2014

gpusvcalibration: A R Package for Fast Stochastic Volatility Model Calibration Using GPUs

Matthew Dixon
University of San Francisco, mfdixon@usfca.edu

Sabbir Ahmed Khan

Mohammad Zubair

gpusvcalibration: A R Package for Fast Stochastic Volatility Model Calibration using GPUs

Matthew Dixon[1], Sabbir Ahmed Khan[2], and Mohammad Zubair[2]

[1]Department of Analytics, School of Management, University of San Francisco, San Francisco, CA 94117.
[2]Department of Computer Science, Old Dominion University, Norfolk, VA 23529.

Keywords: Software, R, GPGPU Computing, Stochastic Volatility, Calibration

Abstract

In this paper we describe the `gpusvcalibration` R package for accelerating stochastic volatility model calibration on GPUs. The package is designed for use with existing CRAN packages for optimization such as `DEOptim` and `nloptr`. Stochastic volatility models are used extensively across the capital markets for pricing and risk management of exchange traded financial options. However, there are many challenges to calibration, including comparative assessment of the robustness of different models and optimization routines. For example, we observe that when fitted to sub-minute level mid-market quotes, models require frequent calibration every few minutes and the quality of the fit is routine sensitive.

The *R* statistical software environment is popular with quantitative analysts in the financial industry partly because it facilitates application design space exploration. However, a typical R based implementation of a stochastic volatility model calibration on a CPU does not meet the performance requirements for sub-minute level trading, i.e. mid to high frequency trading. We identified the most computationally intensive part of the calibration process in R and off-loaded that to the GPU. We created a map-reduce interface to the computationally intensive kernel so that it can be easily integrated in a variety of R based calibration codes using our package. We demonstrate that the new R based implementation using our package is comparable in performance to a $C/C++$ GPU based calibration code.

1. INTRODUCTION

Parallel computing in R The *R* statistical software package is an easy to use modeling environment and programming language. It is becoming increasingly popular with the financial industry, especially for advanced financial modeling and analytics which requires a significant amount of model design space exploration. The R environment is single threaded. To overcome this limitation, a number of shared and distributed parallel programming libraries already exist for R [14]. Most notably, `snowfall` [10] provides a thread safe abstraction layer by hiding the communications details and operates over MPI, PVM or sockets. The `parallel` package provides a way of running parallel computations in R on machines with multiple cores or CPUs. There has been recent effort in providing R packages that take advantage of GPGPU computations in the R environment [4]. However, most of these packages support a limited set of basic functions. There is a need for GPU optimized functions at a higher level that can be easily integrated in a programming environment such as R.

Stochastic Volatility modeling In this paper, we focus on the problem of how to effectively calibrate stochastic volatility option pricing models to exchange listed option prices, which is a topic of great interest to practitioners who require pricing models for building volatility surfaces to fit the market. A primary reason for this is that the volatility surface is used to price and measure the risk of more exotic options which may be traded over the counter. Of the range of stochastic volatility models favored by practitioners, the Heston stochastic volatility [8] has drawn the most widespread usage for its ability to capture the volatility smile and skew due to, for example, leverage effects. But option markets may move precipitiously and we've observed a need to frequently recalibrate the model which in turn creates the need for high performance financial computations in a statistical modeling environment.

Implementation gap The availability of the Heston model, or any option pricing model, within an R environment has several advantages. Most prominently R users can leverage the extensive set of R libraries to calibrate and easily back-test the model against historical prices to assess the 'best' option pricing model. The authors' experience has been that this is not only useful for research and development but also for testing and diagnosing production grade applications. However, there is a performance penalty in implementing financially intensive computations in the R programming language compared to C/C++. We demonstrate later in this paper the performance gain from simply reimplementing the Heston model in C/C++ and making it available in R through a wrapper function. Migrating stable code from R into an efficiency language also mitigates the implementation gap between prototype R applications and their productionized counterparts by providing a shared library for both prototypes and production grade applications to use.

Calibration The calibration of a stochastic volatility model is performed over M option data points (referred to as a "chain") which remains fixed during the calibration computation. The calibration algorithm starts with an initial guess of the model parameters and iteratively improves the guess using an optimization algorithm until it meets the convergence criteria. A typical organization of this computation involves calling an optimization routine with a pointer to *ErrorFunction()*, which estimates the error between market observed option prices and prices calculated using the stochastic volatility model for the current guess of the parameter set.

Using GPUs for calibration To improve the performance of this calibration approach, Aichinger, Binder, Fürst and Kletzmayr [1] implement a shared memory parallelization of the Heston model calibration routine on a multicore CPU SGI Altix 4700 and a GPU server with two C1060 cards and a GTX260 card. The authors compare the stability and performance of various off-the-shelve global optimizers before concluding that the best performance can be obtained by using a hybrid composed of one of a variety of global optimizers with a Levenberg-Marquardt unconstrained local optimizer. The global optimizers that the authors consider include the differential evolution (DE) algorithm and simulated annealing (SA), both of which have been employed elsewhere in the quantitative finance literature [2]. Dixon and Zubair [6] consider the calibration of a Bates model, a slightly more generalized form of the Heston model which includes jumps, using python and compare the performance tradeoffs of using the `mpi4py` and `multicore` python packages to parallelize computations on a multi-core CPU cluster. Here, in this paper, we depart from both of these works by presenting a R package for off-loading a variety of stochastic volatility model computations onto the GPU. The performance is evaluated for a range of single-name equity option chains traded on the NYSE.

