NAG for HPC in Finance

John Holden
Jacques Du Toit

3rd April 2014

Computation in Finance and Insurance, post Napier
Agenda

- NAG and Financial Services
- Why do Quants love NAG?
  - Problems in numerical computation
- Achieving acceleration with
  - New hardware and new algorithms - Case study: AD of GPU accelerated application
NAG Background

- **Founded 1970**
  - Not-for-profit organisation
  - Surpluses fund on-going R&D

- **Mathematical and Statistical Expertise**
  - Numerical Libraries of components
  - Consulting

- **HPC Services**
  - Computational Science and Engineering (CSE) support
  - Procurement advice, market watch, benchmarking
Financial Services

- Many clients in FSI
- Most Tier 1 Banks have licences, > 60% have global licences
- Typically the NAG Library is embedded in the banks own “quant” libraries (C++, .NET, ..)
- HPC Services - training, code porting, tuning, re-writing
NAG Library and Toolbox Contents

- Root Finding
- Summation of Series
- Quadrature
- Ordinary Differential Equations
- Partial Differential Equations
- Numerical Differentiation
- Integral Equations
- Mesh Generation
- Interpolation
- Curve and Surface Fitting
- Optimization
- Approximations of Special Functions

- Dense Linear Algebra
- Sparse Linear Algebra
- Correlation & Regression Analysis
- Multivariate Methods
- Analysis of Variance
- Random Number Generators
- Univariate Estimation
- Nonparametric Statistics
- Smoothing in Statistics
- Contingency Table Analysis
- Survival Analysis
- Time Series Analysis
- Operations Research
Why Quants use NAG Libraries and Toolboxes?

- Global reputation for quality – accuracy, reliability and robustness...
- Extensively tested, supported and maintained code
- Reduces development time
- Allows concentration on your key areas
- Components
  - Fit into your environment
  - Simple interfaces to your favourite packages
- Regular performance improvements!
Why bother?

- Numerical computation is difficult to do accurately
- Problems of
  - Overflow / underflow
  - Condition
  - Stability
- Important people / organisations get in wrong
  - AMD,
  - Microsoft
  - and even
  - Famous Quants
Why an Exotics Options Quant loves NAG?

- **General Problem**
  - To build solvers for a variety of sophisticated financial models in a timely manner that are
    - robust,
    - stable,
    - quick

- **Solution**
  - Use robust, well tested, fast numerical components
  - This allows the “expensive” quants to concentrate on the sophisticated models
    - avoiding distraction with low level numerical components
Problem: Calibration of Options

- Major banks all need to calibrate their models

- Several different numerical components needed
  - Optimisation functions (e.g. constrained non-linear optimisers)
  - Interpolation functions, Spline functions
  - FFTs, Quadrature, Root Finders..
  - Special functions (Bessel, non-Central Chi, Erf,..)
  - Probability distributions
Problem: Calibration of Options

- Major banks all need to calibrate their models

  *NAG to the rescue*

- Several different numerical components needed
  - Optimisation functions (e.g. constrained non-linear optimisers) ✓
  - Interpolation functions, Spline functions ✓
  - FFTs, Quadrature, Root Finders.. ✓
  - Special functions (Bessel, non-Central Chi, Erf,..) ✓
  - Probability distributions ✓
To get **acceleration** look at the algorithms instead of the hardware

- Even better combine both.

Implementing Adjoint AD algorithms reduces runtime

With Prof. Naumann & RWTH Aachen University NAG are delivering Algorithmic Differentiation (AD) tools and services to the finance community for C and C++ codes.
Algorithmic Differentiation of a GPU Accelerated Application

Jacques du Toit

Numerical Algorithms Group
What is Algorithmic Differentiation?

It’s a way to compute

\[
\frac{\partial}{\partial x_1} f(x_1, x_2, x_3, \ldots) \quad \frac{\partial}{\partial x_2} f(x_1, x_2, x_3, \ldots) \\
\frac{\partial}{\partial x_3} f(x_1, x_2, x_3, \ldots) \quad \ldots
\]

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

\[
\begin{align*}
\text{if}(x_1 < x_2) \quad & \text{then} \\
\quad & f = x_1 \times x_1 + x_2 \times x_2 + x_3 \times x_3 + \ldots \\
\text{else} \quad & f = \sin(x_1 + x_2 + x_3 + \ldots) \\
\text{endif}
\end{align*}
\]
Why use Algorithmic Differentiation

AD computes **exact** derivatives up to machine accuracy, and adjoint AD is fast for large gradients!

