Adjoint Algorithmic Differentiation of a GPU Accelerated Application

Jacques du Toit[*],
Johannes Lotz and Uwe Naumann[†]

Abstract

We consider a GPU accelerated program using Monte Carlo simulation to price a basket call option on 10 FX rates driven by a 10 factor local volatility model. We develop an adjoint version of this program using algorithmic differentiation. The code uses mixed precision. For our test problem of 10,000 sample paths with 360 Euler time steps, we obtain a runtime of 522ms to compute the gradient of the price with respect to the 438 input parameters, the vast majority of which are the market observed implied volatilities (the equivalent single threaded tangent-linear code on a CPU takes 2hrs).

1 Introduction

In a financial setting risk is more or less synonymous with sensitivity calculations. The foundational results of mathematical finance show how to hedge any contingent claim by using a dynamic trading strategy based on mathematical derivatives of the claim with respect to the risk factors. These mathematical derivatives (sensitivities) therefore show us not only where our exposures are most sensitive with respect to movements in the risk factors, but also tell us how (in a perfect world) we can eliminate those exposures completely.

The most common way of computing sensitivities is by finite differences. Through a simple "bump and revalue" approach one can get estimates of any mathematical derivative. There are two problems, however: firstly, those estimates (especially for higher order derivatives) can be arbitrarily bad if one does not have a good feel for the underlying surface; and secondly, bump and revalue is a costly computational exercise.

Algorithmic differentiation has become an attractive alternative to finite differences for many applications. For a thorough introduction to algorithmic differentiation (hereafter simply AD) the reader is referred to [1], but the idea is very simple. At the most basic level, computers can only add, subtract, multiply and divide two floating point numbers. It is elementary to compute the derivatives of these fundamental operations. A computer program implementing a mathematical formula or model consists of many of these fundamental operations strung together. One can therefore use the chain rule, together with the derivatives of these operations, to compute the mathematical derivative of the computer program with respect to some input variables. Furthermore, considering programming language features such as templates and operator overloading, one begins to see how a software tool could be written to perform AD automatically in an efficient and non-intrusive manner.

[*]Numerical Algorithms Group
[†]LuFG Informatik 12, RWTH Aachen University, Germany

The main benefit of AD is exact mathematical derivatives of a computer program. When this is coupled with adjoint methods (see Section 4), a substantial increase in computational efficiency can often be achieved. We refer to [1] for a full discussion of these matters.

A word of caution: AD computes exact mathematical derivatives of *the given computer program*. This presumes two things:

- the given computer program is continuously differentiable,
- the derivative of the computer program bears some resemblance to the derivative of the mathematical formula or algorithm that it implements.

These two conditions are often not satisfied. Indeed for certain variables such as interest rates there is a smallest unit of measurement (the basis point), which leads some practitioners to prefer finite differences when computing sensitivities with respect to these variables. It is important to consider a computer program carefully when deciding whether AD is an appropriate technique.

The main purpose of this article is to demonstrate that it is possible to apply AD, and in particular adjoint AD, to a non-trivial GPU (CUDA) accelerated program. The program consists of three stages: a complex initial setup stage, a compute intensive massively parallel stage, and a post-processing stage. The compute intensive stage is executed as a GPU kernel while the rest of the code is kept on the CPU. By combining an AD tool (dco) with a hand-written adjoint of the GPU kernel, the entire program can be differentiated with respect to the over 400 input variables.

2 Local volatility FX basket option

The problem considered is a foreign exchange (FX) basket call option driven by a multi-factor local volatility model. This is the same problem presented in [2] and we refer the reader there for further details. We will outline the problem briefly here.

A basket call option consists of a weighted sum of N correlated assets and has a price given by

$$C = e^{-r_d T}\mathbb{E}\left(B_T - K\right)^+ \tag{1}$$

where r_d is the domestic risk free interest rate, $K > 0$ is the strike price, and B_T is given by

$$B_T = \sum_{i=1}^{N} w^{(i)} S_T^{(i)}\,. \tag{2}$$

Here $S^{(i)}$ denotes the value of i^{th} underlying asset (FX rate) for $i = 1, \ldots, N$ and the $w^{(i)}$s are a set of weights summing to one. Each underlying asset is driven by the stochastic differential equation (SDE)

$$\frac{dS_t^{(i)}}{S_t^{(i)}} = \left(r_d - r_f^{(i)}\right)dt + \sigma^{(i)}\left(S_t^{(i)}, t\right)dW_t^{(i)} \tag{3}$$

where $r_f^{(i)}$ is the foreign risk free interest rate for the i^{th} currency pair, $(\mathbf{W}_t)_{t\geq 0}$ is a correlated N-dimensional Brownian motion with $\left\langle W^{(i)}, W^{(j)}\right\rangle_t = \rho^{(i,j)}t$ and $\rho^{(i,j)}$ denotes the correlation coefficient for $i, j = 1, \ldots, N$. The function $\sigma^{(i)}$ in (3) is unknown and is calibrated from market implied volatility data according to the Dupire formula

$$\sigma^2(K,T) = \frac{\theta^2 + 2T\theta\theta_T + 2\left(r_T^d - r_T^f\right)KT\theta\theta_K}{\left(1 + Kd_+\sqrt{T}\theta_K\right)^2 + K^2T\theta\left(\theta_{KK} - d_+\sqrt{T}\theta_K^2\right)}. \tag{4}$$

where θ denotes the market observed implied volatility surface and

$$d_+ = \frac{\ln(S_0^{(i)}/K) + (r - q + \frac{1}{2}\theta^2)T}{\theta\sqrt{T}}. \tag{5}$$

The basket call option price (1) is evaluated using Monte Carlo simulation.

