Algorithmic Differentiation of a GPU Accelerated Application

Jacques du Toit

Numerical Algorithms Group

Disclaimer

- This is not a “speedup” talk
- There won’t be any speed or hardware comparisons here
- This is about what is possible and how to do it with the minimum of effort

This talk aimed at people who

- Don’t know much about AD
- Don’t know much about adjoints
- Don’t know much about GPU computing

Apologies if you’re not in one of these groups ...

What is Algorithmic Differentiation?

It's a way to compute

$$\frac{\partial}{\partial x_1}F(x_1, x_2, x_3, \ldots) \qquad \frac{\partial}{\partial x_2}F(x_1, x_2, x_3, \ldots)$$
$$\frac{\partial}{\partial x_3}F(x_1, x_2, x_3, \ldots) \qquad \ldots$$

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

```
if(x1<x2) then
  F = x1*x1 + x2*x2 + x3*x3 + ...
else
  F = x1 + x2 + x3 + ...
endif
```

Why are Derivatives Useful?

If you have a computer program which computes something (e.g. price of a contingent claim), AD can give you the derivatives of the output with respect to the inputs

- The derivatives are exact up to machine precision (no approximations)

Why is this interesting in Finance?

- Risk management
- Obtaining exact derivatives for mathematical algorithms such as optimisation (gradient and Hessian based methods)

There are other uses as well but these are the most common

Local Volatility FX Basket Option

A while ago (with Isabel Ehrlich, then at Imperial College) we made a GPU accelerated pricer for a basket option

- Option written on 10 FX rates driven by a 10 factor local volatility model, priced by Monte Carlo
- The implied vol surface for each FX rate has 7 different maturities with 5 quotes at each maturity
- All together the model has over 400 input parameters

Plan: compute gradient of the price with respect to model inputs including **market implied volatility quotes**

- Want to differentiate through whatever procedure is used to turn the implied vol quotes into a local vol surface
- Due to the large gradient, want to use *adjoint algorithmic differentiation*
- We also want to use the GPU for the heavy lifting

Local Volatility FX Basket Option

If $S^{(i)}$ denotes i^{th} underlying FX rate then

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

where $(\mathbf{W}_t)_{t\geq 0}$ is a correlated N-dimensional Brownian motion with $\langle W^{(i)}, W^{(j)}\rangle_t = \rho^{(i,j)}t$.

The function $\sigma^{(i)}$ is unknown and is calibrated from market implied volatility quotes according to the Dupire formula

$$\sigma^2(K,T) = \frac{\theta^2 + 2T\theta\theta_T + 2\left(r_T^d - r_T^f\right)KT\theta\theta_K}{\left(1 + Kd_+\sqrt{T}\theta_K\right)^2 + K^2T\theta\left(\theta_{KK} - d_+\sqrt{T}\theta_K^2\right)}.$$

where θ the market observed implied volatility surface. The basket call option price is then

$$C = e^{-r_d T}\mathbb{E}\left(\sum_{i=1}^{N} w^{(i)} S_T^{(i)} - K\right)^+$$

Crash Course in Algorithmic Differentiation

Algorithmic Differentiation in a Nutshell

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

- A computer program implementing a model is just many of these fundamental operations strung together
- It's elementary to compute the derivatives of these fundamental operations
- So we can use the chain rule, and these fundamental derivatives, to get the derivative of the output of a computer program with respect to the inputs
- Classes, templates and operator overloading give a way to do all this efficiently and non-intrusively

