KOODERIVE: MULTI-CORE GRAPHICS CARDS, THE LIBOR MARKET MODEL, LEAST-SQUARES MONTE CARLO AND THE PRICING OF CANCELLABLE SWAPS

MARK S. JOSHI

ABSTRACT. We discuss the pricing of cancellable swaps using the displaced diffusion LIBOR market model using a multi-core graphics card. We demonstrate that over one hundred times speed up can be achieved in a realistic case.

1. INTRODUCTION

The advent of multi-core graphics cards places the possibility of a huge amount of computing power in the desk-top at a reasonably low cost. Such cards are known as graphics processing units or GPUs. However, they can be used for many purposes other than graphics. In particular, they are very naturally adapted to Monte Carlo simulation since their design is naturally highly parallel in nature. They work best when threads are performing the same operations with varying initial data.

To gain some idea of the level of computing power, consider NVIDIA's recent GPU, the Tesla K20. This has a peak performance of 3.52 teraflops, that is it can perform 3.52 trillion single precision floating point operations per second. However, possession of raw computing power is not enough. Different architectures require different coding designs and the question is therefore what level of performance can one achieve for realistic practical problems? For example, Aldrich, Fernandez-Villaverde, Gallant, and Rubio-Ramirez (2011) demonstrate the effectiveness of GPUs for solving dynamic equilibrium problems in economics using iterative methods.

The pricing of non-early exercisable derivatives using GPUs is straightforward and large speed ups can be obtained. The case of Asian options was studied by Joshi (2010). An early piece of work on pricing exotic interest derivatives using the LIBOR market model was Giles and Xiaoke (2008) where a 120 times speed up was achieved. Whilst this was an interesting pilot, the case studied was a little simple in that a one-factor model was used.

The questions we address here are

- Can a complex multi-factor displaced diffusion LIBOR market model be implemented in such a way as to achieve large speed ups whilst maintaining genericity?
- Is it possible to implement an effective and fast pricer for early exercisable Bermudan derivatives using the GPU?

We answer both questions in the affirmative and achieve over a hundred times speed-ups over single-threaded CPU C++ code. The second problem is much tougher than the first in that the

Date: June 2, 2014.

Key words and phrases. GPU, CUDA, LIBOR market model, Bermudan option, least-squares, Monte Carlo simulation.

pricing of early exercisable derivatives via Monte Carlo simulation is much more complex than pricing path-dependent derivatives lacking this feature. The fundamental reason is that an exercise strategy has to be developed and the development of the strategy requires interaction between paths. Thus one cannot set many threads going, let each one handle a different path and accumulate results at the end.

The problem of pricing Bermudan (or American) derivatives by Monte Carlo simulation is well known and used to be regarded as very hard. However, much progress has been made in recent years. We focus on lower bounds in this paper. We leave the problem of upper bounds on the GPU to future work. The use of regression to develop estimates of continuation values and therefore to decide exercise decisions has, in particular, proven popular. This was introduced by Carrière (1996) and popularized by Longstaff and Schwartz (2001). Whilst these methods work reasonably well for finding lower bounds in many cases, they are not always successful. Continued work has therefore focussed on enhancing these techniques. (Kolodko & Schoenmakers, 2006), (Broadie & Cao, 2008), (Beveridge, Joshi, & Tang, 2013). Whilst these improved techniques have proven effective, they are very much designed for the virtues and constraints of a CPU. For example, early termination of a sub-simulation if an inaccurate number is sufficient makes sense on the CPU, but running lots of sub-simulations with differing numbers of paths is unnatural on a GPU. We therefore introduce a new approach to early exercise based on using a cascade of multiple regressions. This builds on previous work (Broadie & Cao, 2008) on double regressions.

In this paper, we present and discuss solutions to the challenges of pricing early exercisable exotic interest derivatives on the GPU using NVIDIA's CUDA language. In particular, we discuss in detail the design choices made in the open source project Kooderive which contains an example of the pricing of a 40-rate cancellable swap using a five-factor displaced diffusion LIBOR market model. The code is fully available for download under the GNU public licence 3.0, (Joshi, 2014). The project is a collection of C++ and CUDA code targeting the Kepler 3.5 architecture and the K20c NVIDIA graphics card. It will however run on other cards with previous architectures. It is designed for use with Windows operating systems and comes with project files for Visual Studio which allows immediate building.

Whilst we focus on the LIBOR market model (LMM), we emphasize the approaches used are generic and they could equally well be applied to other models. In particular, the multi-dimensional Black–Scholes is from the point of implementation really just a simplification of the LMM in which drifts and discounting are much easier. The implementation of the early exercise code is done in generic fashion and does not use specific features of the model and product. In fact, a key part of the design is that the early exercise strategy is generated in a different component that only interacts with the path generator via the paths produced and so will be very broadly applicable.

We focus here on cancellable swaps because their pricing is mathematically equivalent to that of Bermudan swaptions and callable fixed rate bonds. These products are the most common exotic interest rate derivatives. However, no special features of the product are used and other products can be handled by modifying the functions defining the coupons with no changes elsewhere.

We note that there has been previous work on the use of GPUs for Bermudan/American options. Dang, Christara, and Jackson (2010) proceed using a PDE approach. Abbas-Turki and

Lapeyre (2009) use a least-squares Monte Carlo approach. However, the case they study is 4-dimensional and indicative rather than realistic so it is difficult to know how their techniques would translate to the high-dimensional interest rate case. Most other work appears to be focussed on binomial trees and/or the one-dimensional case.

We review the LIBOR market model in Section 2. We discuss its algorithmic implementation in a discretized setting in Section 3. We develop new ideas for early exercise in Section 4. We discuss the software and hardware used in Section 5. We outline the design of the code in Section 6. The intricacies of memory use are examined in Section 7. We study how to evolve the LIBOR market model on the GPU in Section 8. The specification of products is done in Section 9. We go into the implementation details of regression on the GPU in Section 10. Section 11 covers methdologies for data collection to prepare for least-squares. Pricing is in Section 12. We present timings and numerical results in Section 13 and we conclude in Section 14.

