A Portable, Extensible and Fast Stochastic Volatility Model Calibration using Multi and Many-Core Processors

Matthew Dixon\textsuperscript{1}\thanks{Correspondence to: Department of Analytics, University of San Francisco, San Francisco, CA E-mail: mfdixon@usfca.edu} Jörg Lotze\textsuperscript{2} and Mohammad Zubair\textsuperscript{3}

\textsuperscript{1}Department of Analytics, University of San Francisco, San Francisco, CA
\textsuperscript{2}Xcelerit, 28–32 Pembroke Street, Dublin 2, Ireland
\textsuperscript{3}Department of Computer Science, Old Dominion University, Norfolk, VA

SUMMARY

Financial markets change precipitously and on-demand pricing and risk models must be constantly recalibrated to reduce risk. However, certain classes of models are computationally intensive to robustly calibrate to intraday prices- stochastic volatility models being an archetypal example due to the non-convexity of the objective function. In order to accelerate this procedure through parallel implementation, financial application developers are faced with an ever growing plethora of low-level high-performance computing frameworks such as OpenMP, OpenCL, CUDA, or SIMD intrinsics, and forced to make a trade-off between performance versus the portability, flexibility and modularity of the code required to facilitate rapid in-house model development and productionization.

This paper describes the acceleration of stochastic volatility model calibration on multi-core CPUs and GPUs using the Xcelerit platform. By adopting a simple programming model, the Xcelerit platform enables the application developer to write sequential, high-level C++ code, without concern for low-level high-performance computing frameworks. This platform provides the portability, flexibility and modularity required by application developers. Speedups of up to 30x and 293x are respectively achieved on an Intel Xeon CPU and NVIDIA Tesla K40 GPU, compared to a sequential CPU implementation. The Xcelerit platform implementation is further shown to be equivalent in performance to a low-level CUDA version. Overall, we are able to reduce the entire calibration process time of the sequential implementation from 6,189 seconds to 183.8 and 17.8 seconds on the CPU and GPU respectively without requiring the developer to reimplement in low-level high performance computing frameworks.

Copyright © 2010 John Wiley & Sons, Ltd.

KEY WORDS: Calibration, Stochastic Volatility, GPGPU, C++

1. INTRODUCTION

Financial markets change precipitously and on-demand pricing and risk models must be constantly recalibrated to reduce model risk. The problem of how to robustly calibrate stochastic volatility models to quoted option prices is a topic of great importance in financial modeling. Stochastic volatility models are used to build implied volatility surfaces which are subsequently used to price more exotic options which trade in thin markets. Hence the mis-calibration of stochastic volatility models to liquid vanilla options can lead to significant pricing anomalies in relatively illiquid exotic options.

A serial implementation of the Heston stochastic volatility model calibration on a CPU does not meet the performance requirements for on demand calibration to sub-minute best bid-offer...
snapshots of liquid option chains. **Financial application developers** seek to deploy computationally intensive calculations on accelerator platforms without concern for low-level high performance implementations. This preference arises from an “implementation skillset gap” - oftentimes high level programmers, with application domain expertise, lack the low-level high performance programming skills to be able to deploy their code on multi and many-core accelerator platforms [1][2]. Moreover, platform optimized variants of the code, reduce maintainability and portability of the production code, and lengthen software testing and thus time to deployment.

**Software managers** must confront a number of practical considerations too. For example, the decision to reimplement legacy C++ production code in to low-level languages increases operational risk because of increased complexity of the code and increased dependency on specialist programming skill-sets, which may become increasingly difficult and expensive to retain over time.

In this paper we describe our latest work on speeding up the calibration of stochastic volatility models on multi-core CPUs and GPUs using the Xcelerit platform. The calibration of a stochastic volatility model is performed over n 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 model prices for the current guess of the parameter set. This paper describes the efficient implementation of the ErrorFunction() on multi and many-core architecture using the Xcelerit platform.

