Adjoint Algorithmic Differentiation of a GPU Accelerated Application

Jacques du Toit (Numerical Algorithms Group) and Uwe Naumann (RWTH Aachen)

Local Volatility FX Basket Option by Monte Carlo

We consider a basket call option written on 10 FX rates given by

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

for $i = 1, \ldots, 10$ where r_d is the domestic rate, $r_f^{(i)}$ is the foreign rate and $(\mathbf{W}_t)_{t \geq 0}$ is a correlated 10-dimensional Brownian motion with $\langle W^{(i)}, W^{(j)} \rangle_t = \rho^{(i,j)} t$. The local volatility function $\sigma^{(i)}$ is calibrated from market implied volatility data. The price of the option is

$$P = e^{-r_d T} \mathbb{E} \left(\sum_{i=1}^{10} w^{(i)} S_T^{(i)} - K \right)^+$$

where $K > 0$ is the strike and the $w^{(i)}$s are a set of weights summing to one. P is computed by Monte Carlo simulation.

Input Parameters to the Model

Input parameters are the strike, maturity, weights, rates, correlation structure, and the 10 **market implied volatility surfaces**. In total there are **438 inputs** and the task is to compute sensitivities of the price with respect to all.

GPU Accelerated Code: CPU → GPU → CPU

We developed a 3 stage code to price the basket option as follows:

- Stage 1: Setup (on CPU, double precision). Process implied vol surfaces into local vol surfaces
- Stage 2: Monte Carlo (GPU, single precision). Compute sample paths
- Stage 3: Payoff (CPU, double precision)

The aim was to create a GPU accelerated adjoint version of this code using AD to compute the gradient.

Algorithmic Differentiation (AD) in a Nutshell

AD is a program transformation technique. It yields derivative code of up to arbitrary order. AD software tools have been developed to provide support to the application programmer. Refer to [1] for further details.