The inputs to this model are therefore the domestic and foreign risk free rates, the spot values of the FX rates, the correlation matrix, the maturity and strike of the option, the weights, and the market observed implied volatility surface for each of the underlying FX rates. The task is to compute the sensitivity of the price with respect to each of these variables.

3 The GPU accelerated computer program (pre-AD)

Let us examine the computer program evaluating (1) before any AD is introduced. As mentioned previously, the program is divided into three stages.

3.1 Stage 1: setup

The Dupire formula (4) implies there are infinitely many implied volatility quotes. However the market only trades a handful of call options at different strikes and maturities. One therefore has to create a sufficiently smooth function θ from this fairly small set of discrete quotes which can be used in (4). Doing this in an arbitrage-free manner is not easy and has been a topic of much research. A common approach taken by market practitioners is to ignore the finer points of arbitrage free interpolation and simply to use cubic splines. This is the approach that we have adopted.

The first stage of the program takes the input implied volatility surface and produces a local volatility surface represented by a collection of one dimensional cubic splines. Fuller details of this procedure are given in [2] and here we just summarise the main steps to give a feel for the complexity.

The FX market typically provides option quotes for five "deltas" per tenor. We fit a cubic spline in the strike direction through the 5 quotes at each tenor. For each tenor we then fit 5 piecewise monotonic Hermite polynomials from time 0 to time T, with each polynomial passing through one of the quotes. Since each tenor's quotes are at different strikes, this requires interpolating values with the previously fitted cubic splines. For all the Monte Carlo time steps that lie between two tenors, we use the Hermite polynomials to interpolate implied volatility values at each of the 5 strikes. Then for each of the of the Monte Carlo time steps cubic splines are fitted in the strike direction through the 5 implied volatility values just computed. These splines can be used to compute θ, θ_K and θ_{KK} in (4) above. Essentially the same procedure can be used to compute θ_T. Finally, these derivatives are substituted into (4) to produce local volatility points on a regular grid, and sets of cubic splines are fitted to these points. The slopes at the end points of these splines are projected to form linear extrapolation functions to cover the left and right (in the strike direction) tails of the local volatility surface.

3.2 Stage 2: GPU accelerated Monte Carlo

We use an Euler-Maruyama scheme on the log-process to solve (3). This is by far the most compute intensive part of the entire program and is executed on a GPU. To perform the Euler time stepping we use n_T steps each of length Δt so that $n_T \Delta t = T$, the maturity of the basket option. With $S_\tau^{(i)} \equiv S_{\tau\Delta t}^{(i)}$ for $\tau = 1, \ldots, n_T$ the discretised version of (3) then

becomes

$$\log\left(S_{\tau+1}^{(i)}\right)-\log\left(S_{\tau}^{(i)}\right)=\left(r_d-r_f^{(i)}-\tfrac{1}{2}\sigma^{(i)}(S_{\tau}^{(i)},\tau\Delta t)^2\right)\Delta t+\sigma^{(i)}(S_{\tau}^{(i)},\tau\Delta t)\sqrt{\Delta t}\,\mathbf{d_i}\mathbf{Z}_{\tau} \quad (6)$$

where $\mathbf{Z}_\tau$ is an $N \times 1$ column vector of standard Normal random numbers and $\mathbf{d_i}$ is the i^{th} row of the Cholesky factorisation of the $N \times N$ correlation matrix, in other words $\mathbf{d_{i_1}}\mathbf{d_{i_2}}^T = \rho^{(i_1,i_2)}$ for all $i_1, i_2 = 1, \ldots, N$. The output of this stage is the final value $\log\left(S_{nT}^{(i)}\right) \equiv \log\left(S_{nT}^{(i)}(j)\right)$[a] for each of the j Monte Carlo sample paths and each of the i underlying assets $i = 1, \ldots, N$.

3.3 Stage 3: payoff

The final stage consists of copying the terminal values of the Monte Carlo sample paths back to the CPU and computing the price (1). While the payoff could certainly be computed on the GPU (an indeed this was done in [2]) we deliberately carried out these final steps on the CPU in order to mimic financial applications where there is a non-trivial post-processing stage which has to be carried out on the CPU.

3.4 Implementation

The program described above is written in C++ and the GPU kernel is written in CUDA. Since the vast majority of the runtime is spent in the Monte Carlo kernel, the CPU C++ code was written in a natural manner with no special attention to optimisation. By contrast, quite some care was taken to produce efficient CUDA code. Probably the most important design decision was to load a single cubic spline for a Monte Carlo time step τ into shared memory and advance all sample paths from time τ to time $\tau+1$ before loading the cubic spline for the next time step. The spline is kept in "cache" for as long as it is needed, and as little cache as possible is consumed. This was done so that the code would work well with small as well as large cubic splines, thereby covering as many asset classes as possible.