For the Heston model, speedups of up to 29.5x and 293x are respectively achieved on an Intel Xeon CPU and NVIDIA Tesla K40 GPU, compared to a sequential CPU implementation. The Xcelerit platform implementation is further shown to be equivalent in performance to a low-level CUDA version. Overall, we are able to reduce the entire Heston model calibration process time of the sequential implementation from 6,189 seconds to 183.8 and 17.8 seconds on the CPU and GPU respectively.

1.1. Literature review

Heston’s stochastic volatility [3] has drawn widespread usage for its ability to capture the volatility smile and skew due to, for example, leverage effects. However the robust and fast calibration of this model requires judicious choice of numerical approximation and optimization schemes.

Mikhailov and Nögel [4] address the problem of calibrating Heston’s stochastic volatility model by providing guidance on how to calibrate the model to vanilla call options. The authors draw attention to the fact that the calibration procedure is non-trivial – it is a non-linear programming problem with a non-linear constraint and non-convex objective function. Since multiple local-minima may exist, Mikhailov et al. propose using a combination of global search and local optimizers. The authors further note that the use of common stochastic algorithms for global search, such as simulated annealing, generally renders the calibration problem more computationally burdensome.

To improve the performance of this calibration approach, Aichinger, Binder, Fürst and Kletzmayr [5] 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. Aichinger et al. compare the stability and performance of various off-the-shelf 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 [6]. Finally, the stability of the hybrid calibration method is evaluated on a snapshot of the option chain on the FTSE-100 index.

The use of an unconstrained local optimizer is unsatisfactory for two reasons. First, bounds on the model parameters are necessary to ensure that the solution lies in a feasible region which is consistent with the modeling parameter definitions. Second, the non-linear constraint enforces the
Feller condition which prevents the stochastic volatility process from reaching zero. Neither of these conditions are satisfied apriori by the use of the Levenberg-Marquardt optimizer and the latter condition is not satisfied by using an off-the-shelf global solver either. The authors address a violation of the Feller condition by a posteriori adjustment of the parameter set to satisfy the Feller condition. However, as the authors further note, this adjustment reduces the performance of the local solver.

Dixon and Zubair [7] and Dixon, Khan and Zubair [8] approach the problem from a parallel architecture and software engineering perspective motivated by [9]. Starting from the premise that implementation of financial models in high level programming environments and languages such as R and Python is conducive to design exploration, rapid prototyping and model validation, the authors evaluate parallel implementations of the calibration designed for clusters of multi-core CPUs and GPUs.

For the R implementation, they show how to efficiently off-load the error function into the CUDA programming environment using a map-reduce design pattern [9]. While the implementation is highly efficient, the performance increase comes at the price of compromising model flexibility, modularity and portability because the stochastic volatility model must be reimplemented in CUDA instead of enabling modeling specific code to remain implemented in R. Furthermore, the parallel implementation is specific to GPU architectures with more cores than there are options in the chain, rather than hardware independent.

While R and Python are very convenient high level modeling languages, C++ and C# still remain the primary languages used by quants in investment banks because many performance critical models are often already available. The contribution of this paper is to demonstrate and evaluate how the Xcelerit platform can be used to efficiently calibrate stochastic volatility models on either GPUs or multi-core CPUs with a single code base written in C++. We begin by introducing the calibration procedure in Section 2. This is followed by a case study introducing the Heston model and the option pricing formula. Section 5 briefly describes the Xcelerit platform and demonstrates how the tool can be used to implement portable, extensible and accelerated stochastic volatility calibration algorithms. Performance benchmarks of the Xcelerit implementation are presented in Section 6. A brief overview of the extension of the approach to other stochastic volatility models is provided in Section 7. Section 8 concludes.

2. CALIBRATION

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

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

subject to the bound constraints \( a_i \leq z_i \leq b_i \) (an additional non-linear constraint may be imposed by specific models). \( V(S_0, K_i, \tau_j; x) \) denotes the model option price and \( \hat{V}_{ij} \) denotes the quoted mid-price of the option with an underlying price \( S_0 \), maturity \( \tau_j \) and strike \( K_i \). 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. PROGRAMMING MODEL

The Xcelerit platform is a tool which makes it easy for financial application developers to develop high performance applications in C++ [10, 11]. Xcelerit-enabled applications efficiently make use of
Multi-core CPUs, GPUs, and combinations of these in a grid from a single high-level code-base. The source code is free from parallel constructs, and the efficient execution is managed automatically.