I thank Oh Kang Kwon for his assistance with coding a Brownian bridge and with skipping a Sobol generator. I am also grateful to Jacques Du Toit for his comments on an earlier version of this manuscript.

2. The LIBOR market model

In this section, we briefly review the displaced diffusion LIBOR market model. This is standard material. We refer the reader to standard texts such as Andersen and Piterbarg (2010), Brace (2007), Brigo and Mercurio (2006), and Joshi (2011) for more details.

Since it was given a firm theoretical base in the fundamental papers by Brace, Gatarek, and Musiela (1997), Musiela and Rutkowski (1997), and Jamshidian (1997), the LIBOR market model has become a very popular method for pricing interest rate derivatives. It is based on the idea of evolving the yield curve directly through a set of discrete market observable forward rates, rather than indirectly through use of a single non-observable quantity which is assumed to drive the yield curve.

Suppose we have a set of tenor dates, $0 = T_0 < T_1 < \ldots < T_{n+1}$, with corresponding forward rates $f_0, \ldots, f_n$. Let $\delta_j = T_{j+1} - T_j$, and let $P(t,T)$ denote the price at time t of a zero-coupon bond paying one at its maturity, T. Using no-arbitrage arguments,

$$f_j(t) = \frac{\frac{P(t,T_j)}{P(t,T_{j+1})} - 1}{\delta_j},$$

where $f_j(t)$ is said to reset at time T_j, after which point it is assumed that it does not change in value. We work solely in the *spot LIBOR measure*, which corresponds to using the discretely-compounded money market account as numeraire, because this has certain practical advantages; see Joshi (2003a). This numeraire is made up of an initial portfolio of one zero-coupon bond expiring at time T_1, with the proceeds received when each bond expires being reinvested in bonds expiring at the next tenor date, up until T_n. More formally, the value of the numeraire portfolio at time t will be,

$$N(t) = P\left(t, T_{\eta(t)}\right) \prod_{i=1}^{\eta(t)-1} (1 + \delta_i f_i(T_i)),$$

where $\eta(t)$ is the unique integer satisfying

$$T_{\eta(t)-1} \leq t < T_{\eta(t)},$$

and thus gives the index of the next forward rate to reset.

Under the displaced-diffusion LIBOR market model, the forward rates that make up the state variables of the model are assumed to be driven by the following process

$$df_i(t) = \mu_i(f,t)(f_i(t)+\alpha_i)dt + \sigma_i(t)(f_i(t)+\alpha_i)dW_i(t), \tag{2.1}$$

where the $\sigma_i(t)$'s are deterministic functions of time, the α_i's are constant displacement coefficients, the W_i's are standard Brownian motions under the spot LIBOR martingale measure, and the $\mu_i's$ are uniquely determined by no-arbitrage requirements. It is assumed that W_i and W_j have correlation $\rho_{i,j}$ and throughout $\{\mathcal{F}_t\}_{t\geq 0}$ will be used to denote the filtration generated by the driving Brownian motions. In addition, all expectations will be taken in the spot LIBOR probability measure. The requirement that the discounted price processes of the fundamental tradeable assets, that is the zero-coupon bonds associated to each tenor date, be martingales in the pricing measure, dictates that the drift term is uniquely given by

$$\mu_i(f,t) = \sum_{j=\eta(t)}^{i} \frac{(f_j(t)+\alpha_j)\delta_j}{1+f_j(t)\delta_j}\sigma_i(t)\sigma_j(t)\rho_{i,j};$$

see Brigo and Mercurio (2001).

Displaced-diffusion is used as a simple way to allow for the skews seen in implied caplet volatilities that have long persisted in interest rate markets; see Joshi (2003a). In particular, the use of displaced-diffusion allows for the wealth of results concerning calibrating and evolving rates in the standard LIBOR market model to be carried over with only minor changes. The model presented collapses to the standard LIBOR market model when $\alpha_i = 0$ for all values of i.

3. The LMM in discrete time

In a computer program, we discretize time into a number of steps. Each step is typically from one reset date to the next. Thus it is a time-discretized version of the LMM that is important when implementing. Our philosophy as expounded in Joshi (2011) is that calibration should be done post discretization. We thus have a finite strictly increasing sequence of reset and payment times for LIBOR rates T_j, and a similar sequence of evolution times as inputs to our calibration. We will take these to be equal in what follows for simplicity, although this is certainly not a necessity. We assume N time steps and an F-factor model.

The effective calibration of the LIBOR market model is the subject of many papers. We will not address it here but instead will assume that a CPU routine has already produced a calibration which we use as an input for our pricing routine. The output of our calibrator is the following:

- the initial value of each forward rate, $f_r(0)$;
- the displacement for each of these rates, α_r;
- the pseudo-square root, A_{j-1}, of the covariance matrix of the log forward rates for each time step from T_{j-1} to T_j.

Note that one could equivalently specify the covariance matrix, C_j, for the time step instead of the pseudo-square root. The pseudo-root uniquely determines the covariance matrix, of course, and it is the covariance matrix that determines the drifts. However, since we are working with reduced-factor models, a pseudo-square root is a more natural object. See Joshi (2011) for discussion of this approach to calibration. Plus when working with low-discrepancy numbers, it is generally believed that working with a spectral pseudo-square root can improve convergence, (Giles, Kuo, Sloan, & Waterhouse, 2008), (Jäckel, 2001), so it is convenient to specify this explicitly.

