7*x*a1;                  // dx_dy * a1
  z1 += -1*a1;                     // dx_dz * a1
}
```

Tedious, but not difficult. Good use of a quant's time?

Computing Adjoints

Now consider something like

```
void f(double &z) {
  z = exp(z);
  z = sin(z);
}
```

Computing Adjoints

```
void f1(double &z, double &z1) {
  z = exp(z);           // a = exp(z);
  z = sin(z);           // b = sin(a);
}
```

Computing Adjoints

```
void f1(double &z, double &z1) {
  z = exp(z);          // a = exp(z);
  z = sin(z);          // b = sin(a);
  z1 = cos(z)*z1;
}
```

Computing Adjoints

```
void f1(double &z, double &z1) {
  z = exp(z);           // a = exp(z);
  z = sin(z);           // b = sin(a);
  z1 = cos(z)*z1;       // a1 = cos(a)*b1;
  // = cos(b)*z1; WRONG
}
```

Computing Adjoints

```
void f1(double &z, double &z1) {
  z = exp(z);           // a = exp(z);
  push(z);
  z = sin(z);           // b = sin(a);
  pop(z);
  z1 = cos(z)*z1;       // a1 = cos(a)*b1;
}
```

Computing Adjoints

```
void f1(double &z, double &z1) {
  z = exp(z);           // a = exp(z);
  push(z);
  z = sin(z);           // b = sin(a);
  pop(z);
  z1 = cos(z)*z1;       // a1 = cos(a)*b1;
  z1 = exp(z)*z1;
}
```

Computing Adjoints

```
void f1(double &z, double &z1) {
  z = exp(z);           // a = exp(z);
  push(z);
  z = sin(z);           // b = sin(a);
  pop(z);
  z1 = cos(z)*z1;       // a1 = cos(a)*b1;
  z1 = exp(z)*z1;       // z1 = exp(z)*a1;
  // = exp(a)*z1; //WRONG
}
```

Computing Adjoints

```
void f1(double &z, double &z1) {
  push(z);
  z = exp(z);          // a = exp(z);
  push(z);
  z = sin(z);          // b = sin(a);
  pop(z);
  z1 = cos(z)*z1;      // a1 = cos(a)*b1;
  pop(z);
  z1 = exp(z)*z1;      // z1 = exp(z)*a1;
}
```

Computing Adjoints

```
void f1(double &z, double &z1) {
  push(z);
  z = exp(z);          // a = exp(z);
  push(z);
  z = sin(z);          // b = sin(a);
  pop(z);
  z1 = cos(z)*z1;      // a1 = cos(a)*b1;
  pop(z);
  z1 = exp(z)*z1;      // z1 = exp(z)*a1;
  // return value is now wrong:
  // it's input z, not b
```

Adjoints for general code not easy to compute

- It can become tricky, depending on underlying code
- Can be managed if underlying code under your control
- And still need to handle data flow reversal problem

Computing Adjoints

Options for making adjoint code:

1. Write by hand
   - best runtime performance
   - laborious to write and highly error prone, 2 sets of source
2. Use AD compiler to generate adjoint code
   - good runtime performance, less laborious
   - very restrictive, 2 sets of source
3. Use runtime (tape based) operator overloading tool (e.g. `dco/c++`)
   - flexible and 1 set of source
   - runtime overheads
4. ?

AD via Tape Based Overloading

- ```
  double f(double x, double y, double z) {
    double a = 4.7*x*y - z;
    double b = a*a*z - x;
    return x*sin(b) + 3.1*a;
  }
  ```
- Templatise the code

  ```
  template<typename AT>
  AT f(AT x, AT y, AT z) {
    AT a = 4.7*x*y - z;
    AT b = a*a*z - x;
    return x*sin(b) + 3.1*a;
  }
  ```
- Call function f with AD types, e.g.