To use the Xcelerit API, compute-intensive algorithms are expressed as a graph of processing stages (termed actors) which are connected together. A series of actors are continuously applied to streams of data, flowing through the graph. Source actors generate the data, generic actors take data from their input ports and compute the output data to be placed into their output ports, and sink actors consume data (for example, save it into a file).

Figure 1 shows a processing graph for the SV model calibration. The SequenceSource source actor generates all combinations of \( i, j \) indices, the IFourierCosine actor evaluates each Fourier-Cosine series term over all options in the chain, the PartialSumReduce actor sums the Fourier-Cosine series to obtain a Heston model price for each option, and the Error sink actor computes the root mean square of the error between the model and quoted option prices.

As demonstrated in Appendix A, Actors are implemented as regular C++ classes. To write a program using the Xcelerit API, actor objects are instantiated and set up, and then connected to form a graph. When this graph is run, the Xcelerit SDK runtime library manages its efficient execution on the available computation devices, such as multi-core CPUs and GPUs. The Xcelerit API ensures that common parallel programming bugs such as race conditions or deadlocks cannot be introduced.

The Xcelerit SDK takes care of the parallelisation and hardware specific optimizations. For example, it employs data and task parallelism, overlaps data transfers with computation tasks to hide latencies, ensures efficient memory access on the execution device, and uses memory locality to reduce data access latencies. It also optimizes the execution at runtime for the specific target execution device used, for example a Kepler-generation GPU or an Ivy Bridge server CPU, and supports using multiple execution devices. More technical details about the Xcelerit platform can be found in [10, 11].

A concrete implementation for a specific stochastic volatility model is implemented by deriving from the IFourierCosine class and providing a method to compute one Fourier term for a specific option in the chain. This way, different models can be added easily and immediately benefit from the performance of multi-core CPUs and GPUs.

4. EXAMPLE: HESTON MODEL

The calibration problem and implementation described earlier in this paper generalize to a wide range of stochastic volatility models. However, to fix notation and detail the model which shall be used for benchmarking the Xcelerit implementation, a brief introduction to the Heston stochastic volatility model is provided here.

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:

\[
\begin{align*}
\frac{dS_t}{S_t} &= \mu dt + \sqrt{V_t} dW_t^1, \\
\frac{dV_t}{V_t} &= \kappa (\theta - V_t) dt + \sigma \sqrt{V_t} dW_t^2,
\end{align*}
\]

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'.

4.1. Pricing

With marginal loss of generality, we will restrict the scope of this section to European equity options. The Heston stochastic volatility model permits 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

\[
C(S_0, K, \tau; z_0) = S_0 P_1 - K e^{-(r-q)\tau} P_2,
\]

$P_1$ and $P_2$ can be expressed as:

\[
P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \Re \left[ \phi_j(S_0, \tau, u; z_0) e^{-iu \ln K} \right] du, \quad j = 1, 2.
\]

where $\phi_j$ are Heston analytic characteristic functions and are given in a convenient form in [12], and $z_0$ is the vector of Heston model parameters. Following Fang and Oosterlee [13], 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

\[
C(S_0, K, \tau; z_0) \approx K e^{-r\tau} \Re \left[ \sum_{k=0}^{N-1} \phi \left( \frac{k\pi}{b-a}; z_0 \right) e^{ik\pi \frac{x-a}{b-a}} U_k \right],
\]

where $x := \ln(S_0/K)$ and $\phi(w; z_0)$ 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). The Fourier-Cosine approach is shown to be superior in convergence properties to other FFT and quadrature based methods in [7].