Adjoints in a Nutshell

AD comes in two modes: forward (or tangent-linear) and reverse (or adjoint) mode

- Consider $f : \mathbb{R}^n \to \mathbb{R}$, take a vector $\mathbf{x^{(1)}} \in \mathbb{R}^n$ and define the function $F^{(1)} : \mathbb{R}^{2n} \to \mathbb{R}$ by

$$y^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x^{(1)}}) = \nabla f(\mathbf{x}) \cdot \mathbf{x^{(1)}} = \left(\frac{\partial f}{\partial \mathbf{x}}\right) \cdot \mathbf{x^{(1)}}$$

  where the dot is regular dot product.
- $F^{(1)}$ is the *tangent-liner model* of f and is the simplest form of AD.
- Let $\mathbf{x^{(1)}}$ range over Cartesian basis vectors and call $F^{(1)}$ repeatedly to get each partial derivative of f
- To get full gradient ∇f, must evaluate the forward model n times
- Runtime to get whole gradient will be roughly n times the cost of computing f

Adjoints in a Nutshell

Take any $y_{(1)}$ in $\mathbb{R}$ and consider $F_{(1)} : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^n$ given by

$$\mathbf{x_{(1)}} = F_{(1)}(\mathbf{x}, y_{(1)}) = y_{(1)} \nabla f(\mathbf{x}) = y_{(1)} \frac{\partial f}{\partial \mathbf{x}}$$

- $F_{(1)}$ is called the *adjoint model* of f
- Setting $y_{(1)} = 1$ and calling adjoint model $F_{(1)}$ **once** gives the full vector of partial derivatives of f
- Furthermore, can be proved that in general computing $F_{(1)}$ requires no more than five times as many flops as computing f

Hence adjoints are extremely powerful, allowing one to obtain large gradients at potentially very low cost.

Adjoints in a Nutshell

So how do we construct a function which implements the adjoint model?

- Mathematically, adjoints are defined as partial derivatives of an auxiliary scalar variable t so that

$$y_{(1)} = \frac{\partial t}{\partial y} \quad \text{and} \quad \mathbf{x}_{(1)} = \frac{\partial t}{\partial \mathbf{x}} \quad \text{(note: latter is a vector)}$$

- Consider a computer program computing y from x through intermediate steps

$$x \mapsto \alpha \mapsto \beta \mapsto \gamma \mapsto y$$

How do we compute the adoint model of this calculation?

Adjoints in a Nutshell

$$x \mapsto \alpha \mapsto \beta \mapsto \gamma \mapsto y$$

Using the definition of adjoint we can write

$$\begin{aligned} x_{(1)} = \frac{\partial t}{\partial x} &= \frac{\partial \alpha}{\partial x}\frac{\partial t}{\partial \alpha} \\ &= \frac{\partial \alpha}{\partial x}\frac{\partial \beta}{\partial \alpha}\frac{\partial t}{\partial \beta} \\ &= \frac{\partial \alpha}{\partial x}\frac{\partial \beta}{\partial \alpha}\frac{\partial \gamma}{\partial \beta}\frac{\partial t}{\partial \gamma} \\ &= \frac{\partial \alpha}{\partial x}\frac{\partial \beta}{\partial \alpha}\frac{\partial \gamma}{\partial \beta}\frac{\partial y}{\partial \gamma}\frac{\partial t}{\partial y} = \frac{\partial y}{\partial x} y_{(1)} \end{aligned}$$

which is the adjoint model we require.

Adjoints in a Nutshell

Note that $y_{(1)}$ is an *input* to the adjoint model and that

$$x_{(1)} = \left(\left(\left(y_{(1)} \cdot \frac{\partial y}{\partial \gamma}\right) \cdot \frac{\partial \gamma}{\partial \beta}\right) \cdot \frac{\partial \beta}{\partial \alpha}\right) \cdot \frac{\partial \alpha}{\partial x}$$

- Computing $\partial y / \partial \gamma$ will probably require knowing γ (and/or β and α as well)

Effectively means have to run the computer program *backwards*

- To run the program *backwards* we first have to run it *forwards* and store all intermediate values needed to calculate the partial derivatives
- In general, adjoint codes can require a huge amount of memory to keep all the required intermediate calculations.

