CVA on an iPad Mini
Part 4: Cutting the IT Edge

Aarhus Kwant Factory PhD Course

January 2014

Jesper Andreasen
Danske Markets, Copenhagen
kwant.daddy@danskebank.com

Outline

- Intro.

- Multi threading:

  - Why parallel computing is hard.
  - Hardware and GPUs versus CPUs.
  - C++ standard.

- Adjoint differentiation:

  - The magic of adjoint differentiation.
  - C++ implementation: templates and tapes.
  - Memory limitations and gate checking.
  - What else can go wrong?

    - AD in finance.

- Summary.

Intro

- So we have achieved fast CVA calculation using all the trickery we can do with models and CVA calculation algorithms.
- Two questions?
- Can it go faster?
- What about risk reports, i.e. how the CVA values change as market prices (rates, prices, volatilities) change?
- To go even faster on the pricing we are going to use MT, i.e. use multiple computer cores in parallel.
- To do fast risk we are going to do the magic AD in conjunction with MT.

Parallel Computing

- Many financial calculations are (or appear) trivially parallelizable, i.e. can be computed over many computer cores at the same time.
- Most obvious example is Monte-Carlo simulations where the different paths are delegated out on different computer cores.
- However, the industry has generally not used real parallel computing in large scale.
- There is long list of reasons for this:
- Typically, parallelizing over trades give the same benefits – for big nightly runs, at least. And this is a very simple trick to use.

- Writing parallel code is actually a lot harder than one might think.
- The benefits obtained by improving algorithm have traditionally (and still do) generally outperform the benefits of more massive computing.

Parallel Algorithms

- It is difficult to write parallel code and even more so for highly optimized and sophisticated algorithms.
- Say for example that we are using Sobol sequences for Monte-Carlo simulation.
- We want to spread the 1024 simulations on 8 cores with 128 simulations on each core so that core 0 does simulation 0 to 127, core 1 does simulation 128 to 255, etc.
- So we need to be able to skip in the Sobol sequence.
- My guess is that less than 20 people on the planet would know how to do this. More can figure it out … but still.

- Here is the solution, btw:

```
//    skip ahead
void
kSobol::skipAhead(
      unsigned long long      skip) //    Argument = number of entries to skip
{

      //    Check skip
      if (!skip) return;

      unsigned int i;

      //    Reset Sobol to entry 0 (not 1, hence must reset even though reset has already been called in init)
      for(i=0;i<mySobolDim;++i) myIntegerSequence(i) = 0;
      myRandomGenerator.reset();    //    Also reset the random generator

      //    The actual Sobol skipping algo
      unsigned long long im = skip;
      unsigned int         two_i = 1, two_i_plus_one = 2;

      i = 0;
      while (two_i <= im)
      {
            if ( ( (im + two_i) / two_i_plus_one ) & 1 )
            {
                  for (unsigned int k=0; k<mySobolDim; ++k)
                  {
                        myIntegerSequence( k) ^= myDirectionIntegers( k, i);
                  }
            }
```

```
            two_i <<= 1;
            two_i_plus_one <<= 1;
            ++i;
        }

        //    End of skipping algo

        //    The random generator must also skip
        if(mySimDim>mySobolDim) myRandomGenerator.skipAhead( skip * (mySimDim - mySobolDim));

        //    Update next entry
        mySimCount = unsigned long(skip);
        next();
}
```

Non Thread Safe Code, Tools, and Memory

- Another reason is that, existing code has been highly optimized for single threaded use.

- This generally means that quants throw out all the good advice of Bjarne Stoustrup about C++ standards – and make their code non-thread-safe – but fast.

- Thread safe means that memory can be read from different cores but different cores should not write to the same memory at the same time.

- Thread safe code is normally obtained by ensuring that objects don't change their internal state during valuations.

- The non-thread-safeness is most often the result of using caches of intermediate values to speed calculations.

- To rewrite non-thread safe code as thread safe, ie eliminate the caches, without performance hits can be very hard and very few people can do it.