5. IMPLEMENTATION

Note unless otherwise stated that the details in this section are model agnostic. For calibrating the option price model we consider a sample chain of $n$ option data $ch[n]$, where the $i$th 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

\[\text{We use the average price of the bid and ask (mid-price) as the option price.}\]
The calibration algorithm starts with an initial guess of five 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 $\text{ErrorFunction}(p)$ given by Equation (1), which estimates the error between market observed option prices and prices calculated using the model, $V()$. For the current guess of the parameter set $p$. More specifically, the $\text{ErrorFunction}(p)$ computes option prices using the option model for a list of tuples $<c[i], u, ch[i], s, ch[i], m, p>$, $0 \leq i < n$ using the current estimates of the parameter vector $p$ and compares it with the corresponding data in $ch[i], p$. In our discussion, we focus on the parallel implementation of the $\text{ErrorFunction}(p)$ as it dominates the overall computation.

A high level description of the sequential version of the $\text{ErrorFunction}(p)$ is given in Algorithm 1. Note that for reasons of keeping the description simple, we have avoided some subtleties of the implementation.

We implemented Algorithm 1 together with the Fourier-Cosine method and model pricing function in C++. The $\text{ErrorFunction}(p)$ is called by non-linear optimization packages NLopt and DEoptim. More specifically, we combine the DEoptim global optimizer with one of three constrained local optimization solvers provided in the NLopt package. These optimizers are (i) the Sequential Least Squares Programming (SLSQP) method; (ii) the L-BFGS-B algorithm; and (iii) the Truncated Newton (TNC) method. Each method exploits the smoothness of the error function over the feasible region by approximating the Jacobian with first order forward differences under perturbations of each parameter. A small number of Hessian vectors are also computed at each main iteration in the L-BGFS-B algorithm. The SLSQP method has the advantage that the change of parameterization detailed in Section 2 is not necessary as the method already incorporates non-linear inequality constraints.

The number of function evaluations per iteration is thus dependent on the number of model parameters. The global optimizer is terminated if either the objective function is below a threshold or the number of iterations exceeds a limit. The specifiable stopping criterion varies for each of the local optimizers. However, for ease of comparison of convergence properties between each, it is possible to terminate if either the absolute difference in function values between successive iterations is within a tolerance or the number of function evaluations exceeds a limit. In practice, a tolerance on the absolute difference of the function value is neither intuitive or ideally suited to calibration. In further experiments, not reported here, we find that specifying the tolerance on the norm of the difference in solution iterates leads to more stable parameters over successive calibrations. Of the three aforementioned local solvers, only the TNC method permits a tolerance of this form.

When using the Heston or Bates diffusion model, it is necessary when applying off-the-shelf global and local solvers to Equation (1) with a priori enforcement of the Feller constraint, to apply the following transformation $\kappa' = 2\kappa \theta - \sigma^2$, with bounds $\epsilon \leq \kappa' \leq b'$ where $\epsilon << 1$ and $b'$ is given by twice the product of the upper bounds on $\kappa$ and $\theta$ minus the square of the lower bound on $\sigma$. The parameter set in Equation (1) becomes $z' := [\theta, \sigma, \kappa', \rho, v_0]^\top$ subject to $a' \leq z' \leq b'$. The original parameter $\kappa$ is recovered from $\kappa = \frac{\kappa' + \sigma^2}{2b'}$ and since $\kappa', \theta > 0$ it follows trivially that $\kappa > 0$. We note however that we have surrendered direct control of the bounds of $\kappa$, although in practice this can be
accommodated provided there is flexibility on the choice of bounds of \( \sigma \) and \( \theta \). The other constraints are unaffected, \( \rho \in [-1,1], \nu_0 > 0, \alpha > -1, \sigma_j > 0 \) and \( \lambda > 0 \).

### 6. EXPERIMENTAL RESULTS

All performance results reported in this section are obtained using an Intel Xeon CPU and an NVIDIA Tesla K40 GPU (Kepler architecture). These results are limited to calibration of the Heston model. The details of the configuration are given in Table I.