4 Adjoint algorithmic differentiation

Adjoint methods, whether analytic or obtained through algorithmic differentiation, are sophisticated numerical techniques. We can but give the briefest of introductions here – the reader is referred to [1] for a fuller exposition.

AD is a technique for differentiating computer programs. There are two main modes (or models) of AD. Consider the case of a computer program implementing a function $f : \mathbb{R}^n \to \mathbb{R}$ namely $y = f(\mathbf{x})$. Take a vector $\mathbf{x^{(1)}} \in \mathbb{R}^n$ and define the function $F^{(1)} : \mathbb{R}^{2n} \to \mathbb{R}$ by

$$y^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x^{(1)}}) = \nabla f(\mathbf{x}) \cdot \mathbf{x^{(1)}} = \left(\frac{\partial f}{\partial \mathbf{x}}\right) \cdot \mathbf{x^{(1)}} \quad (7)$$

where the dot denotes the regular vector dot product. The function $F^{(1)}$ is called the *tangent-linear model* of f and is the simplest form of AD. Letting $\mathbf{x^{(1)}}$ range over the Cartesian basis vectors in $\mathbb{R}^n$ and calling $F^{(1)}$ repeatedly one obtains each of the partial derivatives $y^{(1)}$ of f with respect to each of the input variables x_i for $i = 1, \ldots, n$. Hence in order to get the full gradient ∇f one must evaluate the tangent-linear model n times, which means the runtime of the risk calculation will be roughly n times the cost of computing f (there are precise results in this direction).

[a] We will write $S_\tau^{(i)}(j)$ when we wish to highlight the role of sample paths, otherwise we suppress the (j) to simplify notation

Now consider the function $F_{(1)} : \mathbb{R}^{n+1} \to \mathbb{R}^n$ defined by

$$\mathbf{x_{(1)}} = F_{(1)}(\mathbf{x}, y_{(1)}) = y_{(1)} \nabla f(\mathbf{x}) = y_{(1)} \frac{\partial f}{\partial \mathbf{x}} \tag{8}$$

for any value $y_{(1)}$ in $\mathbb{R}$. The function $F_{(1)}$ is called the *adjoint model* of f. Setting $y_{(1)} = 1$ and calling the adjoint model $F_{(1)}$ once gives the full vector of partial derivatives of f. Furthermore, it can be proved that in general computing $F_{(1)}$ requires no more than five times as many flops[b] as computing f. Hence adjoints are extremely powerful, allowing one to obtain large gradients at potentially very low cost.

Mathematically, adjoints are defined as partial derivatives of an auxiliary scalar variable t so that in (8) above we have

$$\begin{aligned} y_{(1)} &= \frac{\partial t}{\partial y} \\ \mathbf{x_{(1)}} &= \frac{\partial t}{\partial \mathbf{x}} \end{aligned} \tag{9}$$

This has a subtle and significant implication for the adjoint model. Consider a computer program evaluating a function $f : \mathbb{R} \to \mathbb{R}$ which takes an input x and by a series of simple operations (e.g. add, subtract, multiply, divide) propagates x through a sequence of temporary variables to an output y

$$x \mapsto \alpha \mapsto \beta \mapsto \gamma \mapsto y \tag{10}$$

where α, β, γ are temporary variables. Using the definition of adjoint (9), we can write

$$\begin{aligned} x_{(1)} = \frac{\partial t}{\partial x} &= \frac{\partial \alpha}{\partial x} \frac{\partial t}{\partial \alpha} \\ &= \frac{\partial \alpha}{\partial x} \frac{\partial \beta}{\partial \alpha} \frac{\partial t}{\partial \beta} \\ &= \frac{\partial \alpha}{\partial x} \frac{\partial \beta}{\partial \alpha} \frac{\partial \gamma}{\partial \beta} \frac{\partial t}{\partial \gamma} \\ &= \frac{\partial \alpha}{\partial x} \frac{\partial \beta}{\partial \alpha} \frac{\partial \gamma}{\partial \beta} \frac{\partial y}{\partial \gamma} \frac{\partial t}{\partial y} = \frac{\partial y}{\partial x} y_{(1)} \end{aligned} \tag{11}$$

which is nothing other than the adjoint model $F_{(1)}$ of f. Note however that $y_{(1)}$ is an *input* to the adjoint model. Hence to evaluate the adjoint model one starts with $y_{(1)}$ and then propagates it upwards through (11) in order to reach $x_{(1)}$, i.e. one has to compute

$$\left(\left(\left(y_{(1)} \cdot \frac{\partial y}{\partial \gamma} \right) \cdot \frac{\partial \gamma}{\partial \beta} \right) \cdot \frac{\partial \beta}{\partial \alpha} \right) \cdot \frac{\partial \alpha}{\partial x} \tag{12}$$

This effectively means that the computer program implementing f has to be run *backwards*. Since computing $\partial y / \partial \gamma$ might well require knowing γ (in general one may need β and α as well), it is clear that to run the program backwards one first has to run the program forwards and store all the intermediate values needed to calculate the partial derivatives. In general, adjoint codes can require a huge amount of storage to keep all the required intermediate calculations.