Parallel design The package is designed to hide the parallelism from the R user and be used in a variety of calibration and other optimization functions. We employ the map-reduce parallelization design pattern to accelerate the computationally intensive component of the calibration process. This approach is conceptually similar to the *Split-Apply-Combine Strategy for Data Analysis* [16], although the calibration is compute intensive rather than data intensive.

Performance benchmarks We demonstrate the use of package in the calibration of a Heston model and obtain a factor of up to $760x$ improvement over the R sequential implementation by off-loading the *ErrorFunction()* on a system with Intel Core i5 processor and NVIDIA Tesla K20c (Kepler architecture) consisting of 2496 cores. Note that not all the performance gain is due to GPU, partly this is due to the reduction in overhead of R for the Heston model calculation. For comparison we also implemented the calibration code using $C/C++$. We observed a speed up of up to $230x$ for the GPU based implementation over the $C/C++$ indicating that a $3.4x$ improvement is due to avoiding the R overhead for the Heston model calculation.

Paper overview The remainder of the paper is organized as follows- Sections 2. and 3. introduce the stochastic volatility model calibration problem, option pricing formulation and numerical approximation. Section 4. provides an overview of the `gpusvcalibration` package and is intended for a reader who is less concerned with the details of the parallel implementation[1]. Section 5. describes the serial implementation of the *ErrorFunction()* and the optimization routines. Section 5.1. describes the GPU implementation of *ErrorFunction()* with a focus on the parallel implementation of the stochastic volatility pricing model. Section 6. describes the experimental performance results of the GPU off-loaded *ErrorFunction()* applied to six datasets. Section 7. concludes.

2. CALIBRATION

Calibration of an option pricing stochastic volatility model involves finding the parameters which minimize the error function - the error between the model prices and the observed prices across a set of options on the same underlying instrument, but whose contract maturities $\mathcal{T}$ and strikes $\mathcal{K}$ differ. This is formulated as a constrained non-linear least squares optimization problem of the form

$$\min_{\mathbf{z}} f(\mathbf{z}) = \left(\sum_{i=1}^{|\mathcal{K}|} \sum_{j=1}^{|\mathcal{T}|} w_{ij} [V(S_0, K_i, \tau_j; \mathbf{z}) - \hat{V}_{ij}]^2 \right)^{1/2}, \quad (1)$$

subject to the bound constraints $a_i \leq z_i \leq b_i$ (an additional non-linear constraint may be imposed by specific models). $V(S_0, K_i, \tau_j; \mathbf{x})$ denotes the model option price and $\hat{V}_{ij}$ denotes the quoted mid-price of the option with an underlying price S_0, maturity T_i and strike K_j. The overall quality of fit is sensitive to the choice of weights. An intuitive choice is to emphasize the most liquid contracts in the chain by choosing the weights to be the reciprocal of the bid-ask spread $w_{ij} = 1/(\hat{V}_{ij}^{ask} - \hat{V}_{ij}^{bid})$.

3. HESTON MODEL

The above calibration and pricing problem is presented for a wide range of stochastic volatility models, although the parameters and constraints are model dependent. However, to

[1]For the reader who is interested in learning more about GPGPU programming, Section A provides a brief introduction to GPUs and the CUDA programming environment

fix notation and detail the model which shall be used for benchmarking the GPU implementation, a brief introduction to the Heston stochastic volatility model is provided here. Please note that other stochastic volatility models are provided by the `gpusvcalibration` package and are detailed in Appendix B.

The Heston model describes the evolution of a stock price S_t whose variance V_t is given by a mean reverting square root process:

$$\frac{dS_t}{S_t} = \mu dt + \sqrt{V_t} dW_t^1, \quad (2)$$

$$\frac{dV_t}{V_t} = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t} dW_t^2, \quad (3)$$

A key characteristic of the model is that the Wiener processes are correlated $dW_t^1 \cdot dW_t^2 = \rho dt$. This feature enables the model to exhibit the 'leverage effect'. In the notation of Equation 1, the parameter set $\mathbf{z} := [\theta, \sigma, \kappa, \rho, v_0]$ and the additional non-linear constraint (the Feller condition) $2\kappa\theta - \sigma^2 > 0$ is imposed during the calibration to ensure that V_t is positive.

3.1. Pricing

With marginal loss of generality, we will restrict the scope of this section to European equity options. Stochastic volatility models permit semi-analytical closed-form solutions for computing risk neutral European option prices. The price can be represented as a weighted sum of the delta of the European call option P_1 and P_2 - the probability that the asset price will exceed the strike price at maturity. Adopting standard option pricing notation, the call price of a vanilla European option is

$$C(S_0, K, \tau; \mathbf{z}_0) = S_0 P_1 - K \exp\{-(r-q)\tau\} P_2, \quad (4)$$