We use log-rates $x_r = \log(f_r + \alpha_r)$ and the rates f_r as convenient. We need to compute drifts; these are state-dependent so only the drift at the start of the first step is known in advance. We use a predictor-corrector algorithm so they have to be computed twice per step. The discretized drift of a log forward rate f_r across step j, is

$$\mu_{j,r} = -0.5C_{j,rr} + \sum_{l=0}^{r} C_{j,rl} \frac{(f_r(t) + \alpha_r)\delta_r}{1 + f_r(t)\delta_r}$$

where $t = T_{j-1}$ when predicting and T_j when correcting. Whilst this expression is correct, it is inefficient from a computational perspective and an algorithm giving the same numbers with lower computational order is presented in Joshi (2003b) and we use it here.

Our evolution algorithm for the rates on each path is therefore as follows.

(1) Draw uncorrelated NF standard normals for a quasi-random generator.
(2) Use a Brownian bridge to develop these into F Brownian motion paths.
(3) Take the successive difference of these paths to get a vector, Z_j, of F standard normals for each step j.
(4) Compute drifts $\mu_{j,r}$ for step j using $(f_r(T_{j-1}))$. (For the initial step, use the stored values instead.)
(5) Let
$$(\hat{x}_r(T_j)) = (x_r(T_{j-1})) + A_j Z_j + (\mu_{j,r}).$$
(6) Compute drifts $\hat{\mu}_{j,r}$ for step j using $e^{\hat{x}_r(T_j)} - \alpha_j$.
(7) For each r, let
$$x_r(T_j) = (\hat{x}_r(T_j)) + \frac{1}{2}(\hat{\mu}_{j,r} - \mu_{j,r}).$$
(8) Let $f_r(T_j) = \exp(x_r(T_j)) - \alpha_r$.
(9) Unless at end of path go back to 3.

Once the forward rate path has been developed, other ancillary quantities such as discount factors are easy to compute.

4. Multiple regression

In this section, we discuss a new algorithm for developing the exercise strategy. This algorithm is designed to be simple but requires a large number of paths, making it suited to GPU programming. We first recall the least-squares method for cancellable products. (Carrière, 1996), (Longstaff & Schwartz, 2001), (Amin, 2003). The method generally works in three phases. In the first phase, a set of paths is generated. In the second phase, regression coefficients are estimated, and in the third these are used to develop a lower-bound price. We refer the reader to Joshi (2011) for further discussion.

The main choice in the algorithm regards which basis functions are used to regress continuation values against, and this can have a great effect on results, (Brace, 2007) (Beveridge et al., 2013). Here we make the distinction between basis functions and *basis variables.* We make the former polynomials in the latter. The crucial point is that the functions are easily generated from the variables. A variable would typically be a stock price, a forward rate, a swap-rate or a discount bond. We do not investigate the question of basis function choice here but instead refer the reader to the extensive analysis in (Beveridge et al., 2013).

In the second phase, a backwards inductive algorithm is used. First, at the final exercise time, the remaining discounted cash-flows for each path are regressed against the basis function values for that point on that path. The regression coefficients then yield an estimate of the continuation value. This estimate is compared to the exercise value for the path and the value of the product at this final exercise time is set to the exercise value if it is greater, and to the discounted value of the remaining cash-flows if it is not.

We then step back to the previous exercise time. We discount the value at the succeeding exercise time and add on the discounted value of any cash-flows that occur in between on each path. We then regress again and repeat all the way back to zero.

Whilst in simple cases, the method works well, in complicated ones it can do unacceptably poorly or be highly basis function dependent. Many techniques have been developed for improving the method, e.g. Beveridge and Joshi (2008), Kolodko and Schoenmakers (2006), Broadie and Cao (2008), Beveridge et al. (2013). However, their methodologies tend to be better suited to CPU code. For example, the use of sub-simulations with varying numbers of paths appears difficult to code effectively on the GPU.

However, we have the advantage that using large numbers of paths is practical. We therefore adapt and extend an idea from Broadie and Cao (2008) and Beveridge et al. (2013). They suggested using double regression: perform a least-squares regression and then perform a second regression for paths for which the absolute difference between the estimated continuation value and the exercise value is below some threshold such as 3%. The idea is that the second regression yields greater accuracy in the area where it is most needed. However, the approach does implicitly require that the first regression be sufficiently accurate that the exercise boundary is within this truncated domain.

We adapt this approach by picking a fraction $\theta \in (0,1)$ and a regression depth d equal to say 5. We regress using least-squares, discard the fraction $1-\theta$ of paths farthest from the estimated boundary, and then repeat. We do this until we reach the depth d. If we initially have N_1 paths, we finish with

$$N_d = N_1 \theta^d$$

paths. Typically, we would only require the proportion discarded to be approximately correct rather than exact for efficiency. The value θ would be chosen to make N_d of the size required. For example, we might let

$$\theta = 0.1^{1/5}$$

when using 327680 paths so that the final regression has roughly 32768 paths. The advantage of this approach is that by only discarding a small fraction of paths that are very far from the money at each stage, we are less likely to be affected by a substantial misestimation of the boundary.

Many authors use second-order polynomials in forward rates and swap-rates as basis functions following Piterbarg (2004). We will take the first forward rate, the adjoining co-terminal swap-rate and the final discount factor as our basis variables. The basis functions are then quadratic polynomials in these with or without cross-terms.

5. Packages and hardware

In this section, we briefly describe the software and hardware used. The principal software tool is the CUDA 5.5 toolkit available from NVIDIA for free download. This is used in conjunction with Visual Studio Professional Edition 2012 as IDE and as a C++ complier. The Thrust open source library for developing algorithms on the GPU is used extensively both for algorithms and memory allocation. The Thrust library now ships as part of the CUDA toolkit.

The CUDA code discussed in this paper is all part of the Kooderive open source library. Project files for Visual Studio are included and allow immediate building after download. This has a number of components. In particular, "gold" projects are written in C++ and run on the CPU. The main static library project is "kooderive" and the examples discussed here are in the project "kooexample."

The hardware used as a GPU is a single K20c Tesla card[1]. Its programming is purely done using the CUDA language. The compute capability of the device is 3.5 and the code assumes that such a GPU is available. The CUDA code is written in 64 bits.

