Adjoint on GPU: C++11 takes the "hand" out of Handwritten Adjoint

Jacques du Toit, Klaus Leppkes and Johannes Lotz

Numerical Algorithms Group and STCE Aachen University
Introduction
Introduction

- Local volatility FX basket option code as P.O.C of GPUs in finance
- Basket on 10 currency pairs, local vol computed from market quotes
- Over 450 input parameters
- Pricing by Monte Carlo, which suits GPU very well
Introduction

- Local volatility FX basket option code as P.O.C of GPUs in finance
- Basket on 10 currency pairs, local vol computed from market quotes
- Over 450 input parameters
- Pricing by Monte Carlo, which suits GPU very well

Question then how to compute 1st order sensitivities

- Key quantities for hedging
- Usual approach is finite differences: bump and revalue
- Adjoint methods are alternative
- Algorithmic differentiation (AD) is a way of creating an adjoint
Our Goal

Wanted to create an adjoint of FX basket option code, and still wanted this code to use GPU
AD in a Nutshell
What is Algorithmic Differentiation?

It’s a way to compute

\[
\begin{align*}
\frac{\partial}{\partial x_1} F(x_1, x_2, x_3, \ldots) & \quad \frac{\partial}{\partial x_2} F(x_1, x_2, x_3, \ldots) \\
\frac{\partial}{\partial x_3} F(x_1, x_2, x_3, \ldots) & \quad \ldots
\end{align*}
\]

where \( F \) is given by a computer program, e.g.

\[
\begin{align*}
\text{if}(x_1 < x_2) & \text{ then} \\
F & = x_1 \times x_1 + x_2 \times x_2 + x_3 \times x_3 + \ldots \\
\text{else} & \\
F & = x_1 + x_2 + x_3 + \ldots \\
\text{endif}
\end{align*}
\]
Algorithmic Differentiation in a Nutshell

Computers can only add, subtract, multiply and divide numbers.

- Computer program implementing a model is many of these basic operations strung together
- Elementary to compute the derivatives of these
- Chain rule + basic derivatives = program derivative

AD comes in two flavours: Tangent Model and Adjoint Model
Tangent Model

\[ F: \mathbb{R}^n \xrightarrow{f} \mathbb{R}^{m_1} \xrightarrow{g} \mathbb{R}^{m_2} \xrightarrow{h} \mathbb{R}^m \]
Tangent Model

\[ F : \mathbb{R}^n \xrightarrow{f} \mathbb{R}^{m_1} \xrightarrow{g} \mathbb{R}^{m_2} \xrightarrow{h} \mathbb{R}^m \]

Apply Chain rule

\[
\frac{\partial y}{\partial x(j)} = \frac{\partial h}{\partial x_2} \left( \frac{\partial g}{\partial x_1} \left( \frac{\partial f}{\partial x(j)} \right) \right)
\]

\[
\left[ \mathbb{R}^{m \times m_2} \right] \left[ \mathbb{R}^{m_2 \times m_1} \right] \left[ \mathbb{R}^{m_1 \times 1} \right]
\]
Consider following code:

```c
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 to compute $\frac{\partial f}{\partial x}$
This is relatively straightforward:

```c
double f_x(double x, double y, double z, double &df_dx)
{
    double a = 4.7 * x * y - z;
    double da_dx = 4.7 * y;
    double b = a * a * z - x;
    double db_da = 2 * a * z;
    double db_dx = -1.0;
    double c = x * sin(b) + 3.1 * a;
    double dc_dx = sin(b);
    double dc_db = x * cos(b);
    double dc_da = 3.1;
    double df_f_x = dc_dx + dc_da * da_dx + dc_db * (db_dx + db_da * da * dx);
    return c;
}
```
Tangent Model

- We want ability to compute $f_x, f_y$ or $f_z$ not just $f_x$
- Several ways to do this (e.g. pass in a flag)
- But consider the following code:

```c
double g(double x, double y, double z)
{
    double a = x*y, b=x-z, c=z*z*x;
    return f(a, b, c);
}
```

$$\frac{\partial g}{\partial x} = f_a \frac{\partial a}{\partial x} + f_b \frac{\partial b}{\partial x} + f_c \frac{\partial c}{\partial x}$$

- Want one solution that works in all cases, so should make $F$ returning

$$F(x, dx) = \nabla f(x) \, dx = \left( \frac{\partial f}{\partial x} \right) \, dx$$

- Then to get $f_x$, simply call $F$ with $dx[] = \{1, 0, 0\}$
Tangent Model of AD

Tangent model $F^{(1)}$ of $F$ is

$$F^{(1)}(x, z) = \nabla F(x) \cdot z$$

where $\nabla F \in \mathbb{R}^{m \times n}$ is Jacobian of $F$ and $z \in \mathbb{R}^n$.

- Getting $\nabla F$ needs $n$ calls to $F^{(1)}$ setting $z$ as Cartesian basis vectors in $\mathbb{R}^n$.
- Runtime for $\nabla F$ roughly $n$ times longer than $F$.
- Same computational complexity as finite differences.
- Problem: often $n$ very big!
What do we Mean by Big

**Figure:** Sensitivity of drag coefficient to each mesh point on surface of car. Gradient has $n = 5,500,000$ entries. Tangent model (or finite differences) would take roughly 5 years. (Towara & Naumann, 2013).
Adjoint Model

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

$$x^{(1)} = x^{(1)} + \nabla F(x)^T z$$

where $z \in \mathbb{R}^m$. 

$F : \mathbb{R}^n \xrightarrow{f} \mathbb{R}^{m_1} \xrightarrow{g} \mathbb{R}^{m_2} \xrightarrow{h} \mathbb{R}^m$
Adjoint Model

\[ F : \mathbf{x} \xrightarrow{f} \mathbf{x}_1 \xrightarrow{g} \mathbf{x}_2 \xrightarrow{h} \mathbf{y} \]

\[ \mathbb{R}^n \xrightarrow{\mathbb{R}^m} \mathbb{R}^{m_1} \xrightarrow{\mathbb{R}^{m_2}} \mathbb{R}^m \]

Matrix adjoint is transpose. *Adjoint model* \( F^{(1)} \) of \( F \) is

\[ \mathbf{x}^{(1)} = \mathbf{x}^{(1)} + \nabla F(\mathbf{x})^T \mathbf{z} \]

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

\[
\begin{pmatrix}
\frac{\partial y}{\partial x}
\end{pmatrix}^T \mathbf{z} = \begin{pmatrix}
\frac{\partial f}{\partial x}
\end{pmatrix}^T \begin{pmatrix}
\frac{\partial g}{\partial x_1}
\end{pmatrix}^T \begin{pmatrix}
\frac{\partial h}{\partial x_2}
\end{pmatrix}^T \begin{pmatrix}
\mathbf{z}
\end{pmatrix}^T
\]

\[ \mathbb{R}^{n \times m_1} \xrightarrow{\mathbb{R}^{m_1 \times m_2}} \mathbb{R}^{m_2 \times m} \xrightarrow{\mathbb{R}^m} \]
Matrix adjoint is transpose. \textit{Adjoint model} $F^{(1)}$ of $F$ is

$$x^{(1)} = x^{(1)} + \nabla F(x)^T z$$

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

$$
\begin{bmatrix}
\frac{\partial y}{\partial x}
\end{bmatrix}^T z = 
\begin{bmatrix}
\frac{\partial f}{\partial x}
\end{bmatrix}^T 
\begin{bmatrix}
\frac{\partial g}{\partial x_1}
\end{bmatrix}^T 
\begin{bmatrix}
\frac{\partial h}{\partial x_2}
\end{bmatrix}^T z
\end{bmatrix}

\begin{bmatrix}
\mathbb{R}^{n \times m_1} \\
\mathbb{R}^{m_1 \times m_2} \\
\mathbb{R}^{m_2 \times m}
\end{bmatrix}
\begin{bmatrix}
\mathbb{R}^m
\end{bmatrix}
$$

- Entire Jacobian in $m$ calls of model \textit{regardless of} $n$