  ```
  f< dco::gaIs<double>::type >(x, y, z);
  ```
- "Driver code" in `main()` to register active inputs and output(s)
- AD types store intermediates to tape, user then interprets tape
  ```

AD via Tape Based Overloading

Main advantages of tape based adjoints:

- Can handle any C++ source code
- "Bullet proof" – if tool is well designed, it's hard to make a mistake
- Prioritise user productivity
- Only one source – the primal source
- Performance (for good tools e.g. `dco/c++`) is "good enough"

Main disadvantages

- Memory use can grow quickly – requires user intervention to control
- Performance is only "good enough"
  - Tape based AD of general codes always has some degree of unstructured memory access
  - Good tools minimise this, but can't eliminate it
  - Performance always slower than handwritten adjoint

AD via Tape Based Overloading

Can't really do tape-based AD on GPUs

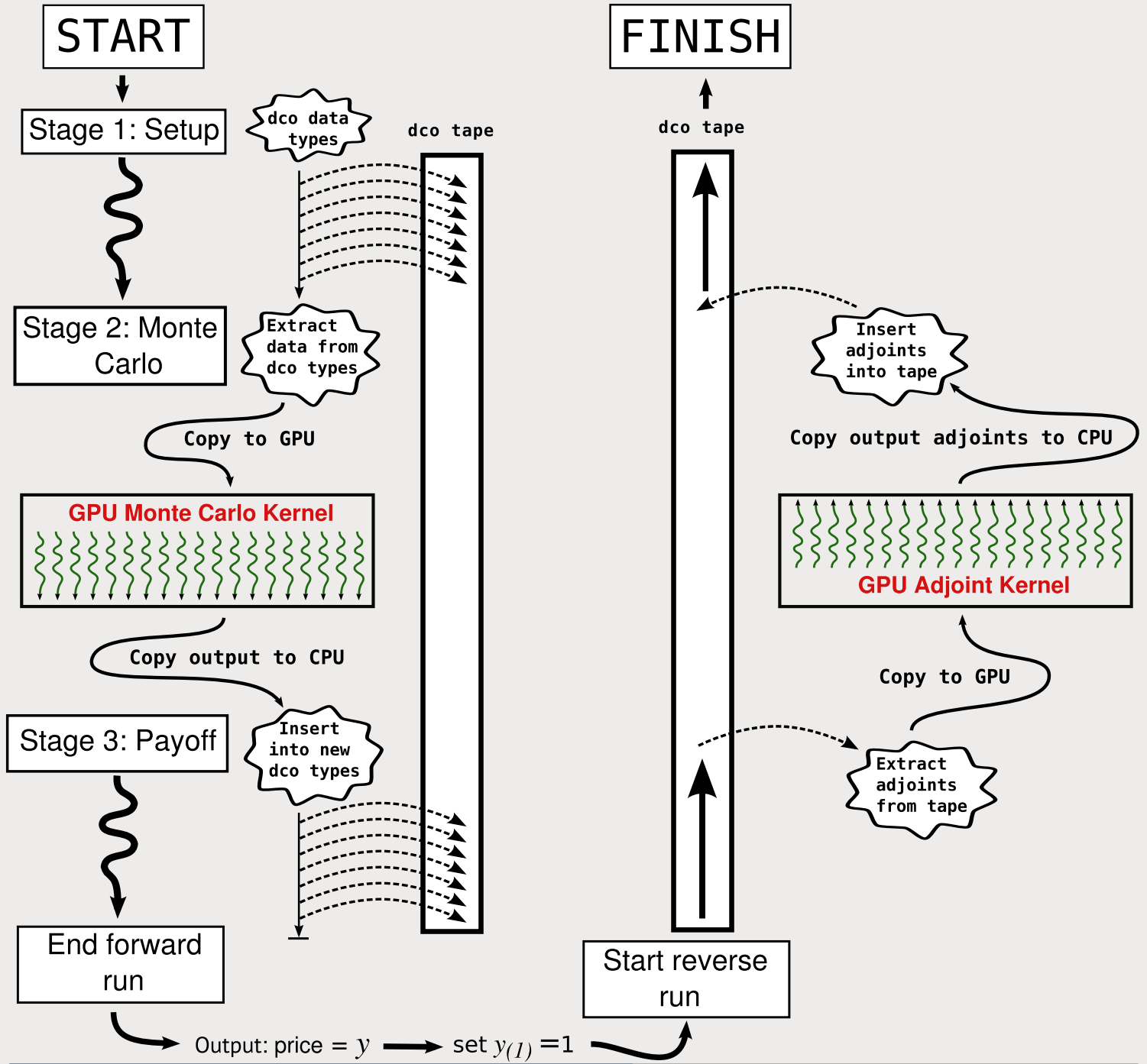
- 50,000 threads in flight at once, only 12GB RAM

Why would you want to combine AD with GPU?