5 dco

To implement adjoint AD there are two options: one can write the adjoint code by hand, or one can use a software tool which computes adjoints "automatically." Writing adjoints

[b] Floating point operations

by hand is a tedious and error prone task, and for large codes is simply too expensive. In this case, tools are the only option.

Developers of AD tools have come up with two different approaches to generate adjoint code. Both approaches require the source code to be available and both approaches need to collect data during a first step (forward run), reusing it during a second step (reverse run) as shown above (see [1] for a much more rigorous discussion).

The first approach is to use source code transformation tools which make a static program analysis and generate a new program which computes adjoints. The resulting code is usually quite efficient. The major drawback of this approach is language coverage: while source transformation tools may cover a large subset of simpler programming languages such as C, they typically have very limited, or no coverage, of more advanced languages such as C++. For C++ in particular a non-negligible, non-trivial code rewrite is typically required before the tool can understand the program.

The second approach is to use operator overloading techniques to compute adjoints. This approach of necessity uses the language that the underlying program is written in and so has full language coverage. The basic idea is that one goes through the underlying program and replaces all floating point data types with a new specialized data type. For a simple C++ code this is roughly all that is required, but for real world applications further preprocessing may be required. The downside to overloading techniques is that the resulting code may suffer from varying drops in runtime performance. All AD tools built on operator overloading create a data structure during the forward run which keeps track of the dependencies from the inputs to the outputs. This data structure is typically called a *tape*. During the reverse run the tape is played back to compute the adjoint. The performance of the tool is usually closely tied to how efficient this data structure is.

Although the previous two approaches may seem mutually exclusive, for a language such as C++ there is in fact quite some overlap. The AD tool dco tries to exploit this. At its heart it is an operator overloading approach, but it uses efficiency-raising C++ features such as expression templates (see e.g. [4]) to exploit as much compile time optimization as possible. Additionally, dco provides an *external function interface* which is a user friendly way to integrate with itself any code that implements an adjoint model. This code would typically come either from a source transformation tool (the first approach outlined above) or would be hand written. This is the key feature we exploit for the GPU accelerated program considered here.

5.1 Performance and features

The typical performance measure for AD tools is given by the ratio of the time for performing one adjoint projection (i.e. computing the full set of first derivatives) to the time for one execution of the underlying program:

$$R = \frac{\text{(Time for one adjoint projection)}}{\text{(Time for executing underlying program)}} = \frac{time\,(\nabla f)}{time\,(f)}\ .$$

This is an efficiency measure and shows how much overhead is introduced by the AD tool in order to calculate the gradient. When dco is applied to real world programs, the ratio R typically varies between 5 and 15. For a given code, the best achievable R is highly dependent on the code efficiency (e.g. cache use) and the amount of effort a programmer is willing to invest to tune the adjoint code (dco has several run time and compile time options influencing memory use and run time). This is simply a trade-off between man hours and performance.

Some of dco's key features are:

- Generic derivative types: the dco data types can be instantiated with arbitrary base types (e.g. double/float)

- Higher order derivatives can be computed
- External function interface: this allows
  - Checkpointing (e.g. time stepping in PDE solution) to control memory use
  - User-defined intrinsics (e.g. smoothed pay-offs)
  - Low-memory adjoint ensembles (e.g. Monte Carlo) allowing the programmer to exploit parallelism
  - Supporting accelerators (e.g. GPUs) as is done in this paper
- NAG Library support: dco versions of any NAG Library routine can be made, allowing dco to treat the routine as an intrinsic

To summarise, in it's simplest form dco performs adjoint AD by the following steps:

1. Replace floating point data types in the program with dco data types
2. Run the program forward, recording optimised versions of intermediate calculations in the tape
3. Set $y_{(1)} = 1$ and play the tape back, effectively running the whole calculation backwards and computing the gradient

5.2 Adjoint AD on GPU accelerators

Readers familiar with GPU programming will know that the hardware constraints of these platforms are quite different from traditional CPU programming. In particular, there is a rather small pool of memory (around 6GB) shared by a relatively large number of threads (in a typical CUDA kernel one may have upwards of 40,000 threads). In addition L1 and L2 caches are rather small. This means that memory use is of primary importance when programming GPUs.

Unfortunately adjoint AD methods can require prohibitive amounts of memory. Moreover there are currently *no AD tools which support CUDA or OpenCL* – all GPU adjoint codes are hand-written (CUDA support for dco in tangent-linear mode is in development). For this reason, there are few examples of non-trivial adjoint codes running on GPUs. Mike Giles's LIBOR market model code (available on his website, see [3]) is the only one in the public domain of which we are aware.

6 Implementing adjoint AD for the FX basket option

We use dco for the CPU code, namely stages 1 and 3 in Section 3 above. At the various points where the code calls into the NAG C Library (e.g. for the interpolation routines), we use the dco enabled versions of those routines. Since dco in adjoint mode does not support CUDA, something different has to be done to handle the GPU Monte Carlo kernel. Here we make use of dco's external function interface. This effectively allows one to insert "gaps" in dco's tape and to provide "external functions" which fill those gaps.