**Adjoint Model**

\[ F : \mathbf{x} \xrightarrow{f} \mathbf{x}_1 \xrightarrow{g} \mathbf{x}_2 \xrightarrow{h} \mathbf{y} \]

\[ \mathbb{R}^n \quad \mathbb{R}^{m_1} \quad \mathbb{R}^{m_2} \quad \mathbb{R}^m \]

Matrix adjoint is transpose. **Adjoint model** \( F^{(1)} \) of \( F \) is

\[ \mathbf{x}^{(1)} = \mathbf{x}^{(1)} + \nabla F(\mathbf{x})^T \mathbf{z} \]

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

\[
\left[ \begin{array}{c}
\frac{\partial y}{\partial x}^T \\
\frac{\partial f}{\partial x}^T \\
\frac{\partial g}{\partial x_1}^T \\
\frac{\partial h}{\partial x_2}^T \\
\end{array} \right] \mathbf{z} = \left[ \begin{array}{c}
\frac{\partial f}{\partial x}^T \\
\frac{\partial g}{\partial x_1}^T \\
\frac{\partial h}{\partial x_2}^T \\
\end{array} \right] \left( \left[ \begin{array}{c}
\mathbf{z}^T \\
\end{array} \right] \right)
\]

\[ \mathbb{R}^{n \times m_1} \quad \mathbb{R}^{m_1 \times m_2} \quad \mathbb{R}^{m_2 \times m} \quad \mathbb{R}^m \]

- Entire Jacobian in \( m \) calls of model *regardless of* \( n \)
- Order of complexity: input space to output space
**Adjoint Model**

\[
F : \begin{array}{cccc}
x & \rightarrow & x_1 & \rightarrow & x_2 & \rightarrow & y \\
\uparrow & & \uparrow & & \uparrow & & \uparrow \\
\mathbb{R}^n & & \mathbb{R}^{m_1} & & \mathbb{R}^{m_2} & & \mathbb{R}^m
\end{array}
\]

Matrix adjoint is transpose. *Adjoint model* \( F^{(1)} \) of \( F \) is

\[
x^{(1)} = x^{(1)} + \nabla F(x)^T z
\]

where \( z \in \mathbb{R}^m \). Implementing this model gives

\[
\begin{bmatrix}
\frac{\partial y}{\partial x}
\end{bmatrix}^T z = \begin{bmatrix}
\frac{\partial f}{\partial x}
\end{bmatrix}^T \left( \begin{bmatrix}
\frac{\partial g}{\partial x_1}
\end{bmatrix}^T \left( \begin{bmatrix}
\frac{\partial h}{\partial x_2}
\end{bmatrix}^T z \right) \right)
\]

\[
\begin{bmatrix}
\mathbb{R}^{n \times m_1} \\
\mathbb{R}^{m_1 \times m_2} \\
\mathbb{R}^{m_2 \times m}
\end{bmatrix}
\]

- Entire Jacobian in \( m \) calls of model *regardless of* \( n \)
- Order of complexity: input space to output space
- \( F^{(1)} \) requires at most 5 times more flops than \( F \)
Adjoint Model

\[ F : \mathbb{R}^n \xrightarrow{f} \mathbb{R}^{m_1} \xrightarrow{g} \mathbb{R}^{m_2} \xrightarrow{h} \mathbb{R}^m \]

Matrix adjoint is transpose. **Adjoint model** \( F^{(1)} \) of \( F \) is

\[ x^{(1)} = x^{(1)} + \nabla F(x)^T z \]

where \( z \in \mathbb{R}^m \). Implementing this model gives

\[
\begin{bmatrix}
\frac{\partial y}{\partial x}
\end{bmatrix}^T z = \begin{bmatrix}
\frac{\partial f}{\partial x}
\end{bmatrix}^T \left( \begin{bmatrix}
\frac{\partial g}{\partial x_1}
\end{bmatrix}^T \left( \begin{bmatrix}
\frac{\partial h}{\partial x_2}
\end{bmatrix}^T z \right) \right)
\]

- Entire Jacobian in \( m \) calls of model *regardless of* \( n \)
- Order of complexity: input space to output space
- \( F^{(1)} \) requires at most 5 times more flops than \( F \)
- Order of computation for adjoint is *backwards*
Writing Adjoint by Hand

Recall our function:

```c
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 to compute an adjoint of this
Writing Adjoint Code by Hand

We can take as definition

\[ x(1) = \frac{dt}{dx} \quad \text{so that} \quad x(1) = \frac{dc}{dx} \frac{dt}{dc} = \frac{dc}{dx} c(1) \]

```c
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;
    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
}
```
To compute adjoint of

\[ F : \mathbb{R}^n \xrightarrow{f} \mathbb{R}^{m_1} \xrightarrow{g} \mathbb{R}^{m_2} \xrightarrow{h} \mathbb{R}^m \]

we have to work backwards: solve a data flow reversal problem

- Either store all intermediate outputs during forward run (tape)
- Or store just enough to recompute intermediate outputs during backward run (checkpoints)
Creating Adjoint Codes

Options for creating adjoint code:

1. Write by hand: the example looked simple, in real codes it’s much more complicated!
   - best runtime performance
   - laborious, highly error prone, 2 sets of source

2. Use an AD compiler
   - good runtime performance, less laborious
   - very restrictive, 2 sets of source

3. Use a runtime tool (e.g. dco/c++)
   - flexible, 1 set of source
   - runtime overheads

No AD tool support for GPUs
Local volatility FX basket option
Local Volatility FX Basket Option

Standard multidimensional local vol model

\[
\frac{dS_t^{(i)}}{S_t^{(i)}} = \left( r_d - r_f^{(i)} \right) dt + \sigma^{(i)}(S_t^{(i)}, t) dW_t^{(i)}
\]

with \( \langle W^{(i)}, W^{(j)} \rangle_t = \rho^{(i,j)} t \).
Local Volatility FX Basket Option

Standard multidimensional local vol model

\[
\frac{d S_t^{(i)}}{S_t^{(i)}} = (r_d - r_f^{(i)}) dt + \sigma^{(i)} \left( S_t^{(i)}, t \right) dW_t^{(i)}
\]

with \( \langle W^{(i)}, W^{(j)} \rangle_t = \rho^{(i,j)} t \).

Computing local vol surfaces:

- Heuristic procedure based on fitting 1D cubic splines, interpolating values, fitting new splines, etc.
- Final output: at each Monte Carlo time step have a 20 knot cubic spline in \( K \) with linear extrapolations in the wings

With 360 Euler time steps, have total of \( 360 \times 10 \) 1D splines defining the 10 local vol surfaces
Pricing Code
Pricing Code

Pricing code structured as follows:

- Create local vol splines on host, copy (with other data) to GPU
- Generate random numbers on GPU
- Generate all Monte Carlo sample paths on GPU
- Copy final values of paths to CPU, compute option price
**GPU Monte Carlo Kernel**

Performance is good
- 10,000 sample paths and 360 Euler time steps
- GPU Monte Carlo kernel takes 14.5ms (single precision)
- PCIe bus traffic takes 0.5ms
Designing an Adjoint Code
Basket Option Code

We used \texttt{dco/c++} on the host and initially a handwritten GPU adjoint kernel spliced together with \texttt{dco/c++}'s \textit{external function interface}.

- To use \texttt{dco/c++} replace \texttt{double} with \texttt{dco::als<double>}
- \texttt{dco/c++} records all intermediate values to a tape
- External function interface inserts gaps into this tape (stops recording)
- User provides functions which fill this gap: one function for forward (passive) run, one function implementing adjoint
Data Dependencies

Recall that for $c \equiv c(x)$ we have

$$x(1) = \frac{dt}{dx} = \frac{dc}{dx} \frac{dt}{dc} = \frac{dc}{dx} c(1)$$
Data Dependencies

Recall that for \( c \equiv c(x) \) we have

\[
\frac{dx}{dx} = \frac{dc}{dx} \frac{dt}{dc} = \frac{dc}{dx} c(1)
\]