Adjoints in Practice

So to do an adjoint calculation we need to

- Run the code forwards storing intermediate calculations
- Then run it backwards and compute the gradient

This is a complicated and error-prone task

- Difficult to do by hand: for large codes (few 1000 lines), simply infeasible
- Very difficult to automate this process efficiently
- In either case, can be tricky to do without running out of memory

This is not something you want to do yourself!

Prof Uwe Naumann and his group at Aachen University produce a tool dco which takes care of all of this for you.

dco = Derivative Computation through Overloading

dco

Broadly speaking, dco works as follows:

- Replace all active datatypes in your code with dco datatypes
- Register the input variables with dco
- Run the calculation forwards: dco tracks all calculations depending on input variables and stores intermediate values in a **tape**
- When forward run complete, set adjoint of output (price) to 1 and call `dco::a1s::interpret_adjoint`
- This runs the tape backwards and computes the adjoint (gradient) of output (price) with respect to all inputs

dco

dco is one of the most efficient overloading AD tools

- Has been used on huge codes (e.g. Ocean Modelling and Shape Optimisation)
- Supports checkpointing and user-defined “adjoint functions” (e.g. a hand-written adjoint)
- Supports the NAG library: derivative calculations can be carried through calls to NAG Library routines

Unfortunatley, dco doesn’t (yet) support accelerators

- In fact I’m not aware of any AD tools that support accelerators

Basket Option Code

Basket Option Code

The basket option code is broken into 3 stages

- Stage 1: Setup (on CPU) — process market input implied vol quotes into local vol surfaces
- Stage 2: Monte Carlo (on GPU) — copy local vol surfaces to GPU and create all the sample paths
- Stage 3: Payoff (on CPU) — get final values of sample paths and compute payoff

Doing final step on CPU was a deliberate decision to mimic banks' codes and has nothing to do with performance

Basket Option Code

Stage 1 is the longest (i.t.o. lines of code)

- Series of interpolation and extrapolation steps
- Cubic splines and interpolating Hermite polynomials
- Several calls to NAG Library routines
- Stages 1 and 3 are the cheapest i.t.o. flops in the forward run

Stage 2 GPU Monte Carlo code is pretty simple

- It is the most expensive i.t.o. flops in the forward run
- However it's executed quickly on a GPU because it's highly parallel

Now dco can handle the CPU code no problem: but what about GPU code?

External Function Interface

Recall dco supports user-defined adjoint functions

- These called **external functions**
- Effectively allow you to put “gaps” in dco’s tape
- You then provide the code that fills those gaps
- Code can be arbitrary, just has to implement an adjoint model
  - Take input adjoints from dco
  - Provide output adjoints to dco
- How it does that is up to the user
- So we can use external functions to handle GPU

Adjoint of Monte Carlo kernel will be the most expensive i.t.o. flops in the entire calculation.

- Really want this on GPU if at all possible (means we’ll need hand-written adjoint of Monte Carlo kernel)

Monte Carlo Kernel: Forward Run

So what do we need to store from the Monte Carlo kernel? The Euler-Maruyama discretisation is

$$S_{i+1} = S_i + S_i * \left((r_d - r_f)\Delta t + \sigma(S_i, i * \Delta t)\sqrt{\Delta t} Z_i \right)$$

- At Monte Carlo time step i we need to know S_i to compute S_{i+1}
- S_i was computed at previous time step
- Nothing else is carried over from previous time step
- To run this calculation backwards (i.e. start with S_{i+1} and compute S_i) we'll need to know S_i to calculate $\sigma(S_i, i\Delta t)$ since σ is not invertible