When running the code forward, the Monte Carlo kernel becomes an external function. Stage 1 is computed with dco data types. At the start of stage 2 one extracts the values of the inputs (local volatility spline knots and coefficients, Cholesky factorisation of correlation matrix, domestic and foreign interest rates, etc) from the dco data types, copies them to the GPU, launches the kernel and copies the results back from the GPU to the CPU. The results are then inserted into new dco data types and the calculation proceeds to stage 3 as normal.

When the tape is played back a function must be provided which implements the adjoint model of the Monte Carlo kernel in stage 2. Since there is no AD tool support for CUDA, this function has to be written by hand. The Monte Carlo kernel is massively parallel. Therefore the adjoint model of it is also massively parallel (each thread computes the adjoint of one or more sample paths independently) and is well suited to GPU acceleration provided the memory use can be constrained. This adjoint GPU kernel is therefore run when the tape reaches stage 2: the input adjoints are extracted from the dco tape and copied to the GPU, the kernel is launched, the output adjoints are copied to the CPU and inserted into the dco tape, and the calculation continues as normal.

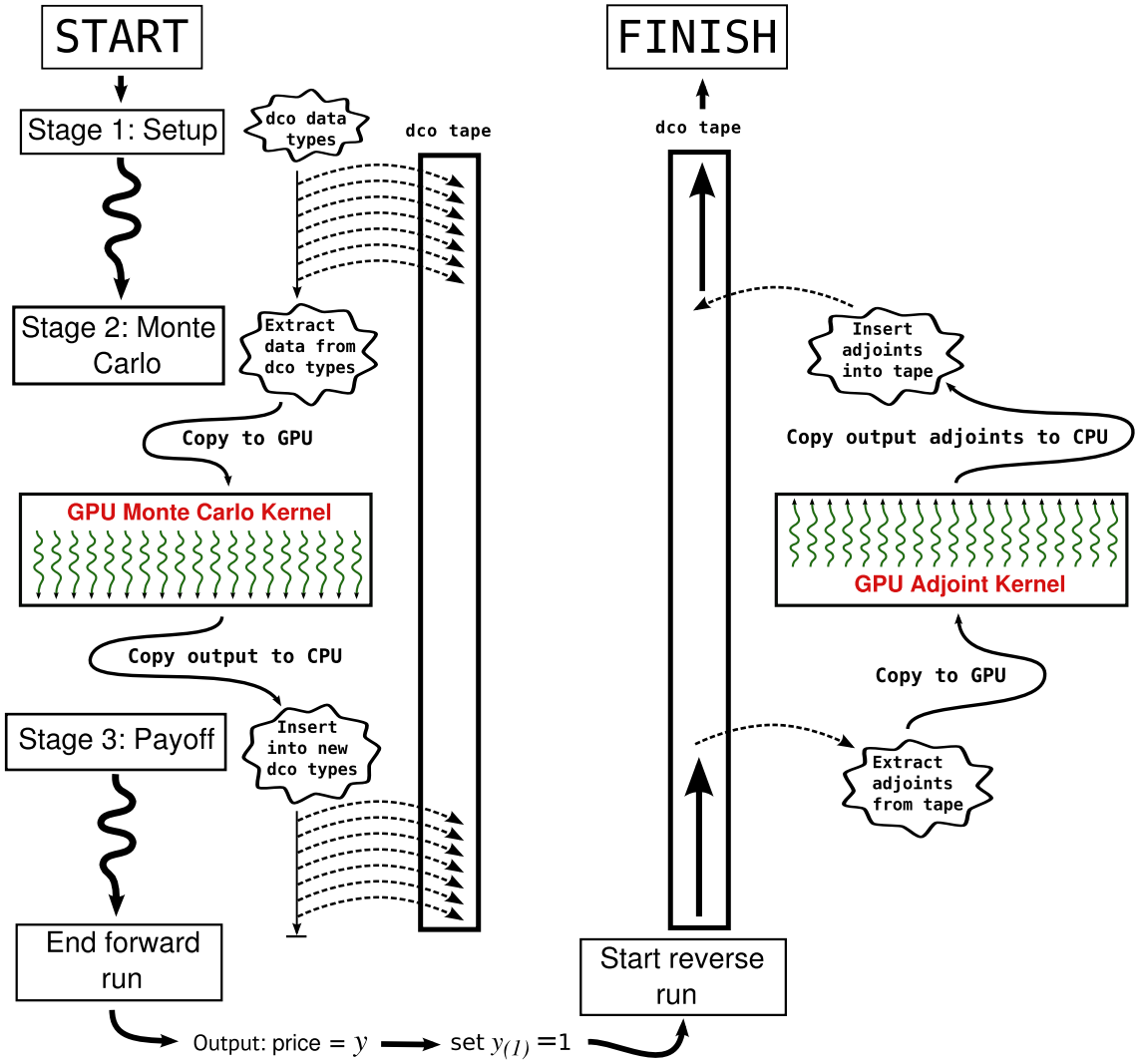

Figure 1: The control flow for the adjoint program. In stages 1 and 3, dco data types record to a tape. The tape has a gap for stage 2. To compute the adjoint the tape is played back, and the hand-written adjoint kernel fills in the gap left by stage 2.