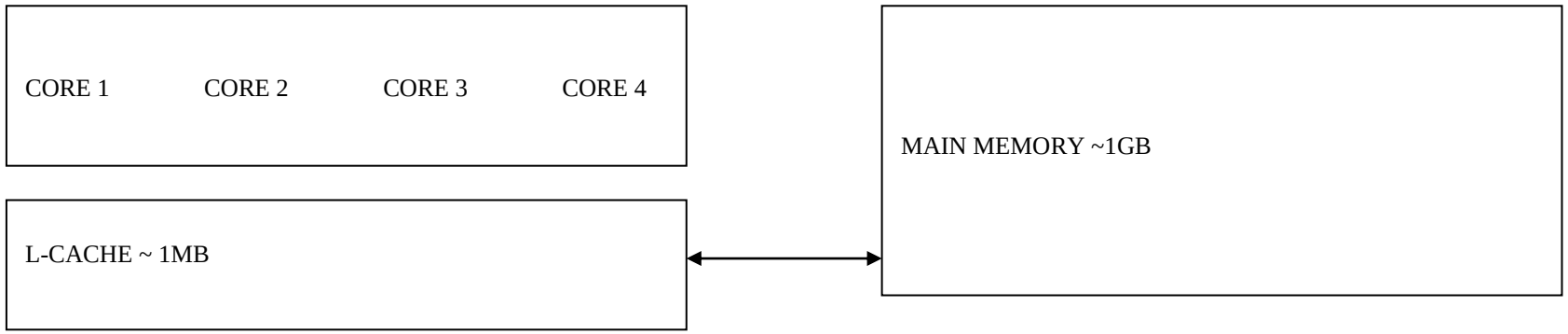
- The tools for writing multi-threaded code are poor and non-standard:

  - One often needs to use hardware and operative system specific features.
  - Debuggers are poorly developed for MT code.

- Hardware is optimized for single threaded calculations, and at some stage, memory handling and memory bandwidth becomes the bottle neck.

Hardware and Computers

- A computer looks like this



- The cores do the calculation and can do so very quickly on the data that is in the cache.

- The stuff that is the main memory has to go into the cache before it can be computed on.

- A CPU based computer has 4-6 cores, 1-10MB cache, 8-100GB of main memory, and a quite effective operative system for automatically moving memory between main memory and the cache.

GPUs

- A GPU based computer has 16-1024 cores, ***1-16KB*** cache, and 8-100GB of main memory, but no operative system for moving memory between the main memory and the cache.
- On a GPU, the memory handling has to be manually coded by the programmer.
- So the GPU has a lot of horse power but almost no fuel tank.
- This means that it is extremely resource demanding to code anything but the most simple models and interfaces on GPUs.
- Getting something like Jive on the GPU would be almost impossible.

- A new programmer is needed every time you want to do a new product.
- Further, the technology changes all the time. New versions of the GPUs come out and they require new code and the old code will not run on them.
- Besides that, best speed performance that people report is of a magnitude of a factor 50 – not 1000.
- We tried the GPUs but ended concluding that they are not compatible with our code and model philosophies.

CPU

- 128 core CPU machines exist and they are about the size of a small oven. It would cost you around 500,000 DKK.
- Similar GPU power would probably cost you around 50,000 DKK and the physical size would be only a tad smaller.
- So need to buy ***a lot*** of computer power to balance hardware savings against skilled programmer salaries.
- Further, the new C++ standard as of 2011 includes sophisticated MT features in the C++ language.

- This means that C++ MT development done on Windows will be transferable to Linux, Unix, Mac, Android, whatever – provided that these platforms have modern compilers.

- This includes shared memory, ie that the different cores simultaneously work on (read from) the same memory.
- So use of CPUs is the direction that we have taken.

- But we have not gone massive scale yet. Our biggest machine has 16 cores.

CVA and Multi Thread'ing

- Currently, we are using MT'ing for Monte-Carlo simulation (including Jive), Longstaff-Schwartz regression and some calibration.
- All Jive parsing, processing, compression, and decoration is still single threaded.
- For risk, we have MT'ed the AD based simulation but not the "Jacobian" calculations.
- So in the examples I've shown, there are still some speed ups to be made.
- It took us about 3 months to MT the (non-AD) Beast simulations.
- The rest of the time has been spent on AD and risk.

The Magic of Adjoint Differentiation

- Consider a function

$$f : \mathbb{R}^n \to \mathbb{R}$$

- … and the value

$$y = f(x)$$

- The magic of AD allows you to compute all the n derivatives

$$\frac{\partial y}{\partial x_i}, i = 1, \ldots, n$$

- … at the (theoretical) computational cost of ***4 times*** the cost of computing y.