The CPU used is an Intel(R) Xeon (R) CPU E5-2643 at 3.30 GHz. We use routines from the QuantLib open source library as a comparison. This is compiled using Visual C++ in 32 bit mode since that is the configuration specified by the current release.

6. Design overview

The pricing of a Bermudan contract by Monte Carlo can be divided into three phases.

(1) A number of paths, N_1, is generated and the relevant aspects of these paths for developing an exercise strategy are stored.
(2) A backwards induction is performed, generating regression coefficients and updating continuation values at each exercise date.
(3) A second Monte Carlo simulation is run using the exercise strategy generated in the second phase using N_2 paths. This is equivalent to pricing a path-dependent derivative with no optionality since the exercise strategy has been fixed.

It is the first phase that requires most care. The reason being that the data generated has to be stored until the end of the second phase. Thus we will require memory proportional to N_1. For our design, we keep all this data on the GPU at all times and so there must be sufficient memory to store it. If the GPU's total global memory is M and we store m bytes per path, we have the immediate constraint

$$N_1 m < M.$$

A Tesla K20c has 4800 megabytes of global memory so if we run 327680 paths, the maximum storage per path is 15360 bytes. In practice, since other data must be stored the maximum would be lower. A float takes 4 bytes so we have storage for less than 3840 floats per path.

[1]We thank NVIDIA for providing this hardware.

If our forward rate evolution has N rates and n steps, to store the entire evolution for a path will take Nn rates. If we take $N = n$, we will run out of memory for some $N < 62$. Whilst one could squeeze some more memory by discarding already reset rates, it would complicate accesses and there would be very little left for other computations. In practice, one will often want to develop many pieces of auxiliary data for all paths simultaneously, such as all the implied discount factors, which multiplies the memory requirements.

We therefore adopt an approach based on batching. Thus rather than storing everything about 327680 paths, we divide into say 10 batches and only store the aspects of the paths that are required for the backwards induction. So what must be stored? First, we note that we only need data at exercise times. We use "exercise step" to mean the step from one exercise date to the next. This may or may not be the same as the step from one reset date to the next.

- The sum of the discounted values of any cash-flows generated by the product during the exercise step. This yields 1 float per exercise step.
- The value of the numeraire at the start of the exercise step again yields 1 float.
- The discounted value of any rebate generated on exercise gives 1 more float.
- The basis variables for the exercise time is an input according to choices of basis functions but typically 3 is enough.

Here "discounted" means discounted to the start of the exercise step. We therefore typically have 6 data points per exercise step per path. If we have 40 exercise dates, and 327680 paths, this requires 300 MB, leaving plenty of space for more dates, paths or other needs.

Our second step processes the data from the first step and outputs regression coefficients. An interesting feature of the algorithm is that the second step uses no specific features of the model or product other than these outputs. This means that it is flexible and generic. Although developed for cancellable swaps in the LIBOR market model, it could equally well be used for Bermudan max-options in a multi-dimensional Black–Scholes model or, indeed, for any Bermudan derivatives pricing problem priced using martingale techniques.

For the third phase, we templatize the cash-flow generation on the exercise strategy using the coefficients produced during the second phase. This means that the exercise strategy is simply an input that could be changed drastically without changing this phase's design. Thus one could use a parametric strategy instead of a least-squares one whilst only making changes to the second step. Note that the third phase is again batched, and we only need to store a single number from the output of each batch: the mean value of the product. In consequence, there are no memory constraints on how many batches we use for this phase. Thus our only constraint on the size of N_2 is time.

In all phases, we divide the algorithm into a sequence of steps. Each step is performed on all paths in a batch (all paths in the second phase) by a single GPU routine. Thus almost everything is done for all these paths before any of them are complete. This is quite different from a typical CPU program such as QuantLib where the first path is complete before anything is done for the second path. Each of these small steps in Kooderive will generally correspond to a single call to the GPU.

7. Memory use, threads and blocks

A CUDA program consists of a C++ or C program together with various calls to *kernels* which run on the GPU. Each kernel is configured as a number of blocks (e.g. 64) and each block

has the same number of threads (e.g. 512). Threads in a block are divided into groups of 16 or 32 called warps. Threads in the same warp are constrained to perform the same instructions, and if the program requires them to do otherwise, idling occurs as the branches are evaluated serially.

A significant difference between GPU programming and CPU programming is the importance of how memory is used. A GPU program has to explicitly use many different sorts of memory and how this is done can have drastic effects on the speed of a program. The principal sorts of memory are (with the amount on a K20c in parenthesis)

- host – the computer's ordinary memory that the CPU uses;
- global – the graphics' card's main memory, (4800 MB), large and plentiful but must be accessed correctly or slowness occurs;
- shared – a small amount of memory shared between threads in the same block (49352 bytes), very fast but amount is very limited;
- constant – read only for the GPU but writable by the CPU (65536 bytes), again fast but use is very constrained;
- textures – a way of placing global memory in a read-only cache, fast for read without constraints, but the memory cannot be changed within a kernel.

Memory transfer between host and global memory is typically slow for large data sets and is often the main bottleneck in GPU programs. Kooderive avoids this issue by simply not using host memory after the set-up phase. Thus the model calibration and product specifications are passed to the GPU initially but the only data passed back thereafter is the mean values for paths.