P_1 and P_2 can be expressed as:

$$P_j = \frac{1}{2} + \frac{1}{\pi} \int_o^\infty \mathrm{Re}\left[\frac{\exp\{-iulnK\}\phi_j(S_0, \tau, u; \mathbf{z}_0)}{iu} \right] du, j = 1, 2. \quad (5)$$

where ϕ_j are Heston analytic characteristic functions and $\mathbf{z}_0$ is the vector of Heston model parameters. Following Fang and Oosterlee [7], the entire inverse Fourier integral in Equation 5 is reconstructed from Fourier-cosine series expansion of the integrand to give the following approximation of the call price

$$C(S_0, K, \tau; \mathbf{z}_0) \approx Ke^{-r\tau} \cdot \mathrm{Re}\{ \sum_{k=0}^{N-1}{}' \phi\left(\frac{k\pi}{b-a}; \mathbf{z}_0 \right) e^{ik\pi\frac{x-a}{b-a}} U_k \}, \quad (6)$$

where $x := ln(S_0/K)$ and $\phi(w; \mathbf{z}_0)$ denotes the Heston characteristic function of the log-asset price, U_k the payoff series coefficients and N denotes the number of terms in the cosine series expansion (typically 128 will suffice). For this approximation of the Heston model call price, the Fourier-Cosine approach is shown to be superior in convergence properties to other FFT and quadrature based methods in [6].

4. GPUSVCALIBRATION

The `gpusvcalibration` accelerates stochastic volatility model calibration by off-loading the error function on to the GPU. The package is designed for use with existing non-convex optimization CRAN packages such as `DEoptim` and `nloptr`. The library currently supports European option pricing under four different stochastic volatility models and more models are planned for the future. The library is currently available as a development version at `https://github.com/mfrdixon/gpusvcalibration` and has only been tested on Linux with Tesla generation GPU architectures.

1. *Initialization*: The option chain data must be loaded into a class and then copied on to the GPU device memory. This step also involves passing some other parameters to the model, including specification of the model type and other model constants.

2. *Execution*: `Error_function` is called with a particular model parameter instance p and returns the error estimate. For each model calibration, this function will typically be called hundreds or even thousands of times by an optimization solver.

3. *Deallocation*: Once the calibration is complete, the memory address pointers for the device and host data structures, storing the option chain data, must be deallocated.

Table 1 summarizes the core functions in the library. It is convenient to refer to source Listing 1 to understand how these functions correspond to the above three steps. Lines 4 and 5 set model constants for the short rate and the dividend yield respectively. These are assumed to be fixed across all option contracts. Line 9 calls a default `Load_Chain` function to load a snapshot of exchange quoted option chain data from a csv file into a `chain` object. This function assumes a specific format of exchange data and the user should implement their own code for populating a `chain` from a different data file format. The specification of the `chain` class is provided later in this Section. Line 10 calls the `Copy_Data` which flattens the chain, passes arrays and parameters via a C/C++ interface to C pointers and primitives, then copies the arrays and parameters from the host to the GPU device memory referenced by CUDA pointers. Lines 11 and 12 show the use of the `Set_Model` to set the stochastic volatility model and the `Set_Block_Size` to set the number of terms in the Fourier Cosine approximation respectively. The current choice of model types are {'Heston','Bates','VG','CGMY'}. We've found 256 to be an adequate number of Fourier Cosine terms in each case. This completes the initialization step.

The `Error_Function` interface enables model calibration to be executed using numerical optimization rou-

tines provided in the *R* packages `DEoptim`and `nloptr`. `DEOptim` is an evolutionary computation package which provides a Differential Evolution (DE) algorithm for global optimization [12]. `DEoptim` performs a global search in which candidate parameter sets are randomly generated and, through a selection criterion, independently evolved at each iteration of the algorithm until either *ErrorFunction* is below a threshold or the number of iterations exceeds a limit. The resulting best parameter set can be subsequently used to initialize a local optimizer which will generally refine the solution using more information about the error function and the constraints.

Lines 22-25 show the call to the `DEoptim` package for a given set of algorithm parameter choices. Specifying the appropriate constraints on the stochastic volatility model parameters is part of the challenge of calibration. The choice of box constraints shown on Lines 15 and 16 is model dependent and just provided as an example. However, all constraints, except on ρ should be positive and $\rho \in [-1,1]$. Because of small rounding errors introduced by the solver, we recommend reducing the absolute value of each bound by $\varepsilon << 1$. The best parameter set is returned to the user as accessed under DEres$optim$mem.

Lines 28 to 33 show the output of the global optimizer being used to initialize the solution parameter vector in the call to `nloptr` which is a R interface to the `NLopt` - a library for non-linear programming which implements a number of algorithms. The listing shows an example calling the COBYLA (Constrained Optimization BY Linear Approximations) algorithm [13]. This algorithm is a derivative free non-linear optimizer which is able to incorporate the non-linear inequality constraint required for the calibration to enforce the Feller condition. Lines 17 to 20 implement the non-linear inequality constraint function. `nloptr` terminates if either the norm of the difference between successive iterations of the parameter vector (relative error) is within a specified tolerance or the number of function evaluations exceeds a limit. We refer the reader to the documentation on `DEoptim` and `nloptr` for further details of argument specification and diagnostic features. The final step is to call `Dealloc_Data` in order to deallocate memory referenced by *C* pointers to host memory and CUDA device pointers.

The `Load_Chain` provides a default file parser for populating a `chain` class. The package provides example chain data taken as a single snapshot of AAPL option prices on the NYSE from the ISE/Hanweck Premium Hosted Database. These prices have been filtered for the most liquid contracts. The listing below defines the chain class for storing the option chain data at a single snapshot.