Case study: Equity call option driven by local volatility model, PDE pricing method (250 × 500 mesh)

<table>
<thead>
<tr>
<th></th>
<th>F.D.</th>
<th>AAD</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \nabla f \in \mathbb{R}^{422} )</td>
<td>98s</td>
<td>12s</td>
<td>8.2x</td>
</tr>
<tr>
<td>( \frac{\partial^2 f}{\partial S_0 \partial x} \in \mathbb{R}^{422} )</td>
<td>200s</td>
<td>24s</td>
<td>8.3x</td>
</tr>
<tr>
<td>( \nabla^2 f \in \mathbb{R}^{422 \times 422} )</td>
<td>20,000s</td>
<td>1,338s</td>
<td>14.9x</td>
</tr>
</tbody>
</table>
Local Volatility FX Basket Option

Our function $f$ is the price of a basket call 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
- Model has 438 input parameters, i.e. $f : \mathbb{R}^{438} \rightarrow \mathbb{R}$

$f$ uses a GPU to generate the Monte Carlo sample paths

Plan: compute $\nabla f$ (1st order Greeks) as quickly as possible using AD

- Differentiate through whatever procedure is used to turn the implied vol quotes into a local vol surface
- Use the GPU for any 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 $(W_t)_{t\geq 0}$ is a correlated $N$-dimensional Brownian motion with $\langle W^{(i)}, W^{(j)} \rangle_t = \rho^{(i,j)} t$.

The function $\sigma^{(i)}$ is given by the Dupire formula

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

where $\theta \equiv \theta(K, T)$ the market observed implied volatility surface. The basket call option price is then

$$
f = e^{-r_d T} \mathbb{E} \left( \sum_{i=1}^{N} w^{(i)} S_T^{(i)} - K \right)^+.
$$
Algorithmic Differentiation in a Nutshell

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

- A computer program implementing \( f \) 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 and operator overloading give a way to do all this efficiently and non-intrusively
Adjoints in a Nutshell

Adjoint (or reverse) AD is as follows: heuristically

- It computes the full gradient $\nabla f$ at once, simultaneously
- In adjoint mode have to run code forward and store all intermediate calculations
- Then run code backwards and use intermediate calculations to build up $\nabla f$ as you go
- Can prove that adjoint uses at most 5 times as many flops as $f$

Adjoints are extremely powerful: can give large gradients at potentially very low cost, \textit{provided} you have enough memory to store all intermediate calculations

We want to use adjoint mode for our code – the gradient is big
How do you implement adjoint AD?

- For simple code, pretty straightforward to do by hand (strict set of rules to follow)
- For large code (such as ours), infeasible to do by hand
- Have to use **AD tools**
- There are many AD tools for CPUs: we use **dco** developed by RWTH Aachen
- There are no AD tools for GPUs

**dco** has internal tape to store intermediate calculations, and allows you to insert **gaps** in the tape with its **external function API**
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.
  - Can use `dco` since it’s CPU code
  - Several calls to NAG Library routines (`dco` understands NAG Library!)

- **Stage 2: Monte Carlo (on GPU)** — copy local vol surfaces to GPU and create all the sample paths
  - Have to write adjoint by hand

- **Stage 3: Payoff (on CPU)** — get final values of sample paths and compute payoff
  - Can use `dco` since it’s CPU code

`dco` can run CPU code backwards and compute adjoints. What about GPU?
Running Monte Carlo Backwards

The Euler-Maruyama discretisation is

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

- At Monte Carlo time step \( i \) we need to know \( S_i \) to compute \( S_{i+1} \)
- 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 \( \sigma \) is not invertible

So what does this mean?

- In forward run, is sufficient to store \( S_i \) for all sample paths and all Monte Carlo time steps
- From these, all other values for adjoint calculation can be recomputed
Writing an adjoint by hand?? That sounds horrid!

- Monte Carlo kernels are typically relatively simple
- In this case, most onerous part was writing an adjoint of a cubic spline evaluation function
- This is about 150 lines of code: just follow the rules mechanically
- The adjoint of the Monte Carlo kernel is massively parallel and can be performed on the GPU as well

Lastly we use dco’s external function API to splice the hand-written GPU adjoint into dco’s internal tape
AD on GPU
Jacques du Toit
Introduction
Local volatility FX basket option
Algorithmic Differentiation
Basket Option Code
Results

START

Stage 1: Setup

dco tape

dco data types

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 \rightarrow set y(1) = 1

FINISH

Start reverse run

dco tape

Insert adjoints into tape

Copy output adjoints to CPU

GPU Adjoint Kernel

Copy to GPU

Extract adjoints from tape
Test Problem and Results

As a test problem we took 10 FX rates

- 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

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 Really Nasty Race Condition

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 $\sigma(S_i, i \times \Delta t)$ 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 of spline data
- 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 its 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 \texttt{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.

But what we really want is for \texttt{dco} to support CUDA.

This is work in progress, watch this space!
In Conclusion

NAG has a wealth of experience in HPC libraries, services, consulting and training

- We are keen to collaborate with customers in building models and risk engines
- Requirements are likely to be varied across FSI
- Talk to us! We want to make sure we know what you need
- The importance of risk analysis is growing and will involve a LOT of computation (Basel III, Solvency II, CVA/DVA, ...)
- We know how to do large scale computations efficiently

This is non-trivial! Our expertise has been sought out and exploited by organisations such as (AMD, HECToR, Microsoft, Oracle, banks, oil & gas companies, ...)

In Conclusion