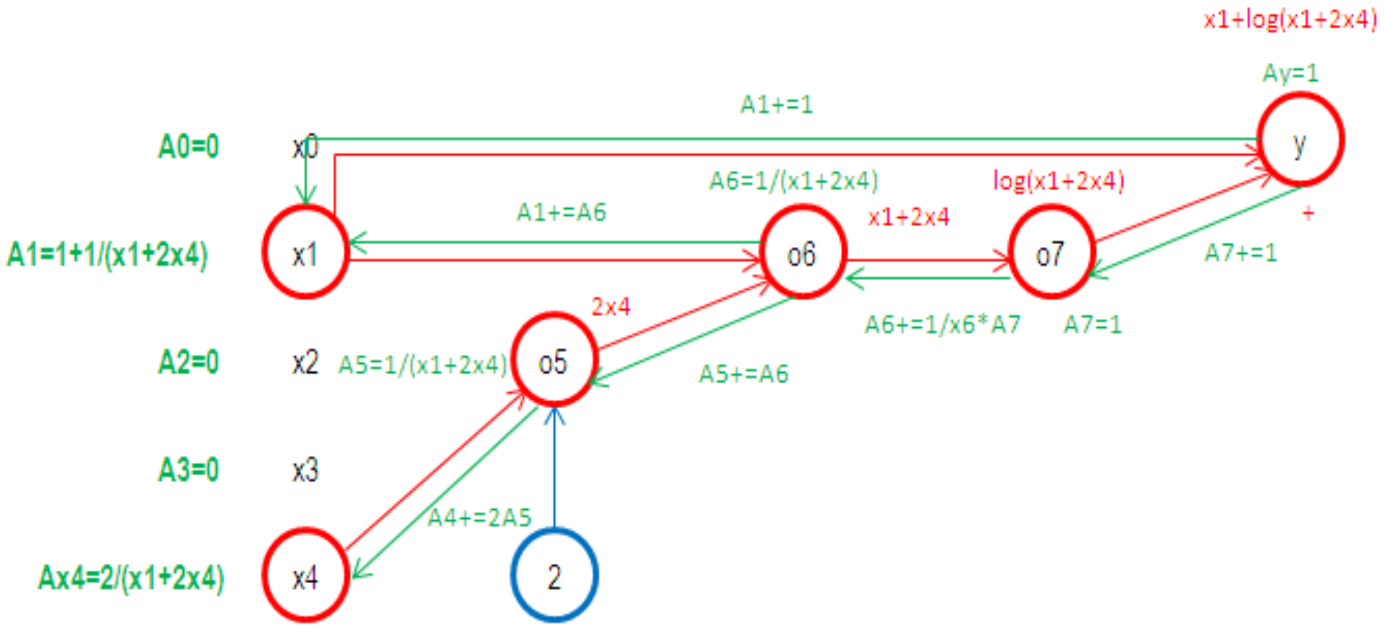
- In practice it is more like a factor 8 than a factor 4.
- So if n=1000 and it takes one second to compute the value of y then it will take only 8 seconds to compute all the 1000 partial derivatives $\partial y/\partial x_1,\ldots,\partial y/\partial x_{1000}$.
- That is magic and nothing less than a game changer.
- First of all, because in our industry 95% of the calculation time is spent on calculating risk reports, ie partial derivatives, by bumping the inputs and re-computing prices.
- Secondly, it opens up for models we have not really attempted to do before on an industrial scale.

- For example:

  - Calibration by Monte-Carlo.
  - All sorts of variations on non-linear pricing: transaction costs, feedback effects, etc.

- So how does it work?

- y=**x1**+log(x1 + 2 * x4)



Templates and Tapes

- AD can be implemented by hand coding the adjoint derivative calculation.
- This is the methodology used in some places including Credit Suisse, Nomura, OpenGamma.
- It is, however, a quite manual procedure and not particularly SuperFly.
- C++ has two build-in features that enables the implementation in a very elegant way.
- These are called ***templates*** and ***operator overload***.

- First step is to templatise the code a la:

```
//   templated bs call function
template <class T>
T    callT(T expiry, T strike, T forward, T sigma)
{
     T v = sigma*sqrt(expiry);
     T x = log(forward/strike)/v + 0.5*v;
     T c = forward*normalCdf<T>(x) - strike*normalCdf<T>(x-v);

     //   done
     return c;
}
```

- Under normal use we would use the function as

```
double expiry  = 2.0;
double strike  = 1.0;
double forward = 0.9;
double sigma   = 0.15;
double value   = callT<double>(expiry,strike,forward,sigma);
```

- When you compute derivatives

```
//  records tape -- forward
kDoubleAd expiry  = 2.0;
kDoubleAd strike  = 1.0;
kDoubleAd forward = 0.9;
kDoubleAd sigma   = 0.15;
kDoubleAd value   = callT<kDoubleAd>(expiry,strike,forward,sigma);

//  now do the ad -- backward
kAd ad(false);
ad.initDerivs();      //  sets all derivs = 0
value.deriv() = 1.0; //  sets deriv on value = 1
ad.evalBwd(value);    //  run tape backward

//  pick up results
double dExpiry  = expiry.deriv();
double dStrike  = strike.deriv();
double dForward = forward.deriv();
double dSigma   = sigma.deriv();
```

- What happens in this code is that the object kDoubleAd is constructed so that it automatically records all operations on a piece of static memory, the so-called ***tape***.