Finally, Listing 3 shows example code for performance benchmarking `Error_Function` against a R implementation `Test_Error_Function`. Further details of performance benchmarks are provided in Section 6..

5. IMPLEMENTATION

For calibrating the option price model we consider a sample chain of n option data $ch[n]$, where the ith chain data has the following key information:

- $ch[i].u$: Underlying asset price
- $ch[i].s$: Strike price
- $ch[i].m$: Time to maturity
- $ch[i].p$: Option price[2]

The calibration algorithm starts with an initial guess of the model parameters and iteratively improves the guess using an optimization algorithm until it meets the convergence criteria. A typical organization of this computation involves calling an optimization routine with a pointer to the *ErrorFunction()* given by Equation 1, which estimates the error between market observed option prices and prices calculated using the model, *SVModel()*, for the current guess of the parameter set z. More specifically, the *ErrorFunction(z)* computes option prices using the option model for a list of tuples $< ch[i].s, ch[i].m >$, $0 \leq i < n$ using the current estimates of the five parameters z and compares it with the corresponding data in $ch[i].p$. In our discussion, we focus on the parallel implementation of the *ErrorFunction(z)* as it dominates the overall computation.

A high level description of the sequential version of the *ErrorFunction(z)* for computing the root mean square error (RMSE) is given in Algorithm 1. Note that for reasons of keeping the description simple, we have avoided some subtleties of the implementation.

Algorithm 1 SEQUENTIAL-ERRORFUNCTION(z)

1: $rmse \leftarrow 0$
2: **for** $i = 0$ to $n-1$ **do**
3: $\quad vp \leftarrow$ SVMODEL($ch[i], z$)
4: $\quad diff \leftarrow ch[i].p - vp$
5: $\quad rmse \leftarrow rmse + diff \times diff$
6: **end for**
7: $rmse \leftarrow$ SQRT($rmse/n$)
8: **return** $rmse$

We implemented Algorithm 1 together with the Fourier-Cosine method and model pricing function using R (v3.0). This is made available through the `Test_Error_Function`. For calibration, we used `DEoptim` (v2.2-2) and `nloptr` (v0.9.3).

[2]We use the average price of the bid and ask (mid-price) as the option price.

Function	Description
`Copy_Data`	Copy the chain object on to the GPU device memory
`Dealloc_Data`	Delete memory allocated on the GPU device and the host for data structures
`Error_Function`	Off-loads the weighted root mean square error calculation on to the GPU device
`Load_Chain`	Default file parser for populating a chain object
`Set_Block_Size`	Set the number of terms in the Fourier-Cosine series approximation
`Set_Model`	Set the stochastic volatility model type
`Test_Error_Function`	Calculates the weighted root mean square error and prices in R for testing purposes

Table 1. This table provides a summary of the core functions and interface provided for testing in the *gpusvcalibration* library.

Listing 1. Source listing from demonstrating how to use the gpusvcalibration package.

```
library("gpusvcalibration")
library("DEoptim")            # http://cran.r-project.org/web/packages/DEoptim/DEoptim.pdf
library("nloptr")             # http://cran.r-project.org/web/packages/nloptr/vignettes/nloptr.pdf
r0 <- 0.01                    # The iannual short rate as a percentage, i.e. 0.01 = 0.01%
q0 <- 0.0                     # The annual dividend yield
eps <- 1e-8                   # Protection against rounding error in the boundary constraints
fileName <-'AAPL-Chain.csv'   # The filename containing the option chain exchange snapshot

chain <- Load_Chain(fileName) # Load a snapshot of the option chain quotes on the exchange
Copy_Data(chain)              # Copy the chain data on to the GPU device memory
Set_Model('Heston')           # Specify the stochasic vol. model {"Heston","Bates","VG",CGMY"}
Set_Block_Size(256)           # Set the number of terms in the Fourier-Cosine series approximation


l <- c(eps,eps,eps,-1.0 + eps, eps)                    # Specify lower bound on solution
u <- c(5.0-eps,1.0-eps,1.0-eps,1.0-eps,1.0-eps)        # Specify upper bound on solution
eval_g_ineq <- function (x) {                          # Implement the non-linear inequality constraint
 grad <- c(-2.0*x[2],-2.0*x[1],2.0*x[3],0,0)           # Jacobian of the Feller condition
 return(list("constraints"=c(x[3]*x[3] - 2.0*x[1]*x[2]), "jacobian"=grad))
}
                                                       # Call DEoptim to perform direct search
DEres <- DEoptim(fn=Error_Function,                    # Call the error-function
                 lower=l,
                 upper=u,
                 control=list(NP=100, itermax=25))     # Set pop. size to 100, max iter to 25

                                                       # Call nloptr to solve the constrained
res <- nloptr(x0=as.numeric(DEres$optim$bestmem),      # non-convex optimization problem.
                 eval_f=Error_Function,                # Call the error-function
                 eval_g_ineq=eval_g_ineq,
                 lb = l,
                 ub = u,
                 opts=list("algorithm"="NLOPT_LN_COBYLA", "xtol_rel" = 1.0e-7))

print(paste("Solution: ", res$solution))
print(paste("RMSE: ", res$objective))
Dealloc_Data()                # Deallocate the date from GPU device memory
```