The layout of how data is stored in global memory greatly affects speed. This is a consequence of the fact that threads do not access global memory independently. Each thread has a thread number, t, and a block number, b. We will call the total of number of threads the width, w. The number of threads per block we denote s for size. If the code is written so that in a warp, thread t accesses location $l+t$ for some l, then the access is *coalesced* and occurs quickly. However, if the mapping is more complicated and each thread accesses $f(t)$ for some non-trivial function f such as $f(t) = l + \alpha t$ for some $\alpha > 1$, memory access is slow. A common approach throughout Kooderive is that thread t in block b is responsible for the path

$$t + bs + kw,$$

for all $k \in \mathbb{N}$ such $t + bs + kw$ is less than the total number of data points (typically the paths in a batch.) Data is then stored with the path in smallest dimension, the time step in the largest dimension, and any other index in the middle. Thus if are R rates, N steps and P paths, forward rate r on time step s for path p would be stored at location

$$p + rP + sRP.$$

With this lay-out, coalescing occurs naturally. This is in constrast to a typical CPU program where all the data for each path would be stored together, and one might use location

$$r + sR + pRN.$$

Coalescing is more important for writes than reads, since textures provide a fast route to memory access provided there are no writes to that part of memory during the kernel. Thus for pieces of data that do not change during the kernel, textures are widely used in Kooderive

to speed up memory access and to avoid coalescing constraints. The textures provide a cache that also speeds up access. Note that this cache can be explicitly accessed using the $\_\_ldg()$ function in the Kepler architecture and this is also sometimes done. An alternate approach, used in Kooderive, is to copy constant data into shared memory at the start of a kernel. However, this relies on the amount of data being small and we therefore use this technique only a little. Whilst constant memory provides an additional alternative, the advantages do not seem sufficient to justify its unwieldiness and it is not used in Kooderive.

Shared memory is also useful as a fast workspace and this is done by the main path generation kernel.

8. Path generation

For both the first and third phases, we want to develop large numbers of LMM paths rapidly. We develop the paths in batches of say 32768 in size. The batch creation is divided into a number of kernels.

- Generation of Sobol numbers as integers.
- Scrambling.
- Conversion to normals.
- Brownian bridging.
- Path generation.

The generation of Sobol numbers is done using a modification of the example in the CUDA SDK. The main differences being a skip method has been added to allow the Sobol sequence to start at an arbitrary point, and the return of unsigned integers rather than floats or doubles. The first facilitates batching, each batch simply skips to the end of the previous batch. The passing back of unsigned integers is to allow scrambling. Here we input a fixed vector of unsigned integers for the batch and apply exclusive-or to each vector of Sobol draws. If the scrambling vector is drawn randomly, we can view the batch average of quantities as an unbiased estimate of any expectations. This is similar to randomized-QMC. See Giles et al. (2008).

To transform to normals, we use the Shaw–Brickman algorithm. (Shaw & Brickman, 2009) This is performed using the "transform" algorithm from the Thrust library. We use one call to do both the conversion to uniforms from unsigned ints and to take the inverse cumulative normal

The Brownian bridge is performed using a two-dimensional grid of blocks. This reflects that the Sobol paths will be of dimension NF with N the number of steps and F the number of factors. We effectively have to create F paths of N steps from these for each overall path. The x-coordinate of the block and the Thread Id determine which overall path. The y-coordinate determines the factor. For each block, we first copy all the auxiliary data needed for the bridge into shared memory. Each thread then develops the bridge for one factor for one path without further interaction with other threads. If the total number of threads across all blocks is less than the total number of paths, then a thread will do multiple paths each separated by the global width. The generated paths are successively differenced at the end so that the outputs would be independent standard $N(0,1)$ random variables if the inputs were. We refer the reader who is interested in high performing Brownian bridges to du Toit (2011) for discussion of an alternative approach.

The main kernel that does the forward rate evolution and computes the implied discount factors is

$$\text{LMM\_evolver\_all\_steps\_pc\_kernel\_devicelog\_discounts}\,.$$

This implements the Hunter, Jäckel, and Joshi (2001) algorithm for predictor-corrector evolutions. A large fraction (roughly 70%) of the total compute time for phases one and three is spent in this kernel, and so its efficiency is very important. We therefore discuss it in detail. Its inputs are

- the pseudo-square roots of the covariance matrices (texture),
- the accruals of the forward rates (texture),
- the displacements (texture),
- the fixed part of the drifts (texture),
- the value of the state-dependent drifts for the first step (texture),
- indices that denote which rates are not yet reset for each step (texture),
- the initial logs of the rates, (texture)
- the quasi-random variates, (texture)
- the numbers of paths, rates and steps (integers).

So all data is passed in as either a texture or an integer. It outputs the full forward rate evolutions, the log displaced rates and the implied discount factors which are all stored in global memory.

Each path is handled by a single thread. If there are more paths than threads, a thread will handle multiple paths with index separated by the total width in a serial manner.

The steps are done one at a time. (The first step is handled differently since the drifts are already known.) The use of the fast drift computation algorithm from Joshi (2003b) requires floats equal to the number of factors to store the partially computed sums. We therefore use factors times block-size floats in shared memory for each thread as storage. Given these facts, the evolution for each path is then straight-forward and the coding of the algorithm is little different from that of a C program.

For step 0, we multiply the variates by the pseudo-root and add them to the log rates. We add on the drifts. We then compute the state-dependent drifts at the end of the step. We correct the values of the log rates and the rates. We store their values in the output data and we use this location to retrieve them when needed during this kernel. We then use the rates to compute the discount factors implied by these forward rates for the step. For the other steps, we first compute the drifts at the start of the step, and then do as for step zero.

Whilst the code is robust against variation in block and grid size, the combination of 128 threads per block, and 256 blocks proved effective when using 32768 paths per batch. Note that this implies that each thread does precisely one path. A slight further optimization could be obtained by rewriting the code not to handle other cases, but the gains do not seem sufficient for this to be worthwhile.

9. Product specification and design

It not enough just to be able to price a single product using handcrafted code. One wants a code design that allows flexibility and changes. One approach advocated in Joshi (2008) is to use an object encapsulating the product's termsheet with virtual functions providing the necessary data. However, this does not seem well adapted to working on the GPU with CUDA. We therefore decompose the product into a number of components which can be

written independently and then slotted together. First, we regard the product as generating a pair of cash-flows at each of a set of evolution times. For each generation, the product is passed three rates as well as the current discount curve and forward rates. The computation or extraction of the three distinguished rates is performed independently and so the product knows nothing of their origin or meaning. Thus they could be swap-rates or forward-rates or something more complicated. In addition, if one wished to incorporate OIS discounting then a distinction between LIBOR rates and OIS rates could be made at this point by adding a spread.