- Real time risk management

Test Code: 10 factor local volatility model, priced 10 asset FX basket option by Monte Carlo

- Roughly 450 inputs (mainly market implied vol quotes)
- Local volatility slices represented as cubic splines
- Overall runtime: 550ms
- Forward Monte Carlo kernel: 18.5ms
- Handwritten Adjoint Monte Carlo kernel: 90ms


START
Stage 1: Setup
dco data types
dco tape
Stage 2: Monte Carlo
Extract data from dco types
Copy to GPU
GPU Monte Carlo Kernel
Copy output to CPU
Stage 3: Payoff
Insert into new dco types
End forward run
Output: price = y
set $y_{(1)}$ =1
FINISH
dco tape
Insert adjoints into tape
Copy output adjoints to CPU
GPU Adjoint Kernel
Copy to GPU
Extract adjoints from tape
Start reverse run

Normal Cubic Spline Code

```
template<typename FP>
__device__ void spline(int n, FP *lamda, FP *c,
                       FP x, FP &s) {
    int j = ... // Find correct place in arrays
    FP k1 = lamda[j+1]; FP k2 = lamda[j+2];
    FP k3 = lamda[j+3]; FP k4 = lamda[j+4];
    FP k5 = lamda[j+5]; FP k6 = lamda[j+6];
    FP c1 = c[j];       FP c2 = c[j+1];
    FP c3 = c[j+2];

    FP e2 = x - k2;     FP e3 = x - k3;
    FP e4 = k4 - x;     FP e5 = k5 - x;
    c1 = ((x-k1)*c2+e4*c[j])/(k4-k1);
    FP c5 = (e2*c3+e5*c2)/(k5-k2);
    FP c7 = (e3*c[j+3]+(k6-x)*c3)/(k6-k3);
    FP c4 = (e2*c5+e4*c1)/(k4-k2);
    FP c6 = (e3*c7+e5*c5)/(k5-k3);
    s = (e3*c6+e4*c4)/(k4-k3);
}
```

Adjoint Cubic Spline Code – 1

```
template<typename FP>
__device__ void spline_b(int n, FP *lamda,
                         FP *lamdab, FP *c, FP *cb,
                         FP x, FP &xb, FP sb) {
    int j = ... // Find correct place in arrays
    FP k1 = lamda[j+1];      FP k2 = lamda[j+2];
    FP k3 = lamda[j+3];      FP k4 = lamda[j+4];
    FP k5 = lamda[j+5];      FP k6 = lamda[j+6];
    FP e2 = x - k2;          FP e3 = x - k3;
    FP e4 = k4 - x;          FP e5 = k5 - x;
    FP c2 = c[j+1];          FP c3 = c[j+2];
    FP invk4mk1 = 1/(k4-k1);
    FP invk5mk2 = 1.0f/(k5-k2);
    FP invk6mk3 = 1.0f/(k6-k3);
    FP invk4mk2 = 1.0f/(k4-k2);
    FP invk5mk3 = 1.0f/(k5-k3);
    FP c1 = ((x-k1)*c2+e4*c[j])*invk4mk1;
    FP c5 = (e2*c3+e5*c2)*invk5mk2;
    FP c7 = (e3*c[j+3]+(k6-x)*c3)*invk6mk3;
```