Listing 2. Definition of the *chain* class used to represent an option chain.

```
repr <- representation(size = "integer",  # The number of option contracts in the chain
                 prices="numeric",        # The chain price vector
                 types="character",       # The chain type vector, i.e. Put ('P') or Call ('C')
                 strikes="numeric",       # The chain strike vector
                 taus = "numeric",        # The chain maturity vector
                 s = "numeric",           # The underyling price
                 weights = "numeric")     # The normalized weights used in the error_function
setClass("chain", repr)                   # Declaration of the chain class
```

Listing 3. Sample code for performance benchmarking the GPU error function off-loading.

```
library("gpusvcalibration")
r0 <- 0.01                    # The iannual short rate as a percentage, i.e. 0.01 = 0.01%
q0 <- 0.0                     # The annual dividend yield
fileName <- 'AAPL-Chain.csv'  # The filename containing the option chain exchange snapshot
m <- 'Heston'                 # Specify the stochasic vol. model {"Heston","Bates","VG",CGMY"}
p0 <- c(0.5,0.5,0.2,0.3,0.5)  # Initial model parameter values kappa, theta, sigma, rho, v0
nInt <- 256                   # Specify the number of terms in the Fourier-Cosine series approximation
chain <- Load_Chain(fileName) # Load a snapshot of the option chain quotes on the exchange
Copy_Data(chain)              # Copy the chain data on to the GPU device memory
Set_Model(m)
Set_Block_Size(nInt)

print("==GPU==")
ptm         <- proc.time()
RMSE        <- Error_Function(p0)
ptm         <- proc.time() - ptm
print(paste("Model:", m))
print(paste("Data:", fileName))
print(paste("RMSE:", RMSE))
print(paste("Elapsed time(s):",ptm[3]))
print("==R==")
ptm         <- proc.time()
RMSE        <- Test_Error_Function(p0)
ptm         <- proc.time() - ptm
print(paste("Model:", m))
print(paste("Data:", fileName))
print(paste("RMSE:", RMSE))
print(paste("Elapsed time(s):",ptm[3]))
Dealloc_Data()                # Deallocate the date from GPU device memory
```

5.1. GPU Implementation of *ErrorFunction()*

The main computation for *ErrorFunction()* is the calculation of an option price using the stochastic volatility model for a given set of parameters and an input option data point(<K,T>). Once the option price is calculated on the GPU the *RMSE* value, which now basically involves taking a difference and squaring it for an option data point, can be calculated on the CPU. The number of option data points, M, we are considering are in the range of 1024, and there will be no benefit of performing the *RMSE* calculation, which is a very small portion of the overall computation, on the GPU.

We map the stochastic volatility model computation on M blocks of the GPU. A block computes the option price for one data point. The number of threads N in a block is determined by the number of terms in the cosine series expansion. A typical value of N is 128. A high level description of the code executed by each thread of the GPU is shown in Algorithm 2.

In Algorithm 2, a thread of a block computes one cosine term of the Fourier Cosine series approximation to the stochastic volatility model price (line 6). This series approximation is given in Equation 6. The 128 terms computed by threads in a block are aggregated to give the option price (up to a multiplicative factor) for a single data point (line 6 to line 12). Observe that in line 6 we keep the result of the stochastic model calculation in shared memory. This is to avoid the overhead of using the global memory for the aggregation. The aggregation is performed using shared memory and multiple cores of the streaming processor with a tree like structure (line 6 to line 11). Note that the <K,T> values for option data points are stored on the device memory. The option price computed by a block is stored on the device memory from where it is transferred to the host memory for *RMSE* calculation. Algorithm 2 is implemented in the source file `gpusvcalibration.cu` for each of the stochastic volatility functions.

Algorithm 2 PARALLEL-FOURIER-COSINE(z)

1: *shared memory smem*[]
2: $tx \leftarrow threadIdx.x$
3: $bx \leftarrow blockIdx.x$
4: $bd \leftarrow blockDim.x$
5: $j \leftarrow bd$
6: $smem[tx] \leftarrow$ CHARACTERISTICFUNCTION($T[bx], z$)
7: **for** $i = 1$ to $log2(bd)$ **do**
8: $\quad j \leftarrow j/2$
9: $\quad$ **if** $tx < j$ **then**
10: $\quad\quad smem[tx] \leftarrow smem[tx] + smem[tx + j]$
11: $\quad$ **end if**
12: **end for**
13: **if** $tx = 0$ **then**
14: $\quad V[bx] \leftarrow K[bx] \times exp(-r0 \times T[bx] \times smem[0])$
15: **end if**
16: **return** $V[bx]$

6. EXPERIMENTAL RESULTS

All performance results reported in this section are obtained using an Intel Core i5 processor and NVIDIA Tesla K20c (Kepler architecture) consisting of 2496 cores. The CUDA compiler nvcc is release 5.0, V0.2.1221. The performance results for the four implementations are summarized in Tables 2 to 6. In the first four of these tables, we list timings for various components of the Heston model code applied to six different single-name equity option chains. Each option chain is a snapshot of tick-by-tick option prices taken over a 30 second interval and varies in size based on the number of liquid contracts over the interval. AAPL is the largest option chain at $M = 1024$.