Second, the product simply generates two flows but does not specify their timing. Instead, separate payment schedules are specified and these are passed to a discounting routine. This allows changes, for example, from in-advance to in-arrears with minimal changes to the code simply by changing these timings.

The actual cash-flow generation routine which turns the rates into the cash-flow sizes is done using a templatized kernel. The template parameter is the product. It steps through the evolution times passing the rates and discount curves to the product and storing the cash-flows as they are generated. The path is terminated when the product indicates to do so. The design is set-up so that the product is able to store auxiliary data such a running coupon if necessary which allows the possibility of path-dependence. Alternatively, one of the rates passed in could be made path-dependent.

Exercise values are generated independently of the product, again allowing maximum flexibility.

10. Least-squares and Multiple Regressions on the GPU

The second phase of the least-squares algorithm is to perform regressions on estimated continuation values. Here we enact multiple regressions for each step. The least-squares algorithm works backwards in time. On each step, we first have to find the values of the basis functions. Note here the distinction between basis functions and basis variables. The latter are extracted in phase 1 and would typically be a forward rate, x, the adjoining swap rate, y, and the final discount factor z. The former are polynomials in these. We focus on quadratic polynomials. We can work with or without cross terms, so we can use

$$1, x, y, z, xy, yz, zx, x^2, y^2, z^2$$

and have ten basis functions, or use

$$1, x, y, z, x^2, y^2, z^2,$$

and have seven. The code is templatized on the algorithm for turning variables into functions to allow flexibility.

Thus at the start of each step, we first use a kernel to generate the basis functions for the step. Note that we only ever store the full basis functions for one step at a time to reduce memory usage.

Once the basis functions are known, we have to the find minimal least-squares error solution of a highly over-determined rectangular system

$$Ax = y,$$

where A has N_1 rows. Each row consists of the values of the basis functions for one path for the step. The target y is the discounted future cash-flows for the path. Typically, $N_1 = 327680$

and there are ten basis functions so the system is very overdetermined. We solve in two phases. First, we write

$$(A^tA)x = A^ty,$$

reducing to a 10 by 10 (or similar) system and then we solve this system.

For the computation of A^tA, we use two kernels. The first kernel computes for each j and k with $j \leq k$

$$\sum a_{ij}a_{ik}$$

with the sum taken over a subset of i. The subsets for different blocks partition the paths, and we use 1024 blocks. So at the end, we have 1024 numbers for each matrix entry still to be summed. These sums are performed by a second kernel which uses a different block for each entry. The computation is then done by copying the data into shared memory and then using a repeated binary summation so that each thread in the first half adds the value for the corresponding thread in the second half. Once a thread reaches the second half it does nothing further. Eventually only thread zero remains and it contains the value of the sum. Note that this approach minimizes the length of the computation chain which is an important consideration when working with floats to avoid round off error. The computation of A^ty is done similarly.

For the second part of solving the small system, we copy the problem to the CPU and solve there. The fact that it is only a 10 by 10 system means that this is fast. The solution of the small system is the vector of regression coefficients.

However, the step is not yet done since we are doing multiple regressions. The regression coefficients yield an implied continuation value for every path. Our multiple regression algorithm requires us to discard the fraction $1 - \theta$ of paths which are furthest from the exercise boundary, that is the ones that yield the largest absolute value for discounted continuation value minus discounted exercise value. First, a cut-off level is found which gives the threshold above which paths are discarded. This is done by repeated bisection with the counting being done using the thrust $\mathrm{transform}$ and count algorithms. Second, the data for the remaining paths are moved to be contiguous in memory. This is performed using thrust's $\mathrm{scatter\_if}$ algorithm. The process is repeated on the remaining data until the pre-set regression depth is reached (e.g. 5) or too few paths remaining according to a pre-set cut-off such as 2048. For subsequent estimates of continuation values, cascading through the coefficients is performed until a threshold level is reached or the maximum depth is obtained.

Once the continuation values have been estimated, the next stage of the algorithm is to set the paths' step-wise values to be either the discounted future flows for the path if exercise does not occur, or to the discounted exercise value if it does. This is straightforwardly performed by a simple kernel. The final action for the step is to deflate to the previous exercise time using the ratio of the numeraire values at the two times. This is again straight-forward.

We then simply repeat back to step 0. The average value after doing step 0 yields the first pass estimate of the discounted cash-flows on or after the first exercise time.

11. The data collection phase

As discussed above, the first phase of the pricing is to collect data for the regression and backward induction. The generation is divided into a number of batches (e.g. 10) and in each one the important data is stored. Each batch is straight-forward. The following operations are carried out.

- The paths are generated as in Section 8.
- The co-terminal swap-rates and their annuities are computed that is the swap-rates with the same final date but the first date varying. (c.f. Jamshidian (1997).)
- The just reset forward rate at each step is extracted as is the adjoining co-terminal swap-rate.
- The final discount factor for each step is also extracted.
- The basis variables are computed from the three previous extracted values and stored.
- The rates underlying the product are also extracted.
- The numeraires along the paths are computed.
- The cash-flows along the paths are generated and discounted to exercise dates. They are then aggregated to each exercise date and stored.
- The exercise values are discounted and stored.
- The numeraire values at exercise dates are stored.

Each of these is done by a dedicated kernel. We have already discussed the cash-flow generation. The other kernels are straight-forward.

12. The pricing phase

The third and final phase is the actual pricing. At this point, the exercise strategy has been generated during the second phase and so we are pricing conditional on a set of regression coefficients. Most of the phase is very similar to the first one. The main differences being that there is no need to store data for use after the batch has been processed, and that the cash-flow generation takes the exercise strategy into account.