While it may sound daunting to write an adjoint code by hand, we found that the key was to break the Monte Carlo kernel up into functions. Adjoint models of each of these functions can then be made, each of which is relatively simple, and the individual adjoint functions can then be strung together to make the adjoint kernel. The most difficult part was the cubic spline evaluation function[c] which consists of about 50 lines of C code. However once the adjoint model of this function has been made it becomes a library

[c]The GPU version of function e02bbc in the NAG C Library

component and is available for use in multiple projects, since the code for evaluating a cubic spline does not change.

6.1 Controlling memory use

As mentioned in Section 4 adjoint AD codes can use prohibitive amounts of memory to store all the intermediate calculations needed to run the program backwards. For a Monte Carlo code this can easily amount to tens or hundreds of gigabytes if some care is not taken. For GPUs it is a general rule of thumb that memory is expensive and flops are free, which suggests one should only store what is essential and recompute everything else.

Consider (6) above, suppress the superscript (i) for ease of notation and suppose that the calculation has to be run backwards from $\tau = n_T$ to $\tau = n_T - 1$ to compute adjoints. To get some feel for what this involves let us apply (9) to (6) in a simplified form and suppose that $y_{(1)}$ is the adjoint of $\log(S_{n_T})$ and $x_{(1)}$ is the adjoint of $\log(S_{n_T-1})$. (In reality $x_{(1)}$ is a vector $\mathbf{x}_{(\mathbf{1})}$ of the adjoints of r_d, $r_f^{(i)}$, the cubic spline data, $\log(S_{n_T-1})$ etc, but here we consider only one output for clarity). Then

$$\begin{aligned}\frac{\partial t}{\partial \log(S_{n_T-1})} &= \frac{\partial \log(S_{n_T})}{\partial \log(S_{n_T-1})}\frac{\partial t}{\partial \log(S_{n_T})} \\ &= \left(1 + \sigma\big(S_{n_T-1}, T-\Delta t\big)\frac{\partial \sigma\big(S_{n_T-1}, T-\Delta t\big)}{\partial \log(S_{n_T-1})}\right. \\ &\qquad \left. + \sqrt{\Delta t}\mathbf{dZ}_{n_T}\frac{\partial \sigma\big(S_{n_T-1}, T-\Delta t\big)}{\partial \log(S_{n_T-1})}\right)\frac{\partial t}{\partial \log(S_{n_T})}\,.\end{aligned} \tag{13}$$

From this it is clear that one cannot perform the adjoint calculation without knowing the value of $\log(S_{n_T-1})$ since $\sigma(x,t)$ is not invertible. Moreover knowing this value all other values can be computed (indeed this is how the original Monte Carlo kernel works, stepping forwards in time).

Therefore to compute the adjoint model all that has to be stored are the values $\log\big(S_\tau^{(i)}(j)\big)$ for all assets $i = 1, \ldots, N$, all time steps $\tau = 1, \ldots, n_T$ and all Monte Carlo sample paths j. With these values known, all other values can be recomputed. In our implementation we chose to store the local volatilities $\sigma^{(i)}\big(S_\tau^{(i)}(j), \tau\Delta t\big)$ as well since the spline calculations are rather expensive. All in all the test problem we considered (see Section 7 below) uses around 420MB of GPU memory.

6.2 Dealing with race conditions and mixed precision

Any parallelised adjoint model of a Monte Carlo kernel will have race conditions. This can easily be seen from (6) above: if the adjoint model is computed for two sample paths j and $j+1$ concurrently, then at some point both threads will be using the input adjoints $\log\big(S_{\tau+1}^{(i)}(j)\big)_{(1)}$ and $\log\big(S_{\tau+1}^{(i)}(j+1)\big)_{(1)}$ to update the adjoint of the domestic interest rate $r_{d(1)}$. If this is not done in a thread safe way, data corruption and/or incorrect results will follow. Of course, in this case each thread can simply have its own copy of $r_{d(1)}$ and at the end of the kernel these can be accumulated in a thread safe way with no difficulty.

The local volatility model, however, has a more unpleasant race condition when updating the adjoints of the spline data. Recall how cubic splines are evaluated. At its simplest a cubic spline consists of a set of knots $\lambda_1 < \lambda_2 < \cdots < \lambda_{n_\lambda}$ and a set of coefficients $c_1, c_2, \ldots, c_{n_c}$. To get the value of the spline at a point x (in (3) this is $\log(S_\tau^{(i)})$ which is essentially random) one must first find the index k such that $\lambda_k \le x < \lambda_{k+1}$. The spline value y is then a function f of several λs and several cs selected by the index k so that

$$y = f(x, \lambda_k, \lambda_{k+1}, \ldots, \lambda_{k+k_1}, c_k, c_{k+1}, \ldots, c_{k+k_2}) \tag{14}$$