In adjoint calculation, data dependencies are reversed:

\[
\begin{align*}
(x, w, z) &\mapsto y \\
y(1) &\mapsto x(1) & \text{as } x(1) + = \frac{dy}{dx} y(1) \\
y(1) &\mapsto w(1) & \text{as } w(1) + = \frac{dy}{dw} y(1) \\
y(1) &\mapsto z(1) & \text{as } z(1) + = \frac{dy}{dz} y(1)
\end{align*}
\]
Data Dependencies

Recall that for $c \equiv c(x)$ we have

$$x(1) = \frac{dt}{dx} = \frac{dc}{dx} \frac{dt}{dc} = \frac{dc}{dx} c(1)$$

In adjoint calculation, data dependencies are reversed:

$$(x, w, z) \mapsto y \quad \text{means} \quad \begin{cases} y(1) \mapsto x(1) \quad \text{as} \quad x(1) + = \frac{dy}{dx} y(1) \\ y(1) \mapsto w(1) \quad \text{as} \quad w(1) + = \frac{dy}{dw} y(1) \\ y(1) \mapsto z(1) \quad \text{as} \quad z(1) + = \frac{dy}{dz} y(1) \end{cases}$$

A cubic spline is a mapping

$$(x, \lambda[1 : n], c[1 : n]) \mapsto y$$

Given input $x$, determine index $j = j(x)$ such that $\lambda[j] \leq x < \lambda[j + 1]$. De Boor’s formula then gives

$$(x, \lambda[j : j + 5], c[j : 3]) \mapsto y$$
Race Conditions

Consider *independent* inputs $x^p, x^{p+1}, \ldots$ e.g. Monte Carlo paths
Race Conditions

Consider *independent* inputs $x^p, x^{p+1}, \ldots$ e.g. Monte Carlo paths

- Mapping independent inputs $x^p, x^{p+1}, \ldots$ to independent outputs $y^p, y^{p+1}, \ldots$ with spline formula

$$(x^p, \lambda[1:n], c[1:n]) \mapsto y^p$$

in parallel is fine (good for GPU)
Race Conditions

Consider *independent* inputs $x^p, x^{p+1}, \ldots$ e.g. Monte Carlo paths

- Mapping independent inputs $x^p, x^{p+1}, \ldots$ to independent outputs $y^p, y^{p+1}, \ldots$ with spline formula

\[(x^p, \lambda[1:n], c[1:n]) \mapsto y^p\]

in parallel is fine (good for GPU)

- Now consider adjoint
Race Conditions

- Independent inputs $y^{p}_{(1)}, y^{p+1}_{(1)}, \ldots$ update independent $x^{p}_{(1)}, x^{p+1}_{(1)}, \ldots$ and shared arrays $\lambda_{(1)}[1:n]$ and $c_{(1)}[1:n]$. 
Race Conditions

- Independent inputs $y_p(1), y_{p+1}(1), \ldots$ update independent $x_p(1), x_{p+1}(1), \ldots$ and shared arrays $\lambda(1)[1:n]$ and $c(1)[1:n]$

- If sample path $p$ and $q$ have same indexes $j(p) = j(q) = j$ then

$$\lambda(1)[j] = \frac{dy_p}{d\lambda[j]} y_p(1) \quad \text{and} \quad \lambda(1)[j] = \frac{dy_q}{d\lambda[j]} y_q(1)$$
Race Conditions

- Independent inputs $y_{(1)}^p, y_{(1)}^{p+1}, \ldots$ update independent $x_{(1)}^p, x_{(1)}^{p+1}, \ldots$ and shared arrays $\lambda_{(1)}[1:n]$ and $c_{(1)}[1:n]

- If sample path $p$ and $q$ have same indexes $j(p) = j(q) = j$ then

\[
\lambda_{(1)}[j] + = \frac{dy^p}{d\lambda[j]} y_{(1)}^p \quad \text{and} \quad \lambda_{(1)}[j] + = \frac{dy^q}{d\lambda[j]} y_{(1)}^q
\]

- Can’t be done in parallel without resolving the race conditions
Race Conditions

- Independent inputs $y_{(1)}^p, y_{(1)}^{p+1}, \ldots$ update independent $x_{(1)}^p, x_{(1)}^{p+1}, \ldots$ and shared arrays $\lambda_{(1)}[1 : n]$ and $c_{(1)}[1 : n]$.