References

[1] Naumann, U (2012). *The Art of Differentiating Computer Programs*. SIAM

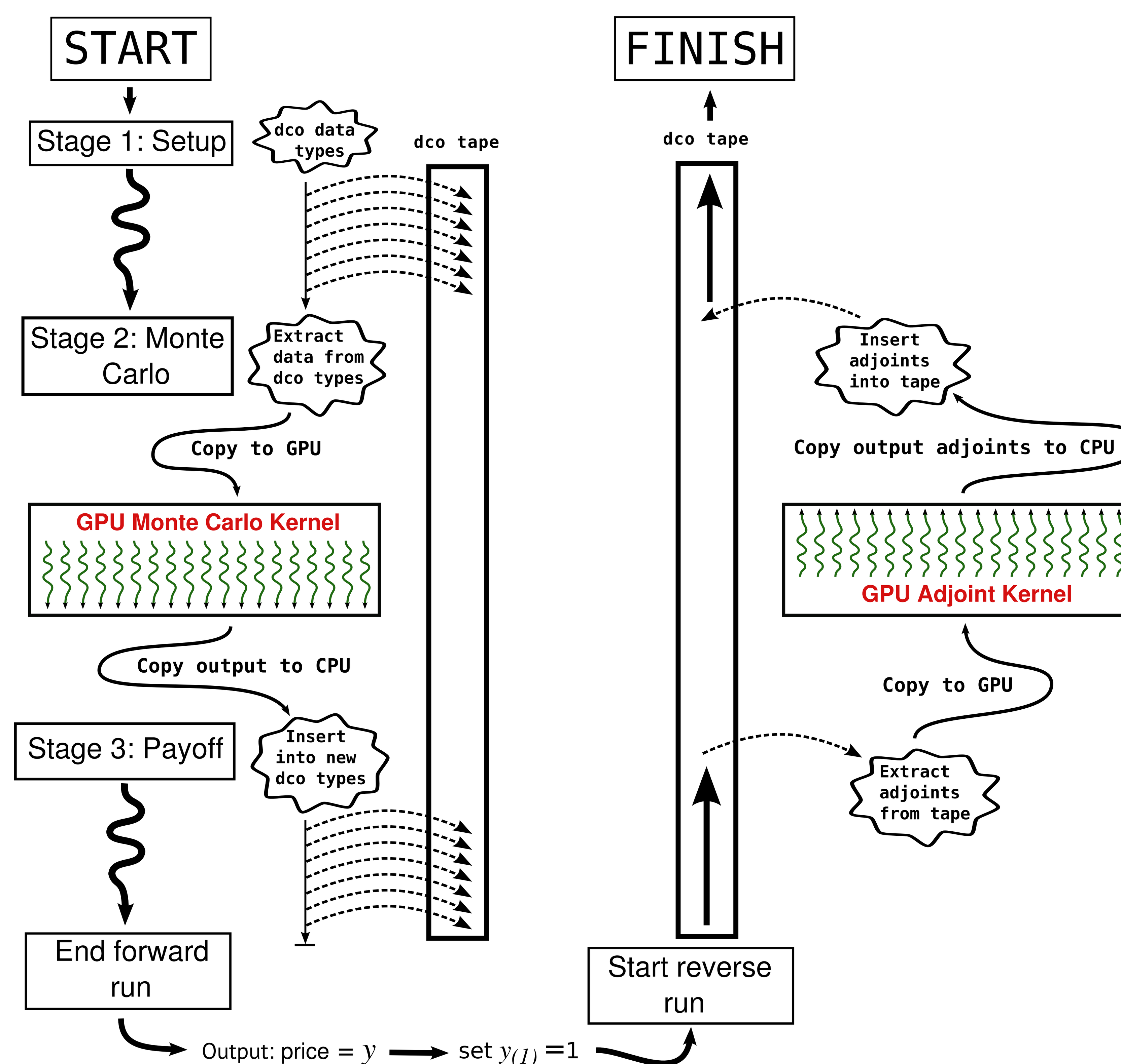
Adjoints, dco and GPUs

Adjoints are sophisticated numerical techniques for computing a large number of gradients quickly. To compute an adjoint, your computer program must be run **backwards**. dco is an AD tool that does this by

- Running the code forwards and storing intermediate values in a **tape**
- Playing the tape back and computing the adjoint

No AD tools support GPUs, so we made a hand-written GPU adjoint Monte Carlo kernel and used dco's **external function interface** to splice this into the tape. The adjoint program then becomes

- Stage 1: Use dco on CPU
- Stage 2: get values from dco, launch GPU Monte Carlo kernel
- Stage 3: get results from GPU and use dco on CPU
- Run dco tape back to get adjoint of Stage 3
- Tape gap left by Stage 2: copy adjoint from dco tape to GPU, run adjoint kernel
- Get results from GPU to dco tape, run tape back to finish



Results

Test Problem and Hardware

We used 10,000 Monte Carlo paths with 360 time steps per path. Implied volatility surfaces were estimated from market data. Hardware was a dual socket Intel Xeon E5-2670 with an NVIDIA K20X.

GPU Accelerated Adjoint vs Tangent Linear (bump and revalue)

Figure 1 shows the relative speeds of a parallel (16 threads) CPU tangent linear version of the code against our GPU adjoint. GPU bar on the left has value 1, the CPU code took 860 times longer. Tangent linear models have the same computational complexity as bump and revalue (finite differences).

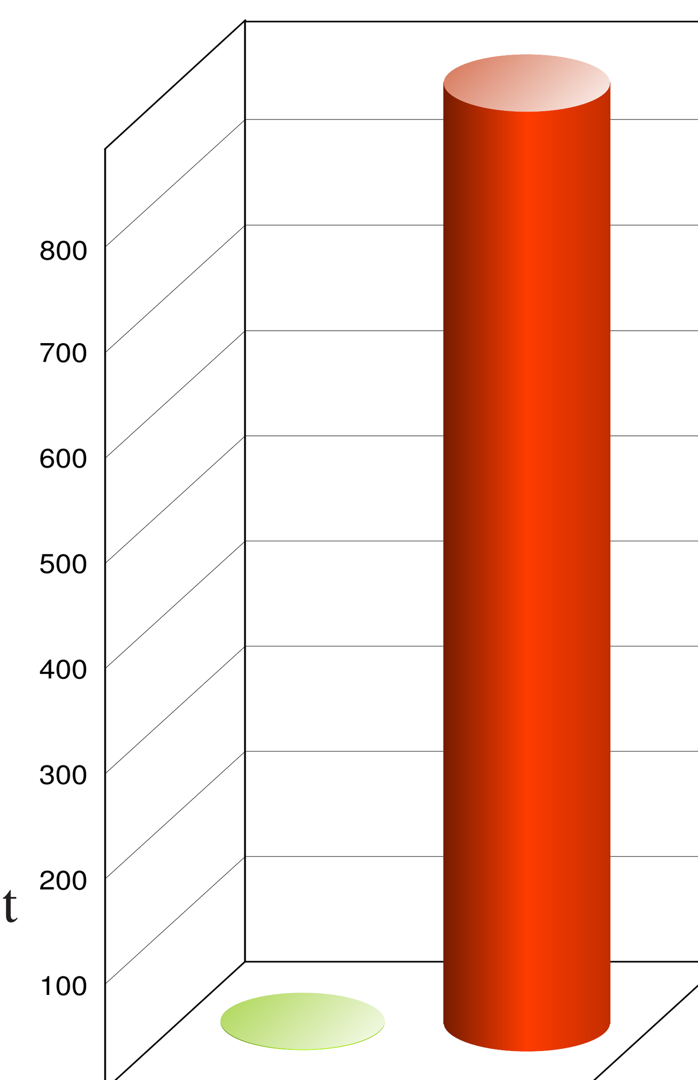


Figure 1: Runtime of parallel CPU tangent linear code (right) relative to GPU adjoint (left, value=1)