Suppose that $k_1 = k_2 = 3$. From (8) the adjoint model of (14) is

$$\begin{bmatrix} x \\ \lambda_k \\ \lambda_{k+1} \\ \lambda_{k+2} \\ c_k \\ c_{k+1} \\ c_{k+2} \end{bmatrix}_{(1)} = y_{(1)} \nabla f(x, \lambda_k, \lambda_{k+1}, \lambda_{k+2}, c_k, c_{k+1}, c_{k+2}) \tag{15}$$

If at the same time another thread is performing the same calculation for some $\lambda_{k+1} \leq x' < \lambda_{k+2}$, then both threads could try to update the adjoints $\lambda_{k+1(1)}, \lambda_{k+2(1)}, c_{k+1(1)}$ and $c_{k+2(1)}$ simultaneously. Moreover x and x' are random – there is no way to predict when these race conditions will occur.

One way around this is to give each thread its own copy of the adjoints of the spline data. However for a GPU kernel with 40,000 threads this could require tens of gigabytes of memory, which is totally impractical. Another approach would be to give each thread its own copy in shared memory and then have each thread block synchronise its threads to combine the results in a safe way. The problem here is that one rapidly runs out of shared memory, or at best ends up with only one thread block per GPU streaming multiprocessor, which gives poor performance. A third option is to give each thread block its own copy of the data in shared memory and then try to do sophisticated inter-thread synchronisation to avoid race conditions. This would create a complicated code which would probably not perform very well. A fourth option is to use hardware atomic operations[d] to update the adjoints in a thread safe way. This approach works, but is rather slow (at least 4 times slower than our implementation).

The solution we chose is to do the GPU calculations in single precision and use a combination of a few atomic operations (purely for ease of programming – the algorithm can be implemented without them) and private arrays for active threads to compute the adjoints. (Note: the CPU calculations for stages 1 and 3 are still done in double precision). In [2] we showed the impact on performance and accuracy of single, double and mixed precision Monte Carlo code on GPUs. Switching from double to single precision halved the runtime of the application considered in this paper and gave results which were within single precision tolerances. In particular, the standard deviation of the Monte Carlo estimate was orders of magnitude greater than the threshold of single precision accuracy (roughly 10^{-6}). As always, care must be taken when working in single precision, especially when calculating things like adjoints where derivative contributions are continually accumulated (added to a variable). In our adjoint kernel we ended up computing $r_{d(1)}$ in mixed precision since the numerical value was rather small and unacceptable round off errors occurred in single precision.

We emphasise that the choice to perform the adjoint calculation in single precision was purely for performance. The calculation could also be done in double precision on the GPU.

6.2.1 A word of caution (*nil desperandum!*)

The problematic race condition described above only arises when large shared arrays are used to update Monte Carlo sample paths from one time step to the next. This is typically a feature of local volatility-type models, but there are many financial models (e.g. Heston) where this does not happen. For these models there are usually a few scalar variables

[d] Atomic operations are guaranteed to be thread safe: if two threads execute an atomic add on the same variable, the variable is guaranteed to contain the correct result

which govern the SDE, and the race conditions which arise are easily handled by giving each thread its own copy of the adjoints of the input variables, and then doing standard parallel reductions to combine the results. This is efficient and quite easy to do.

7 Results

Results are presented in terms of runtimes and accuracy of the derivatives when compared with a full double precision version of the code. As mentioned before, all CPU calculations are done in double precision while the GPU calculations are predominantly in single precision with one or two calculations in the adjoint kernel done in double precision.

7.1 Test problem

We considered a 1 year basket option on 10 currency pairs: EURUSD, AUDUSD, GBPUSD, USDCAD, USDJPY, USDBRL, USDINR, USDMYR, USDRUB and USDZAR all with equal weighting. The spot rates were taken from market data as of Sept 2012 and the correlation matrix was estimated. The interest rates, both foreign and domestic, and the strike price were taken to be zero in the interest of simplicity. There are 5 FX quotes per maturity and 7 maturities per FX rate, which means in total there are 438 input parameters, and hence 438 derivatives to compute.

In the Monte Carlo simulation we used 10,000 sample paths each with 360 time steps.

7.2 Hardware and compilers

The computing platform we used was as follows:

1. CPU: Intel Xeon E5-2670, an 8 core processor with 20MB L3 cache running at a base frequency of 2.6GHz but able to ramp up to 3.3GHz under computational loads. We used gcc version 4.4.6
2. GPU: NVIDIA K20X with 6GB RAM. The GPU clock rate is 700MHz, ECC is off and the memory bandwidth (as measured by the bandwidth test in the CUDA SDK) is around 200GB/s. We used the CUDA toolkit 5.5.

7.3 Runtimes and accuracy

For the problem outlined above the results are as follows:

- Overall runtime: 522.4ms
  - Forward calculation: 367.0ms
    - GPU random number generator: 4.5ms
    - GPU Monte Carlo kernel: 14.5ms
  - Computation of adjoints (backward calculation): 155.4ms
    - GPU adjoint Monte Carlo kernel: 85.1ms
- Memory copies to and from the GPU: $\approx$ 0.5ms
- dco used 268MB of CPU memory to construct the tape and in total about 420MB of GPU memory was used

For comparison, the reference CPU serial double precision program using dco in tangent-linear mode took over 2hrs to compute the 438 derivatives. Since the tangent-linear model has the same complexity as finite differences, one would expect a similar runtime for a finite difference calculation of the gradient.