- If sample path $p$ and $q$ have same indexes $j(p) = j(q) = j$ then

$$\lambda_{(1)}[j] += \frac{dy^p}{d\lambda[j]} y_{(1)}^p \quad \text{and} \quad \lambda_{(1)}[j] += \frac{dy^q}{d\lambda[j]} y_{(1)}^q$$

- Can’t be done in parallel without resolving the race conditions.

Two easy fixes

- Atomic operations (atomicAdd)
- Each thread $p$ has own copy of $\lambda_{(1)}, c_{(1)}$, reduce at end
- Both give good performance, with due care
Handwritten Adjoint Kernel
Handwritten Adjoint

This wasn’t fun
Passive Cubic Spline Code

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

```c++
template<typename FP>
__device__ void splineb(int n, FP *lamda,
                       FP *lambda, 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 e4 = k4 - x;
  FP c2 = c[j+1];
  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

\[
\begin{align*}
\text{FP} & \quad c7 = (e3*c[j+3]+(k6-x)*c3)*invk6mk3; \\
\text{FP} & \quad c4 = (e2*c5+e4*c1)*invk4mk2; \\
\text{FP} & \quad c6 = (e3*c7+e5*c5)*invk5mk3; \\
\text{FP} & \quad tempb = sb/(k4-k3); \\
\text{FP} & \quad tempb0 = -((e3*c6+e4*c4)*tempb/(k4-k3)); \\
\text{FP} & \quad c6b = e3*tempb; \\
\text{FP} & \quad c4b = e4*tempb; \\
\text{FP} & \quad tempb1 = c6b*invk5mk3; \\
\text{FP} & \quad tempb7 = -((e3*c7+e5*c5)*tempb1*invk5mk3); \\
\text{FP} & \quad c7b = e3*tempb1; \\
\text{FP} & \quad tempb3 = c4b*invk4mk2; \\
\text{FP} & \quad c5b = e2*tempb3 + e5*tempb1; \\
\text{FP} & \quad tempb5 = -((e2*c5+e4*c1)*tempb3*invk4mk2); \\
\text{FP} & \quad c1b = e4*tempb3; \\
\text{FP} & \quad tempb2 = c7b*invk6mk3; \\
\text{FP} & \quad e3b = c7*tempb1 + c[j+3]*tempb2 + c6*tempb; \\
\text{FP} & \quad tempb8 = -((e3*c[j+3]+(k6-x)*c3) * tempb2*invk6mk3);
\end{align*}
\]
AD on GPU
Jacques du Toit, Klaus Leppkes and Johannes Lotz

Introduction
AD in a Nutshell
Local volatility FX basket option
Pricing Code
Designing an Adjoint Code
Handwritten Adjoint Kernel
An Easier Way

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;

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

Handwritten adjoint code performs well

- Overall runtime: 550ms
- Forward run: 370ms (GPU Monte Carlo 14.5ms)
- tape interpretation: 180ms (GPU adjoint kernel 85ms)
- dco/c++ tape used 268MB CPU RAM
- Used 420MB GPU RAM (includes random numbers)

Ran on an Intel Xeon E5-2670 with an NVIDIA K20X
Handwritten Adjoint Performance

Handwritten adjoint code performs well

- Overall runtime: 550ms
- Forward run: 370ms (GPU Monte Carlo 14.5ms)
- tape interpretation: 180ms (GPU adjoint kernel 85ms)
- dco/c++ tape used 268MB CPU RAM
- Used 420MB GPU RAM (includes random numbers)

Ran on an Intel Xeon E5-2670 with an NVIDIA K20X

However pain in producing prompted search for an easier way
An Easier Way
An Easier Way

- Adjoint code typically ugly, tedious to write, very difficult to read, and almost un-maintainable.
- Anyone seeing this is discouraged from making changes to the primal code.
An Easier Way

- Adjoint code typically ugly, tedious to write, very difficult to read, and almost un-maintainable
- Anyone seeing this is discouraged from making changes to the primal code

However for “simple” code like the spline, making adjoint code is completely algorithmic
An Easier Way

- Adjoint code typically ugly, tedious to write, very difficult to read, and almost un-maintainable
- Anyone seeing this is discouraged from making changes to the primal code

However for “simple” code like the spline, making adjoint code is completely algorithmic

So is it not possible to automate this?
An Easier Way

- Adjoint code typically ugly, tedious to write, very difficult to read, and almost un-maintainable
- Anyone seeing this is discouraged from making changes to the primal code

However for “simple” code like the spline, making adjoint code is completely algorithmic

So is it not possible to automate this?

Clearly (AD compilers) but can we do better?
C++11

C++ is not one, but three languages

- Procedural language of C
- Object oriented language
- Meta-programming language (templates)
C++11

C++ is not one, but three languages

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

Interesting one here is meta-programming language

- A language to write programs that output programs
- Meta-program is run at compile time, producing code which is then compiled
- Is Turing complete
C++ is not one, but three languages

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

Interesting one here is meta-programming language

- A language to write programs that output programs
- Meta-program is run at compile time, producing 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
Template 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?_
C++11

Template 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.
**C++11**

Template 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 essentially no

With C++11 that changes
The Delights of “auto”

C++11 introduces the keyword `auto`

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