Adjoint Cubic Spline Code – 2

```
FP c7 = (e3*c[j+3]+(k6-x)*c3)*invk6mk3;
FP c4 = (e2*c5+e4*c1)*invk4mk2;
FP c6 = (e3*c7+e5*c5)*invk5mk3;
FP tempb = sb/(k4-k3);
FP tempb0 = -((e3*c6+e4*c4)*tempb/(k4-k3));
FP c6b = e3*tempb;
FP c4b = e4*tempb;
FP tempb1 = c6b*invk5mk3;
FP tempb7 = -((e3*c7+e5*c5)*tempb1*invk5mk3);
FP c7b = e3*tempb1;
FP tempb3 = c4b*invk4mk2;
FP c5b = e2*tempb3 + e5*tempb1;
FP tempb5 = -((e2*c5+e4*c1)*tempb3*invk4mk2);
FP c1b = e4*tempb3;
FP tempb2 = c7b*invk6mk3;
FP e3b = c7*tempb1 + c[j+3]*tempb2 +
                                    c6*tempb;
FP tempb8 = -((e3*c[j+3]+(k6-x)*c3) *
                                tempb2*invk6mk3);
```

Adjoint Cubic Spline Code – 3

```
FP k3b = -tempb7 - e3b - tempb8 - tempb0;
FP k6b = tempb8 + c3*tempb2;
FP tempb9 = c5b*invk5mk2;
FP e5b = c2*tempb9 + c5*tempb1;
FP e2b = c3*tempb9 + c5*tempb3;
FP c3b = e2*tempb9 + (k6-x)*tempb2;
FP tempb10 = -((e2*c3+e5*c2) *
                              tempb9*invk5mk2);
FP k5b = tempb10 + e5b + tempb7;
FP k2b = -tempb10 - e2b - tempb5;
FP tempb4 = c1b*invk4mk1;
FP e4b = c1*tempb3 + c[j]*tempb4 + c4*tempb;
FP xb += c2*tempb4 - c3*tempb2;
FP c2b = (x-k1)*tempb4 + e5*tempb9;
FP tempb6 = -(((x-k1)*c2+e4*c[j]) *
                              tempb4*invk4mk1);
FP k4b = tempb5 + e4b + tempb6 + tempb0;
FP k1b = -tempb6 - c2*tempb4;
```

Adjoint Cubic Spline Code – 4

```
    // Update adjoint
    xb += e2b - e4b + e3b - e5b;
    cb[j] += e4*tempb4;    cb[j+1] += c2b;
    cb[j+2] += c3b;        cb[j+3] += e3*tempb2;
    lamdab[j+6] += k6b;    lamdab[j+5] += k5b;
    lamdab[j+4] += k4b;    lamdab[j+3] += k3b;
    lamdab[j+2] += k2b;    lamdab[j+1] += k1b;
}
```

Is there not a Better Way?

C++11

C++ is not one, but three languages

- Procedural language of C
- Object oriented language
- Meta-programming language (templates)

Key here is meta-programming language

- Meta-program is run at compile time, producing source code, which is then compiled
- Is Turing complete

For blocks of straight-line code (e.g. spline), producing an adjoint code is completely algorithmic

- Construct DAG of computation
- Reverse DAG and compute adjoint

C++11

Meta-programming language is Turing complete!

> *Can we not write a meta-program which would create the adjoint for a block of straight line code?*

Just because it's Turing complete doesn't mean it's simple or **practical** to do! Template meta-programming language is famously obscure.

Before C++11, answer is heavily weighted towards no

With C++11 that changes

The Delights of "auto"

C++11 introduces the keyword **auto**

```
auto n = 10;          // n is int
auto x = 10.0;        // x is double
auto y = foo(n, x);   // y is whatever foo returns
```

Tells the compiler to figure out the type

This is a big deal!