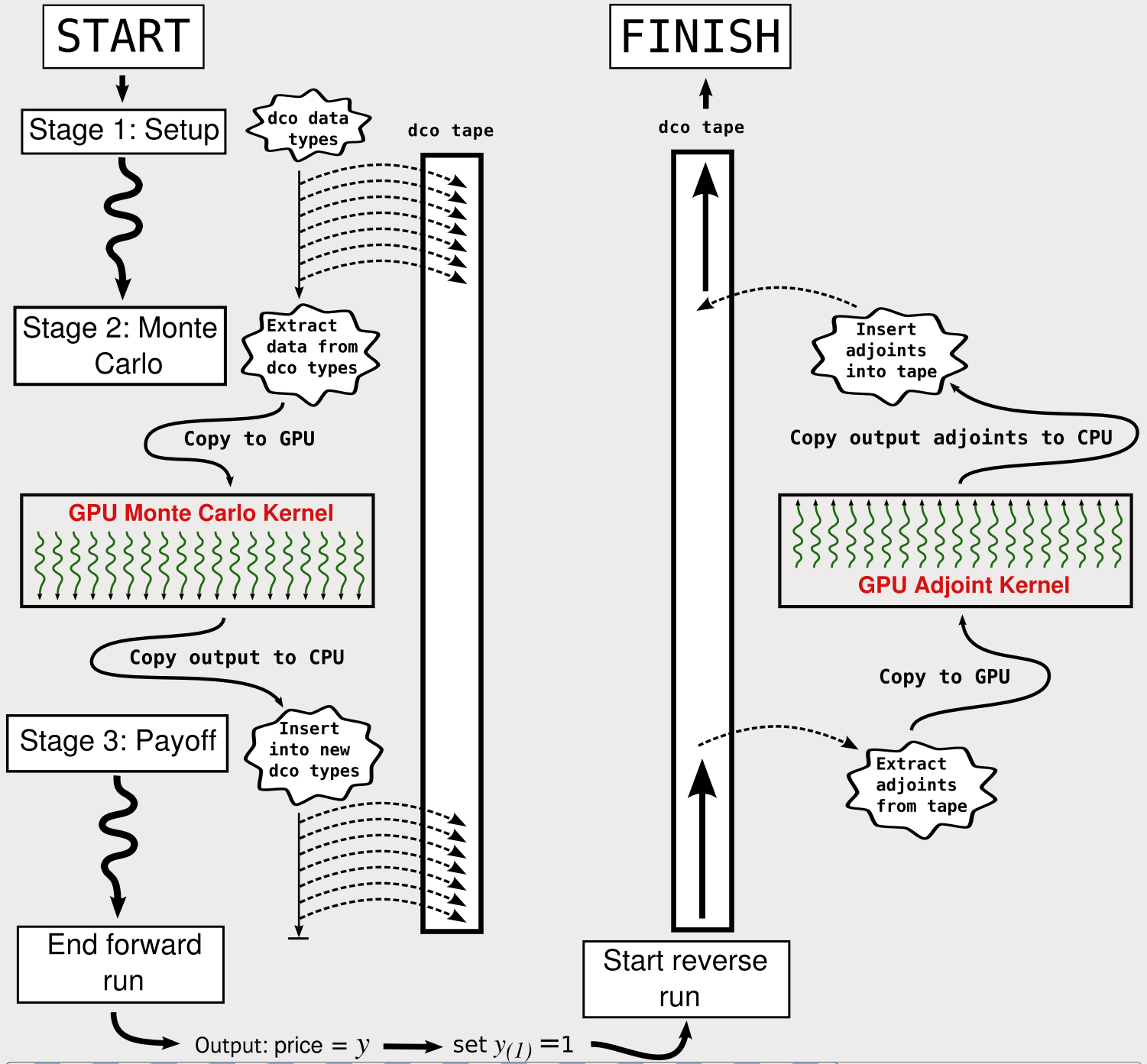
Adjoint of Monte Carlo Kernel

So what does all this mean?