The four versions of the calibration code are denoted: (a) *R*, (b) *RGPU*, (c) *C*, and (d) *CGPU*. The first implementation, *R*, is the base level implementation and runs sequentially on a CPU. The second implementation, *RGPU*, is the version which off-loads the *ErrorFunction()* computation to the GPU. The third version is in *C* and, by comparison with the base level implementation, is used to measure the overhead of *R* in a sequential environment. The final version of the implementation, *CGPU*, is the *C* based code where the *ErrorFunction()* computation is off-loaded to the GPU.

As mentioned in Section 5., initially we use the R implementation of the differential evolution algorithm, `DEOptim` [12] to perform a global search to estimate the calibration parameters. In the next stage, we feed this as the initial guess to a local optimizer (`nloptr` [9]). Both optimizers call the same *ErrorFunction()* routine during the optimization process. The population size in DEOptim is set to 100 and the algorithm is set to perform a single iteration. The relative tolerance in the COBYLA routine is set to 1.0×10^{-7} and the maximum number of iterations is set to 50.

For the Heston model, we observe that the COBYLA routine always converges in under 50 iterations and that the number of *ErrorFunction()* evaluations performed by the COBYLA routine does not vary across the datasets. While nloptr is simpy a wrapper to a C++ implementation of `NLopt`, the `DEOptim` library provided in R is only loosely based on version 4.0 of the original C implementation by Storn and Price [15]. We therefore observe slight variations in timings and numerical results between the R and C versions of `DEOptim`, which we have tried to minimize during performance benchmarking by just performing one iteration of the DE algorithm. Table 2 shows the overall timing of the R base level implementation of the calibration code for each of the six option chains. We observe that the calibration is dominated by the *ErrorFunction()*, constituting at least 99.0% and takes up to 441 seconds for the AAPL chain.

	AAPL	AMZN	BP	CSCO	GOOG	MSFT
DEoptim	293	111	69	62	255	70
nloptr	148	56	35	32	130	36
ErrorFunction	1.46	0.55	0.34	0.31	1.28	0.35
Total ErrorFunction	440	166	103	93	385	105
Total Time	441	167	104	94	386	106
% ErrorFunction	99.8%	99.4%	99.1%	99.0%	99.7%	99.1%

Table 2. Performance results for the R code in seconds. Each columns represents a different option chain.

	AAPL	AMZN	BP	CSCO	GOOG	MSFT
DEoptim	88	33	21	18	76	22
Nlopt	44	17	10	9	38	11
ErrorFunction	0.44	0.17	0.1	0.1	0.38	0.11
Total ErrorFunction	131	49	30	27.6	113	32
Total Time	132	50	31	27	114	33
% ErrorFunction	99.2%	98.0%	96.8%	99.9%	99.1%	97.0%

Table 3. Performance results for the C code in seconds.

Table 3 shows the performance results of the RGPU implementation. The overall time falls to less than 0.5*s* and the *ErrorFunction()* now only accounts for up to 72.4%. In Table 4, the performance results of the C implementation show that the overall calibration time is reduced by up to 3.4x compared to the R base level implementation. As previously indicated, the `DEOptim` time between the R and C implementations is not strictly comparable due to variations in implementations, but the factor reduction in the total time in the error function is more consistent across datasets. The CGPU implementation performance results provided in Table 5 show that *ErrorFunction()* only constitutes up to 67.7% and the overall calibration time is up to 0.62s.

These comparative results of the *ErrorFunction* timings are summarized for each option chain in Table 6. The first row shows the speedup by off-loading the *ErrorFunction* to the GPU, which is up to 1042x for the AAPL option chain. The second row compares the C implementation with the RGPU implementation and the third row compares the CGPU implementation with the RGPU version. We observe here and by comparing the overall timings in Tables 3 and 5 that the overhead of the R wrapper is marginal and hence any benefit of the CGPU code, in practical terms, is offset by the convenience of using the R environment and off-loading the *ErrorFunction()* to the GPU.

7. CONCLUSIONS

This paper has described the `gpusvcalibration` package for accelerating stochastic volatility model calibration on GPUs. The package is based on a GPU optimized kernel for error function evaluation which can be called by CRAN optimization libraries such as `DEOptim` and `nloptr` for calibration of stochastic volatility models. For $M = 1024$ we demonstrate a factor of 760*x* improvement in the overall calibration time over the R sequential implementation by off-loading *ErrorFunction()* on a system with an Intel Core i5 processor and NVIDIA Tesla K20c (Kepler architecture) consisting of 2496 cores. Note that not all the performance gain is due to the GPU- partly it is due to the reduction in the overhead of R for the stochastic volatility model calculation. For comparison we also implemented the calibration code using $C/C++$. We observed a speed up of 230*x* for the GPU based implementation over the $C/C++$ indicating that a factor of 3.4*x* improvement is due to avoiding the *R* overhead for the stochastic volatlity model calculation. However, the overall calibration time using R based optimization routines combined with the GPU off-loaded *ErrorFunction()*— is comparable to a $C/C++$ GPU based calibration code.

	AAPL	AMZN	BP	CSCO	GOOG	MSFT
DEoptim	0.31	0.12	0.08	0.074	0.29	0.08
nloptr	0.16	0.063	0.044	0.041	0.15	0.044
ErrorFunction (ms)	1.41	0.56	0.36	0.33	1.25	0.36
Total ErrorFunction	0.42	0.17	0.11	0.1	0.38	0.11
Total Time	0.53	0.23	0.17	0.16	0.44	0.18
% ErrorFunction	79.2%	73.9%	64.7%	62.5%	86.3%	61.1%

Table 4. Performance results for the RGPU code. Timings are shown in seconds unless stated otherwise.