- This is done through by using another feature of C++ called operator overload.

- That is, the object kDoubleAd has +-*/ operators and functions such as sqrt(), exp(), log(), etc that replace (overload) the standard ones that work on double.

- The code for these operators and functions will automatically put the operations on tape.

- This happens in the function call callT<kDoubleAd>().

- When the tape been recorded, we set the derivatives on the variables and run the tape in “reverse”, ad.evalBwd().

- This corresponds to the backward sweep to obtain the derivatives in our simple example.

- Once the backward sweep is performed, the derivative values can be read off the kDoubleAd variables, as in for example double dSigma = sigma.deriv().

- Actually, what happens is that the kDoubleAd contains a pointer to where on the tape it sits.

Memory Limitations and Gate Checking

- So *in principle*, it is possible to simply templatize existing C++ code and combine it with some kDoubleAd object code.
- Then record one big tape in forward mode and compute all derivatives in backward mode.
- This, however, would only work for the most simple functions and models.
- The problem is that computers are so fast that they can fill up ***lots*** of memory very fast.
- A single core will write approximately 1GB of tape per second.

- Even if you can have 1GB of memory in your computer, you can only really efficiently access it if is in the cache, which is ~ 1-10MB.

- The solution to this to break up the problem of computing to a chain of (forward) calculations

$$x^0 \mapsto x^1 \mapsto \ldots \mapsto x^{n-1} \mapsto x^n$$

- Doing so is not only a trick to handle memory, it also facilitates caching of intermediate values in the total calculation.

- The requirement is that the code is broken up so that each step

$$x^i \mapsto x^{i+1} \quad (\mathbb{R}^{n_i} \to \mathbb{R}^{n_{i+1}})$$

- … is ***Markov***. In the sense that x^{i+1} need to be produceable from x^i only, i.e. there exists some function $f^i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_{i+1}}$ so that

$$x^{i+1} = f^i(x^i)$$

- If the end value y is an element of x^n, i.e. $y = x^n_j$, then we start the recursion by setting the derivatives

$$\frac{\partial y}{\partial x^n_i} = 1_{i=j}$$

- For each step $x^{k-1} \mapsto x^k$ we now use the template/tape machinery combined with the initial boundary condition $\partial y / \partial x^{k-1} = 0$ to produce

$$\frac{\partial y}{\partial x^{k-1}}$$

- The chain is now run backwards step by step to produce the final result

$$\frac{\partial y}{\partial x^{0}}$$

- Note that in each step we do ***not*** explicitly produce the Jacobian matrix

$$\frac{\partial x^{k}}{\partial x^{k-1}} \in \mathbb{R}^{n_k \times n_{k-1}}$$

- Instead over each step we start with one vector $\partial y / \partial x^{k}$ and produce another vector $\partial y / \partial x^{k-1}$.

- This technique also works with finite difference grids and it naturally breaks down the code in thread safe blocks that potentially can be multi threaded.

AD of Monte-Carlo Simulations

- Suppose we produce the value of some product y by Monte-Carlo simulation

$$y = \frac{1}{N} \sum_{j=1}^{N} g(\omega_j; x)$$

- … where x is a vector of model parameters and $\omega_1, \ldots, \omega_N$ index different paths of a Brownian motion.

- It does not seem memory efficient to compute $\partial y / \partial x_i$ the derivatives in one go.

- Clearly, it seems more memory efficient to compute the derivatives $\partial y / \partial x_i$ path by path and then take the average. That is,

$$\frac{\partial y}{\partial x} = \frac{1}{N} \sum_{j=1}^{N} \frac{\partial g(\omega_j; x)}{\partial x}$$

- So the tape is rewound and re-recorded for each path of the Brownian motion ω_j.

- Further, we note that the size x can be very large.

- If for example x is the parameters of a finite difference grid of dimension $(t \times s \times z) = \mathbb{R}^{100 \times 100 \times 100}$ then the x vector is of dimension $O(10^6)$.