Performance Details of GPU Adjoint Code

Code Portion	Runtime	
Overall	523.2ms	(code used 420MB GPU memory)
Forward Run	371.9ms	(of which GPU MC kernel = 14.6ms)
Reverse Run	151.3ms	(of which GPU Adjoint kernel = 85.1ms)

Accuracy of Mixed Precision Code

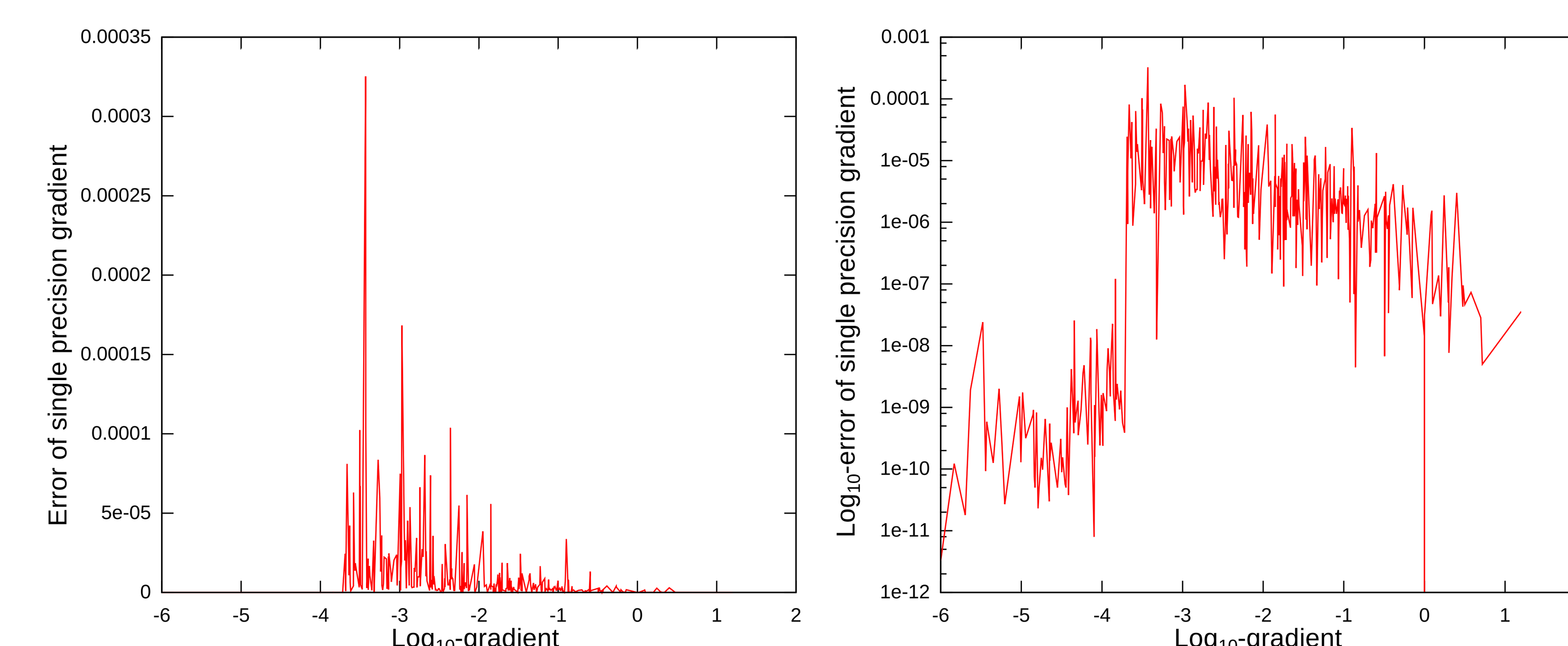


Figure 2: Plot and log-plot of error against log-gradient relative to double precision tangent linear results. If the double precision gradient is less than 0.0002 an absolute error is taken, otherwise a relative error is taken. Note the number of very small gradients. The results are what one would expect for a single precision code – in particular only 16 values are larger than 5.5×10^{-5} and only 5 are larger than 10^{-4}.

Corresponding author: Jacques du Toit (jacques@nag.co.uk)