	AAPL	AMZN	BP	CSCO	GOOG	MSFT
DEoptim	0.38	0.18	0.14	0.13	0.4	0.14
Nlopt	0.16	0.06	0.04	0.04	0.16	0.04
ErrorFunction(ms)	1.4	0.56	0.364	0.34	1.24	0.36
Total ErrorFunction	0.42	0.17	0.11	0.1	0.37	0.11
Total Time	0.54	0.24	0.18	0.17	0.56	0.18
% ErrorFunction	77.7%	70.8%	61.1%	58.8%	66.1%	61.1%

Table 5. Performance results for the CGPU code. Timings are shown in seconds unless stated otherwise.

	AAPL	AMZN	BP	CSCO	GOOG	MSFT
R/RGPU	1042	992	971	933	1024	961
C/RGPU	313	297	288	278	303	297
CGPU/RGPU	1	1	1	1	1	1

Table 6. Relative performance of the RGPU code.

8. ACKNOWLEDGMENTS

The authors would like to gratefully acknowledge the support of Hanweck Associates and the International Securities Exchange in providing access to a data sample from the ISE/Hanweck Premium Hosted Database.

REFERENCES

[1] M. Aichinger, A. Binder, J. Furst, and C. Kletzmayr. A Fast and Stable Heston Model Calibration on the GPU. In *Euro-Par Proc. 2010 Conference on Parallel processing*, pages 431–438, 2010.

[2] D. Ardia, J. David, O. Arango, and N. Gomez. Jump-Diffusion Calibration using Differential Evolution. *Wilmott Magazine*, 55:76–79, Sept. 2011.

[3] D. Bates. Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options. *Review of Financial Studies*, 9:69–107, 1996.

[4] J. Buckner, J. Wilson, M. Seligman, B. Athey, S. Watson, and F. Meng. The gputools package enables GPU computing in R. *Bioinformatics*, 26(1):134–135, 2010.

[5] P. Carr, H. Geman, D. Madan, and M. Yor. The fine structure of asset returns: An empirical investigation. *J. of Business*, 75:305–332, 2002.

[6] M. Dixon and M. Zubair. Calibration of Stochastic Volatility Models on a Multi-Core CPU Cluster. In *Proceedings of the Sixth Workshop on High Performance Computational Finance at SC13*, 2013.

[7] F. Fang and C. W. Oosterlee. A Novel Pricing Method for European Options based on Fourier-Cosine Series Expansions. *SIAM Journal on Scientific Computing*, 31:826–848, 2008.

[8] S. Heston. A Closed-form Solution for Options with Stochastic Volatility. *Review of Financial Studies*, 6:327–343, 1993.

[9] S. G. Johnson. The NLopt nonlinear-optimization package.

[10] J. Knaus, C. Porzelius, H. Binder, and G. Schwarzer. Easier Parallel Computing in R with Snowfall and sfCluster. *The R Journal*, 1:54–59, 2009.

[11] D. Madan, P. Carr, and E. C. Chang. The Variance Gamma Process and Option Pricing. *European Finance Review*, 2:79–105, 1998.

[12] K. Mullen, D. Ardia, D. Gil, D. Windover, and J. Cline. DEoptim: An R Package for Global Optimization by Differential Evolution. *Journal of Statistical Software*, 40(6):1–26, 2011.

[13] M. J. D. Powell. A Direct Search Optimization Method that Models the Objective and Constraint Functions by Linear Interpolation. *Advances in Optimization and Numerical Analysis*, pages 51–67, 1994.

[14] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State of the Art in Parallel Computing with R. *Journal of Statistical Software*, 31(1):1–27, 8 2009.

[15] R. Storn and K. Price. Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization. 11(4):341–â359.

[16] H. Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 2011.

Biography

Matthew Dixon is a Term Assistant Professor in the MS in Analytics Program at the University of San Francisco. He is also a consulting director of risk for HedgeFacts, LLP, a portfolio analytics and fund administration platform for hedge funds. In addition to holding academic appointments as Krener Assistant Professor at UC Davis and postdoctoral researcher at Stanford University, Matthew has worked and consulted for various investment banks and the Bank for International Settlements on quantitative risk methodology. He serves on the Global Association of Risk Professionals San Francisco chapter committee and co-chairs the workshop on high performance computational finance at SC, the International Conference for High Performance Computing, Networking, Storage and Analysis. Matthew holds a Ph.D. in Applied Math (2007) from Imperial College and a M.Sc. in Parallel and Scientific Computation with Distinction (2002) from Reading University, UK.

Sabbir Ahmed Khan is a first year Ph.D. student in Computer Science Department at Old Dominion University, Virginia. He is working with Dr. Zubair. His primary interest is in the area of High Performance Computing. He did his BSc from CSE department of Bangladesh University of Engineeing and Technology (BUET). He also worked as a Software Engineer at Uniqa software and Systems Ltd. and Beximco Pharmaceuticals Ltd.