- In this case we do not want to visit all elements of the vector x to record their derivatives.

- For a particular path ω_j, we only want to visit a particular element x_i and pick up its derivative if the parameter x_i has been hit over this path.

- We have developed C++ machinery that puts the parameters that are hit over a particular path on a stack.

- Normally, the kDoubleAd contains a pointer to an operation on the tape.

- But this is the reverse: a stack whose elements points back to the kDoubleAd.

Anything Else?

- Quite a lot, actually. Here are some examples.
- Values can be correct without the derivatives being so.
- The trouble is that (very) often you record a long tape in forward mode but then you don’t run into trouble before you start computing the derivatives.
- What happens here is typically that parameters have the right value from before the last time the tape was rewound but the wrong derivative.
- Effectively, the parameter never went on tape in the present calculation.
- Spotting where this happens can be a nightmare because you can’t see it in a debugger.

- The way to check this is to make sure that all inputs to operators and functions are already on tape.

- This check costs but you can set yourself up so that this is only checked when running in debug.

- It is a bit like the bounds checker that is used for vectors classes in C++.

- Another classic: functions may be perfectly well-defined on a closed interval but their derivatives are not, for example

$$f(x)=\sqrt{x}:\ [0,\infty[\rightarrow[0,\infty[$$
$$f'(x)=\frac{1}{2\sqrt{x}}:\ ]0,\infty[\rightarrow]0,\infty[$$

- Had to replace the use of the STL std::complex with a new homemade kKomplex class due to this.

- When you want to multi thread AD calculations for example for individual paths in Monte-Carlo then you need a tape for each thread.

- Implementing this is in itself not trivial and it puts further constraints on the cache.

- Generally, it isn’t computing the risk reports with respect to a monster many parameters in that take the coding time.

- Where most of the hard work is spent is on curling these monster many risk parameters back to the conventional risk reports that we use in trading.

AD in Finance

- The idea of AD is, as far as I know, a relatively new technology that dates back only a few decades. But in finance it is even younger.

- The first use of AD in finance that I know of was David Price at Bank of America, Chicago, who in 2002 implemented an ultra fast yield curve construction method based on tapes and overloads.

- The technology started gathering interest when Giles and Glasserman received the quant of the year award in 2006 for a paper on AD applied to a Libor Market model.

- Their code was based on classes created by Ole Stauning, who had written a PhD on AD and other computational methods in mathematics at DTU early 2000s.

- Interestingly, I have worked with both David Price at BofA and Ole Stauning at Nordea without picking up anything…
- I didn't start getting seriously interested before I saw presentations by Luca Capriotti (Credit Suisse) and Martin Baxter (Nomura).
- Generally, AD is picking up interest but real large scale applications are few.
- Most presentations that I have seen are either on particular subjects such as yield curve construction or "proof-of-concept".
- The first reason is that people haven't really understood it yet.

- The second reason it is really hard to do it for very complicated models. Or rather, to pull it through the full hierarchy of models that such models sit on top off.

- I recently spoke to a consultant on AD and he said that, by far, most of the work he was doing was for weather forecasting and the automotive industry.

Is AD the new Black?

- I think it might be, for several reasons.
- For the purpose of CVA it enables us to bring the risk reports to the front line and use these for structuring our clients' portfolios.
- It may also help on improving the quality of the regressions used for CVA and for options.
- But the real lift could be if it allows us to start thinking about using models that can only be solved numerically.
- The bottle neck is terms of model sophistication is usually always analytical tractability and/or quality of data for input for calibration.

- With AD and MT combined it seems natural to start thinking about models that are calibrated by Monte-Carlo.

- Further, there are various types of non-linear pricing problems where AD could be very interesting.

- For example, dynamic hedging under transaction costs. A problem that is normally left to trader intuition, rules of thumb, and wet fingers in the air.

- Transaction costs are proportional to local (cross) gamma. Of the total portfolio – not a single claim.

- AD doesn’t directly give us the local gamma but it gives us the local risk to variance/covariance and these two quantities are related.

- Another example is feedback and optimal liquidation of large stock positions.

Summary

- Turbo charging our CVA model has been a longer journey than we thought.
- The feeling through this has been a bit like climbing a mountain. You can see the peak in the distance but once you get there you realize there is another higher peak behind it.
- We have had to invent and re-invent a lot of trickery as we have gone along.
- That said, we feel that it has been worth it.
- Particularly, we think that AD could be a game changer in terms of practical modeling technology.