- The paths are generated as in Section 8.
- The co-terminal swap-rates and their annuities are computed that is the swap-rates with the same final date but the first date varying. (c.f. (Jamshidian, 1997).)
- The just reset forward rate at each step is extracted as is the adjoining co-terminal swap-rate.
- The final discount factor for each step is also extracted.
- The basis variables are computed from the three previous extracted values and stored.
- The rates underlying the product are also extracted.
- The numeraires along the paths are computed.
- The cash-flows along the paths are generated up to the exercise time. If exercise occurs, the exercise value is taken into account.
- The cash-flows are deflated.
- The cash-flows are summed for each path.
- The pathwise values are averaged using the reduce algorithm from thrust.

For each of these a simple dedicated kernel is used. The only one of much interest is the cash-flow generation kernel. For maximum flexibility, this is templatized on three parameters: the product, the exercise value computer, and the exercise strategy. The auxiliary data for the strategy is moved into shared memory for rapid access. Once this has been done, the routine is very similar to that for the cash-flow generation in the first phase.

Note that each batch produces one number which is the mean pay-off value. Different batches can be achieved either by making the quasi-random generator skip or by using scrambling.

13. Speed comparisons and numerical results

We focus on the case of a 40-rate cancellable swap studied in Beveridge et al. (2013). We choose this as it is challenging enough to be interesting without being esoteric. We also want to study a tough example already in the literature to demonstrate the non-artificiality of our results. The product has 40 underlying rates. The first one starts in 0.5 years. Coupon payments occur at $0.5 + 0.5j$ for $j = 1, 2, \ldots, 40$. It is cancellable at time 1.5 and every 0.5 years thereafter. No rebate is paid on cancellation. The swap pays a fixed rate of 0.04 and receive floating.

The calibration is that we set the forward rate from $0.5j$ to $0.5(j + 1)$ to be

$$0.008 + 0.002j \quad \text{for} \quad j = 0, 1, 2, \ldots, 40.$$

The well-used "abcd" time-dependent volatility structure is used, with

$$\sigma_i(t) = \begin{cases} 0, & t > T_i; \\ (0.05 + 0.09(T_i - t)) \exp\left(-0.44(T_i - t)\right) + 0.2, & \text{otherwise}, \end{cases}$$

and the instantaneous correlation between the driving Brownian motions is assumed to be of the form

$$\rho_{i,j} = \exp\left(-\phi|t_i - t_j|\right),$$

with $\phi = 2 \times 0.0669$. Displacements for all forward rates are assumed to be equal, with

$$\alpha_j = 1.5\%,$$

for all values of j.

We use a five-factor model. To obtain the reduced pseudo-square root matrices, we proceed as follows.

- Compute the full-factor covariance matrix for the step. This involves integrating the product of the volatility functions and multiplying by the instantaneous correlation for each entry.
- Perform a principal components analysis to obtain eigenvalues and eigenvectors, λ_j and e_j with λ_j decreasing.
- Form a column matrix $B = (\sqrt{\lambda_j} e_j)$.
- Scale the rows of B so that the variances of the log-rates is the same as before factor reduction.

Beveridge et al. (2013) achieve a price of 1088 with a standard error of 2.5 using double regression and the exclusion of sub-optimal points. They use policy iteration to get an increase of 6.5 with a standard error of 2. Thus their lower bound price is 1094.5 with a standard error of 3.2. Their upper bound has the slightly lower value of 1094 with a standard error of 3.

We present results on timings and price for varying regression depths in Table 1. We use 10 batches of 32768 paths for the first pass and 32 of them for the second. We separate the time actually spent doing regressions from that spent on doing on other parts of the exercise strategy building. The price increases substantially when we increase from single regression to double regression. Another slight increase occurs from double to triple and it is stable thereafter. The fraction of paths retained after each regression is given by $0.1^{1/d}$ where d is the total regression depth. The implied price whilst slightly lower than that in Beveridge et al. (2013) is within one standard error and so can be regarded as accurate.

regression depth	1	2	3	4	5
time taken for first pass paths	0.207	0.206	0.207	0.206	0.207
time taken for regression set up	0.206	0.169	0.172	0.179	0.179
time taken for regression	0.094	0.191	0.254	0.32	0.39
time taken for second pass	0.697	0.71	0.729	0.747	0.765
total time	1.204	1.276	1.362	1.452	1.541
second pass price	0.10740	0.10911	0.10924	0.10927	0.10925

TABLE 1. Timings and prices for the 40-rate cancellable swap with varying numbers of regressions. All standard errors are between 0.5 and 0.6 basis points.

	time QL	time Kooderive	ratio
first pass	34.958	0.195	179.2718
strategy building	11.037	0.379	29.12137
second pass	122.013	0.648	188.2917
total	168.008	1.222	137.4861

TABLE 2. Timing comparison for QuantLib versus Kooderive for a 38 rate cancellable swap with 38 call dates. Times are seconds. 327680 first pass paths, 1048576 second pass paths. Single regression.

In Table 2, we present timing comparisons for Kooderive versus QuantLib. For simplicity in running the QuantLib (QL) code, we do not consider the first two non-call coupons. The value of these is analytic in any case and a simulation pricing of them is not needed. We see that the first pass of path storage is 179 times faster in Kooderive. Similarly, the second pass pricing is 188 times faster. Note that is despite the fact that QuantLib early terminates path generation when appropriate and Kooderive does not. The timing ratio for the computation of regression coefficients is not quite so impressive as *only* a 29 times speed-up is achieved. Note, however, that the division between stages in Kooderive is slightly different from QuantLib. The generation of basis functions from basis variables is done in the first pass in QuantLib and during strategy building in Kooderive. This means that our numbers overstate the speed of the first pass and the slowness of strategy building. The overall ratio, which is what really matters, at 137 is very large. We can run numbers of paths in seconds that would previously been regarded as silly in a live environment.