Mohammad Zubair is a Professor in the Computer Science Department at Old Dominion University. Prof. Zubair has more than twenty years of research experience in the area of experimental computer science and engineering both at the university as well as in Industry. His primary area of interest is high performance computing and management of large information. His major industrial assignment was at the IBM T.J. Watson Research center for three years, where his research focus was in high performance computing and some of his work was integrated into IBM products. His current interests are in developing high performance algorithms for multi-core architectures such as GPUs and Intel Multi-core systems. He has been successful in obtaining funds to support his research work from NASA, NSF, DTIC, ARPA, Jefferson Laboratory, Los Alamos, AFRL, NRL, JTASC, Sun Microsystems, and IBM Corporation.

A CUDA PROGRAMMING ENVIRONMENT

A typical program on a system with a single GPU device is a C/C++ program with CUDA APIs to move data between system memory and GPU device memory, and to launch computation kernels on GPU. The data between system memory and the device memory is moved using the PCI

Express (PCIe) bus. These transfers are costly and therefore applications that have a higher computation to I/O ratio are suitable for GPU computing. Also, if possible these transfers should be minimized and it is desirable to leave the data on the GPU if a subsequent kernel is going to use the same data. A GPU device uses several memory spaces that differ in their size, access latency, and read/write restrictions. These memory spaces include global, local, shared, texture, and registers. Global, local, and texture memory have the greatest access latency, followed by constant memory, registers, and shared memory.

The GPU device works best for computations that can be executed concurrently on multiple data elements. In general, given an application one would like to partition the computational requirement into thousands of small computations that can be executed simultaneously. These computations are assigned to thousands of threads of the GPU which are executed concurrently on different cores. When implementing applications on a system with multiple GPU devices, the approach for parallelization has to be adjusted. For this case, we partition the application in as many coarse-level chunks of computation as the number of devices available on the system. Next for each chunk, we partition the computation requirement as before into thousands of small computations that can be executed simultaneously. CUDA provides an abstraction of thread hierarchy to allow computation from different domains to map to different cores of the underlying hardware. The GPU hardware consists of a number of streaming multiprocessors which in turn consist of multiple cores. Threads are organized in blocks, where one or more block runs on a streaming multiprocessor. The threads in a block are further partitioned into subgroups of 32 threads referred to as 'Warps'. A Warp, that is a sub block of 32 threads, runs on eight or sixteen cores of a streaming multiprocessor in multiple clock cycles.

B STOCHASTIC VOLATILITY MODEL DESCRIPTIONS

The stochastic volatility models also implemented in the library are briefly listed here. This section is provided to briefly describe the model parameters and is not intended as a self-contained description of the models.

2.1. Bates Model

The Bates Jump-Diffusion model [3] is specified as the following set of coupled stochastic differential equations

$$\frac{dS_t}{S_t} = \mu dt + \sqrt{V_t} dW_t^1 + (Y-1) S_t dN_t, \quad (7)$$

$$\frac{dV_t}{V_t} = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t} dW_t^2, \quad (8)$$

describing the evolution of a stock price S_t whose variance V_t is given by a mean reverting square root process which ensures that the variance is always positive provided that $2\kappa\theta - \sigma^2 > 0$. N_t is a standard Poisson process with intensity $\lambda > 0$ and Y is the log-normal jump size distribution with mean $\mu_j = ln(1+a) - \frac{\sigma_j^2}{2}$, $a > -1$ and standard deviation $\sigma_j \geq 0$.

Both $N(t)$ and Y are independent of the Wiener processes W_t^1 and W_t^2. A key characteristic of the model, which originates from the embedded Heston stochastic volatility diffusion model, is that the Wiener processes are correlated $dW_t^1 \cdot dW_t^2 = \rho dt$. This feature enables the model to exhibit the leverage effect. Note that simply excluding the compound Poisson term $(Y-1)S_t dN_t$ recovers the Heston model.

2.2. Variance Gamma Model

Following [11], the stock price dynamics may be generalized beyond the Brownian motion in the original geometric Brownian motion model by a VG process. Under this model, the stock price at time t is given by a three parameter Lévy process L:

$$S(t) = S(0)exp(mt + L(t;\theta,\nu,\sigma) + \omega(t)), \quad (9)$$

where m is the mean rate of return on the stock under the statistical probability measure and the Martingale correction term is $\omega(t) = \frac{t}{\mu} ln(1 - \theta\nu - \sigma^2\nu/2)$. The parameters θ, ν, σ only indirectly reflect the skewness and kurtosis of the return distribution. θ by itself determines the overall scale of the volatility. The form of the characteristic function is provided in [11].

2.3. CGMY Model

The CGMY model [5] is a more general case of the Variance Gamma model. The parameters in the VG model can be mapped to the CGM representation using the parameters transforms

$$C = \frac{1}{\nu}$$

$$G = \left(\sqrt{\frac{\theta^2\nu^2}{4} + \frac{\sigma^2\nu}{2}} - \frac{\theta\nu}{2}\right)^{-1}$$

$$M = \left(\sqrt{\frac{\theta^2\nu^2}{4} + \frac{\sigma^2\nu}{2}} + \frac{\theta\nu}{2})\right)^{-1}$$

The model parameters are restricted to $C, G, M > 0$ and an additional parameter which controls the peakedness of the probability density function is introduced $Y < 2$. The case $Y = 1$ corresponds to the VG model. The form of the characteristic function is provided in [5].