- C++ types are incredibly powerful
- Types are the fundamental blocks in the template meta-programming language
- We can now play with types while keeping the source code flexible

Implemented this in a new `dco/c++` type (so this all integrates with `dco/c++` system)

Spline Code with New Type

```
template<typename AT>
void spline(int n, const AT *lamda, const AT *c,
                        const AT & x, AT &s)
{
  int j = ... // Find correct place in arrays
  const auto k1 = lamda[j+1]; const auto k2 = lamda[j+2];
  const auto k3 = lamda[j+3]; const auto k4 = lamda[j+4];
  const auto k5 = lamda[j+5]; const auto k6 = lamda[j+6];
  const auto c0 = c[j];       const auto c2 = c[j+1];
  const auto c3 = c[j+2];

  const auto e2 = x - k2;     const auto e3 = x - k3;
  const auto e4 = k4 - x;     const auto e5 = k5 - x;
  const auto c1 = ((x-k1)*c2+e4*c0)/(k4-k1);
  const auto c5 = (e2*c3+e5*c2)/(k5-k2);
  const auto c7 = (e3*c[j+3]+(k6-x)*c3)/(k6-k3);
  const auto c4 = (e2*c5+e4*c1)/(k4-k2);
  const auto c6 = (e3*c7+e5*c5)/(k5-k3);
  s = (e3*c6+e4*c4)/(k4-k3);
}
```

Final assignment triggers DAG reversal

Going Beyond Basic Blocks

Turns out one can go beyond basic blocks

- Nested scoping blocks
- Function calls
- Loops
- If-Else statements

Possible to handle more realistic codes.

Ultimate goal:

- One source from which primal and adjoint codes are instantiated
- Handle realistic codes
- Source must look “natural”
- Runtime performance of adjoint close to handwritten
- Compilation time and compiler memory use to remain reasonable

This is a pretty tough list!

Going Beyond Basic Blocks

What happens to tape/checkpoints?

- Tool creates no tape and stores virtually nothing
- In theory, tool can handle any code through "recompute all" strategy
- In practice, user would checkpoint certain outputs
- Tool provides a flexible and clean interface for this

Test Code:

- Simplified version of FX Basket code
- Only one asset
- Local vol $\sigma(t, x) \equiv \sigma(x)$ and is represented as spline with 19 knots

CPU results below all serial, GPU results parallel, so don't compare

Results of Test Code

Linux			
	clang	gcc	nvcc (80x)
passive	108	121	131
handwritten	248 (2.3x)	263 (2.2x)	581 (4.4x)
New Tool	243 (2.3x)	285 (2.4x)	766 (5.9x)
`dco/c++` tape	554 (5.1x)	865 (7.2x)	N/A
Windows			
	clang	VS2013	Intel2015
passive	110	120	110
handwritten	246 (2.2x)	247 (2.1x)	184 (1.7x)
New Tool	391 (3.6x)	901 (7.5x)	404 (3.6x)
`dco/c++` tape	686 (6.2x)	1,224 (10x)	1,149 (10x)

nag®

```
template<typename AD>
void makePaths(int N, int M, const AD &x0, const AD & dt,
        const AD &r, int n, const AD knots[], const AD cs[],
        const basetype Z[], AD x[], basetype ckpoint[] )
{
for(int p = 0; p < N; p++) {
  AD sigma;
  dco_ntr::loop_var<AD> xi(x0), xii(0);

  DCO_FOR(i, 0, M-1)
  {
    DCO_CKP_CALL( evalSpline<AD>(n, knots, cs, xi, sigma) );

    xii = xi + (r - 0.5*sigma*sigma)*dt
                                  + sigma*sqrt(dt)*Z[i+p*M];
    xi = xii;
  }
  DCO_FOR_STORE_CKP(i) {
    ckpoint[i+p*M] = dco_ntr::value(xi);
  }
  DCO_FOR_LOAD_CKP(i) {
    dco_ntr::value(xi) = ckpoint[i+p*M];
    dco_ntr::derivative(sigma)=0;
    evalSpline<AD>(n, knots, cs, xi, sigma);
  }
  DCO_ENDFOR;

  x[p] = xi;
}
}
```

Final Words

Use cases for new tool:

- People who can't afford memory overheads of taping
- People using accelerators
- People who want to optimise compute kernels
  - 90% of runtime in 10% of code
  - Use the tool for the 10% of code, use `dco/c++` for the rest

Some work still remains

- Interface formalisation, documentation, testing, etc

We're trying to guage interest from industry. If this sounds like useful tool, please talk to us.

Thank you

Questions?