Of course, a CPU implementation could also be multi-threaded and the first and third parts should scale well since they are embarrassingly parallel. For the second part, one would have to solve similar challenges to that presented for the GPU. We do not explore how to carry out such an implementation here. However, we note that if the CPU used t threads, the best we could hope for is a t-times speed up, and so we would still expect the GPU to $137/t$ times faster. It would take a very large number of CPU cores for the CPU to be competitive against a single GPU.

14. CONCLUSION

We have demonstrated that it is possible to develop a full-featured powerful and flexible displaced diffusion LIBOR market model than runs on the GPU. The speed-up obtained is over

100 times and this is using a single GPU. It is now possible to routinely run numbers of paths that would have been regarded as far too large in the past.

We have not explored variance reduction methods nor Greeks in this paper. However, we believe that there are no particular barriers to implementing them and thus to achieving even faster pricing. Another natural extension of the work here is to consider a more complex model incorporating OIS discounting and smiles. We leave these to the future.

References

Abbas-Turki, L., & Lapeyre, B. (2009). American options pricing on multi-core graphic cards. *2009 International Conference on Business Intelligence and Financial Engineering*, 307.

Aldrich, E. M., Fernandez-Villaverde, J., Gallant, A. R., & Rubio-Ramirez, J. F. (2011). Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors. *Journal of Economic Dynamics and Control*, *35*(3), 386 - 393. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0165188910002216` doi: http://dx.doi.org/10.1016/j.jedc.2010.10.001

Amin, A. (2003). *Multi-factor cross currency LIBOR market model: implementation, calibration and examples.* (http://www.geocities.com/anan2999/)

Andersen, L., & Piterbarg, V. V. (2010). *Interest rate modelling*. London, New York: Atlantic Financial Press.

Beveridge, C. J., & Joshi, M. S. (2008). Juggling snowballs. *Risk*, *December*, 100-104.

Beveridge, C. J., Joshi, M. S., & Tang, R. (2013). Practical policy iteration: generic methods for obtaining rapid and tight bounds for bermudan exotic derivatives using monte carlo simulation. *Journal of Economic Dynamics and Control*, *37*, 1342-1361.

Brace, A. (2007). *Engineering BGM*. Sydney: Chapman and Hall.

Brace, A., Gatarek, D., & Musiela, M. (1997). The market model of interest rate dynamics. *Mathematical Finance*, *7*, 127-155.

Brigo, D., & Mercurio, F. (2001). *Interest rate models: Theory and practice*. Heidelberg: Springer Verlag.

Brigo, D., & Mercurio, F. (2006). *Interest rate models-theory and practice: with smile, inflation and credit*. Springer.

Broadie, M., & Cao, M. (2008). Improved lower and upper bound algorithms for pricing American options by simulation. *Quantitative Finance*, *8*, 845-861.

Carrière, J. F. (1996). Valuation of the early-exercise price for options using simulation and nonparametric regression. *Insurance: Mathematics and Economics*, *19*, 19-30.

Dang, D. M., Christara, C. C., & Jackson, K. R. (2010). Pricing multi-asset american options on graphics processing units using a pde approach. In *High performance computational finance (whpcf), 2010 ieee workshop on* (pp. 1–8).

du Toit, J. (2011). *A high-performance brownian bridge for gpus: lessons for bandwidth bound applications.*

Giles, M., Kuo, F. Y., Sloan, I. H., & Waterhouse, B. J. (2008). Quasi-monte carlo for finance applications. *ANZIAM Journal*, *50*, C308–C323.

Giles, M., & Xiaoke, S. (2008). *Notes on using the nVidia 8800 GTX graphics card.* (http://people.maths.ox.ac.uk/ gilesm/hpc/NVIDIA/libor/report.pdf)

Hunter, C., Jäckel, P., & Joshi, M. S. (2001). Getting the drift. *Risk*, *July*, 81-84.

Jäckel, P. (2001). *Monte carlo methods in finance*. New York: John Wiley and Sons Ltd.

Jamshidian, F. (1997). LIBOR and swap market models and measures. *Finance and Stochastics*, *1*, 293-330.

Joshi, M. S. (2003a). *The concepts and practice of mathematical finance*. London: Cambridge University Press.

Joshi, M. S. (2003b). Rapid drift computations in the LIBOR market model. *Wilmott Magazine*, *May*, 84-85.

Joshi, M. S. (2008). *C++ design patterns and derivatives pricing (2nd edition)*. London: Cambridge University Press.

Joshi, M. S. (2010). Graphical asian options. *Wilmott Journal*, *2*, 97-107.

Joshi, M. S. (2011). *More mathematical finance*. Melbourne: Pilot Whale Press.

Joshi, M. S. (2014). *Kooderive version 0.3*. Retrieved from `http://kooderive.sourceforge.net`

Kolodko, A., & Schoenmakers, J. (2006). Iterative construction of the optimal Bermudan stopping time. *Finance and Stochastics*, *10*, 27-49.

Longstaff, F. A., & Schwartz, E. S. (2001). Valuing American options by simulation: a simple least squares approach. *The Review of Financial Studies*, *14*, 113-147.

Musiela, M., & Rutkowski, M. (1997). Continuous-time term structure models: forward-measure approach. *Finance and Stochastics*, *1*, 261-292.

Piterbarg, V. (2004). A practitioner's guide to pricing and hedging callable LIBOR exotics in forward LIBOR models. *Journal of Computational Finance*, *8*, 65-119.

Shaw, W. T., & Brickman, N. (2009). *Differential equations for monte carlo recycling and a gpu-optimized normal quantile* (Tech. Rep.). Citeseer.

Centre for Actuarial Studies, Department of Economics, University of Melbourne, Victoria 3010, Australia

E-mail address: `mark@markjoshi.com`