- In the forward run, it is sufficient to store S_i for all sample paths and all Monte Carlo time steps (i.e. store each step from each path)
- From these, all other values which may be needed for adjoint calculation can be recomputed
- To avoid having to recompute the local vol we store $\sigma(S_i, i * \Delta t)$ as well

Adjoint of the Monte Carlo kernel (currently) has to be written by hand

- This is actually not too difficult
- Local vol surfaces stored as splines, so most onerous part is writing an adjoint of a spline evaluation function
- This is about 150 lines of code, so is not that bad
- The adjoint kernel is massively parallel and can be performed on the GPU as well


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

Test Problem and Results

As a test problem we took 10 FX rates

- For each rate we had market quotes at 5 different strikes at each of 7 different maturities
- Estimated the correlation structure from historical data, then obtained a nearest correlation matrix
- Used 360 Monte Carlo time steps and 10,000 Monte Carlo sample paths
- Full gradient consisted of 438 entries

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

- Overall runtime: 522ms
- Forward run was 367ms (Monte Carlo was 14.5ms)
- Computation of adjoints was 155ms (of which GPU adjoint kernel was 85ms)
- dco used 268MB CPU RAM
- In total 420MB GPU RAM was used (includes random numbers)

A Rather Simple Race Condition

When computing adjoints, dependencies between data are reversed

- If r produced s, then $s_{(1)}$ produces $r_{(1)}$
- If r produced s_1, s_2, s_3, then $s_{1(1)}, s_{2(1)}, s_{3(1)}$ all combine to produce $r_{(1)}$
- This combination is typically additive

Recall the Euler-Maruyama equation

$$S_{i+1} = S_i + S_i * \left((r_d - r_f)\Delta t + \sigma(S_i, i * \Delta t)\sqrt{\Delta t} Z_i \right)$$

- r_d (and r_f) feed into every sample path at every time step
- Hence the adjoints of all sample paths will feed into $r_{d(1)}$ at each time step
- We parallelise the adjoint kernel across sample paths, so different threads will need to update $r_{d(1)}$ at the same time: a race condition

A Rather Simple Race Condition

This race is easy to handle

- Each thread has its own private copy of $r_{d(1)}$ which it updates as it works backwards from maturity to time 0
- When all threads reach time 0, these private copies are combined in a parallel reduction which is thread safe

The same can be done for the adjoints of r_f, Δt and the correlation coefficients

A Really Nasty Race Condition

Recall the local volatility surfaces are stored as splines

- Separate spline for each Monte Carlo time step
- Each spline has several (20+) knots and coefficients
- To compute $\sigma(S_i, i * \Delta t)$ six knots and six coefficients are selected based on value of S_i
- In adjoint calculation, adjoint of S_i will update the adjoints of the six knots and six coefficients
- However another thread processing another sample path could want to update (some of) those data as well: a race condition

So what makes this nasty?

- Scale: 40,000 threads with 10 assets and 360 1D splines per asset
- It's over 21GB if each thread has own copy
- So you have to do something different

This nasty race is a peculiar feature of local volatility models

A Really Nasty Race Condition

So what can we do about this?

- Give each thread it's own copy of spline data in shared memory – leads to low occupancy and poor performance
- Give each thread block a copy in shared memory – need a lot of synchronisation, hence poor performance
- Give each thread block a copy in shared memory and use atomics – works, but is slow (at least 4x slower than current code)

Point is, not all 40,000 threads are **active** at the same time

- So if active blocks could grab some memory, use it and then release it, the memory problems go away
- This is the approach we took
- Each thread block allocates some memory and gives each thread a private copy of spline data
- When block exits, it releases the memory

Summary

- By combining dco with hand-written adjoints, the full gradient of a GPU accelerated application can be computed very efficiently
- In many financial models, some benign race conditions arise when computing the adjoint
- In local volatility-type models (such as SLV) a rather nasty race condition arises
- These conditions can be dealt with through judicious use of memory
- Note that the race conditions are independent of the platform used (CPU or GPU): on a GPU the condition is much more pronounced

Summary

But what we really want is for dco to support CUDA

This is work in progress, watch this space!

Summary

Thank you