```cpp
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 fundamental blocks in template meta-programming language
- We can now play with types while keeping the source code flexible

Implemented this in a new **dco/c++** type called **dco::tfa**
Normal Cubic Spline Code

template<typename FP>
__device__ void spline(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 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;
    FP 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);
}
Spline with New \texttt{dco/c++} Type

\begin{verbatim}
template<typename FP>
__device__ void spline(dco::tfa<FP> *lamda, dco::tfa<FP> *c, dco::tfa<FP> &x, dco::tfa<FP> &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 c2 = c[j+1];
    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*c[j])/(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);
}
\end{verbatim}

Assignment to output triggers meta-program to reverse DAG and compute adjoint
New dco/c++ Type

- Have array class `dco::tfarray` to ease handling of shared input arrays
- Optimises storage, memory access and ensures thread safety

Final version of spline function looks like this
Spline with New dco/c++ Type

```cpp
template<typename FP, typename ARRAY>
__device__ void spline(ARRAY &lamda, ARRAY &c, FP &x, FP &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 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*c[j])/(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);
}
```

One code, callable with active or passive types
Conditionals

We can also handle if-else statements

```cpp
...  
const auto  c4 = c2*k1 - c3*k2;  
const auto  c5 = c1*k5 - c2*k6;  
// Declare output of the if-else block  
dco::tfa::Ifelse_output<FP> out1, out2;  

  dco::tfa::If(c4 > c5, [&] {  
    const auto  x = exp(c5)*sin(c3);  
    out1 = x*c2 - c4*cos(c5);  
    const auto  y = k1*k2 + out1*x;  
    out2 = y*out1 + out1*out1;  
  });, dco::tfa::Else, [&] {  
    out1 = c4*k3 - e3*c5;  
    const auto  z = k1 - out1;  
    out2 = out1*out1 - z*z + cos(c4);  
  }););  
const auto  c6 = out1*k4 - out2*k7;  
...  
```
New dco/c++ Types

New set of dco/c++ types can handle all basic elements of procedural codes
- Code blocks
- Loops
- Conditionals
- Function calls

The resulting code is typically as efficient as handwritten adjoints
Advantages

Users can build handwritten adjoints on CPU or GPU with much greater ease

- No longer need to write low-level adjoints
- Single source code for forward and adjoint programs
- Easy to read and modify code - adjoints always in sync
- Adjoint code typically as efficient as handwritten
- User focuses on overall dataflow reversal and placing of checkpoints

Frees user to think and work at much higher level when making handwritten adjoint codes
Performance of New Approach on Local Vol Basket

Recall runtimes for handwritten adjoint

- Overall runtime: 550ms
- Forward Monte Carlo kernel: 14.5ms
- Handwritten Adjoint Monte Carlo kernel: 85ms

Runtime with new dco/c++ types

- Overall runtime: 555ms
- Forward Monte Carlo kernel: 14.5ms
- Handwritten Adjoint Monte Carlo kernel: 90ms
Thank You

Questions