A plot and log-plot of the errors of the mixed precision derivatives compared with the reference double precision derivatives is given in Figure 2. The x-axis is the log of the

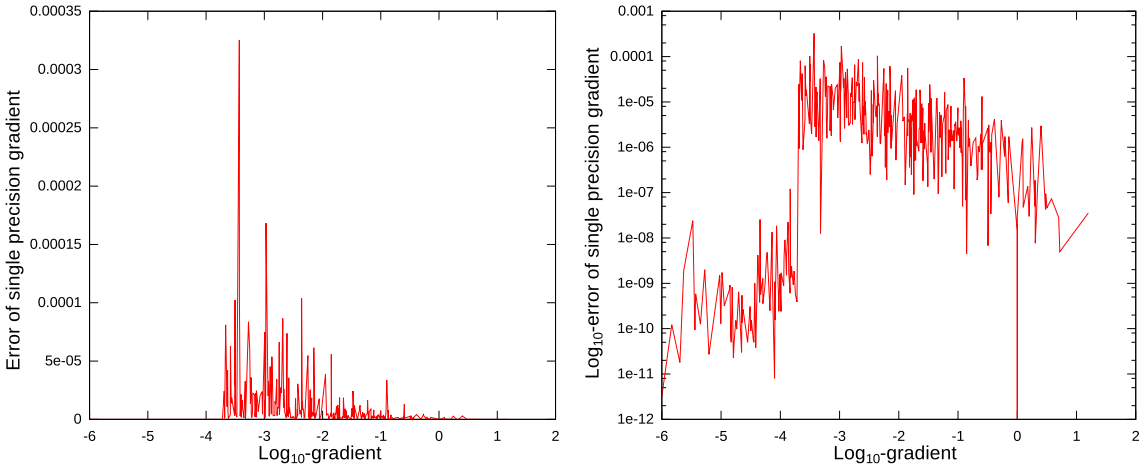

Figure 2: Plot and log-plot of the error of the gradient against the log of the gradient. The error is given by (16) and the logarithms are taken in base 10.

absolute value of the gradient and the error is defined as

$$\text{err} = \begin{cases} |g_{\text{ref}} - g_{\text{mp}}| & \text{if } |g_{\text{ref}}| < 0.0002 \\ \dfrac{\max\left(|g_{\text{ref}}|,|g_{\text{mp}}|\right)}{\min\left(|g_{\text{ref}}|,|g_{\text{mp}}|\right)} - 1 & \text{otherwise} \end{cases} \tag{16}$$

where g_{ref} is the reference double precision gradient and g_{mp} is the gradient from the mixed precision code. Note that many of the gradients are quite small in absolute value (there are 103 values less than 2×10^{-4}) and for these values we measure accuracy as an absolute rather than a relative difference. Only 5 out of 438 relative errors are greater than 10^{-4} and only 16 are greater than 5×10^{-5}. The accuracy is therefore pretty much what one would expect for a single precision code, especially seeing as most of the gradients have such small numerical values.

7.4 Scaling

We ran the same problem with 20,000 Monte Carlo sample paths and got the following results:

- Overall runtime: 611.6ms
- GPU Monte Carlo kernel: 28.902ms
- GPU adjoint kernel: 142.8ms
- dco used about 280MB of CPU memory for the tape

The GPU Monte Carlo kernel scales by a factor of 2 and the GPU adjoint kernel by a factor of 1.7, however the overall runtime scales by a factor of 1.17. This is due to the relatively expensive setup (stage 1) which accounts for around 66% of the runtime and is completely independent of the number of sample paths.

We then ran the problem with 10,000 sample paths but only 5 assets in the basket and obtained:

- Overall runtime: 283.4ms
- GPU Monte Carlo kernel: 6.6ms
- GPU adjoint kernel: 58.7ms
- dco used about 135MB of CPU memory for the tape

The runtimes now scale more or less by a factor of 2 apart from the adjoint kernel, which scales by a factor of 1.4. This is due to the synchronisation overhead of combining the adjoints in a thread safe way.

8 Acknowledgments

The authors would like to thank NVIDIA for providing access to their PSG cluster.

References

[1] Naumann, U. (2012) The art of differentiating computer programs. *Siam.*

[2] du Toit, J., Ehrlich, I. (2013) Local volatility FX basket option on CPU and GPU. *NAG Technical Report*, http://www.nag.co.uk/numeric/gpus/local-volatility-fx-basket-option-on-cpu-and-gpu.pdf

[3] Giles, M., Xiaoke, S. (2007) A note on using the nVidia 8800 GTX graphics card *Research Report*, available at http://people.maths.ox.ac.uk/gilesm/codes/libor/report.pdf

[4] Vandevoorde, D., Josuttis, N. (2002). C++ templates: the complete guide. *Addison Wesley.*