In our earlier experiments [7, 8] we observed that the overall time to calibrate the Heston model is dominated by the ErrorFunction() routine. For this reason, we report the execution time of a single iteration of ErrorFunction() in Tables II and III. Table II compares the timing results in milliseconds of the Xcelerit and CUDA implementations of the Heston error function applied to the AAPL chain along with the speedups against the serial version (shown in the parentheses). The Table shows the comparative performance of the Xcelerit and CUDA versions with varying numbers of Fourier-Cosine terms. Table III shows the performance results of the serial, CUDA and Xcelerit implementations applied to six different single-name equity option chains. The number of Fourier-Cosine terms is kept fixed at 4096. Option chain sizes vary in size, both across instrument and also over time because the number of contracts is dependent on the size of the mid-market. AAPL is the largest option chain at \( n = 1546 \).

We observe that the Xcelerit version outperforms the CUDA implementation in most cases. As the Xcelerit SDK is designed for compute-intensive workloads, a small overhead compared to CUDA can be seen with lighter workloads in Table II (less than 10ms runtime). Nvidia’s Visual Profiler showed that the Xcelerit SDK makes more efficient use of GPU registers, which results in approximately 12% higher occupancy of the Tesla K40 GPU compared to the CUDA implementation (for 4,096 Fourier terms). Further, the utilization of the arithmetic unit is 15% higher with the Xcelerit version. Both versions are compute-bound, i.e., bound by the arithmetic units of the GPU.

The Xcelerit version scales well on the CPU (without source code changes), achieving speedups between 26x and 29.5x compared to the sequential version.

To provide insight into the overall execution time, we also report the total calibration time in Table V.

Tables IV and V respectfully show the optimization parameters and compare the overall timings of the calibration across each local optimizer and serial and parallel implementation. The parallel efficiency of the overall calibration is comparable to the parallel efficiency of the error function. However, the number of iterations to convergence varies across parallel implementation and hence the overall calibration time is not purely determined by the error function cost.
Table II. This table compares the elapsed wall-clock time of the error function in milliseconds between the serial, CUDA and Xcelerit CPU and Xcelerit GPU using the AAPL chain data. The error function is evaluated using varying number of Fourier-Cosine terms. Speedups relative to the sequential version are shown in parentheses.

<table>
<thead>
<tr>
<th>Chain</th>
<th>Serial</th>
<th>Xcelerit CPU</th>
<th>CUDA</th>
<th>Xcelerit GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAPL</td>
<td>7,112.7</td>
<td>248.30 (28.6x)</td>
<td>29.40 (242x)</td>
<td>7.41 (265x)</td>
</tr>
<tr>
<td>AMZN</td>
<td>2,701.1</td>
<td>92.77 (29.1x)</td>
<td>11.31 (239x)</td>
<td>9.26 (292x)</td>
</tr>
<tr>
<td>BP</td>
<td>1,679.9</td>
<td>61.17 (27.5x)</td>
<td>7.30 (230x)</td>
<td>6.35 (265x)</td>
</tr>
<tr>
<td>CSCO</td>
<td>1,485.9</td>
<td>52.57 (28.3x)</td>
<td>6.59 (225x)</td>
<td>5.87 (253x)</td>
</tr>
<tr>
<td>GOOG</td>
<td>6,335.6</td>
<td>214.60 (245x)</td>
<td>19.99 (317x)</td>
<td>17.8 (349x)</td>
</tr>
<tr>
<td>MSFT</td>
<td>1,708.3</td>
<td>59.42 (28.8x)</td>
<td>7.41 (231x)</td>
<td>6.35 (269x)</td>
</tr>
</tbody>
</table>

Table III. This table compares the elapsed wall-clock time of the error function in milliseconds between the serial, CUDA and Xcelerit CPU/GPU implementations for six single-name equity option chains for 4,096 Fourier terms. Speedups relative to the sequential version are shown in parentheses.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>Step size of finite differences</td>
<td>$1.0 \times 10^{-5}$</td>
</tr>
<tr>
<td>$\tau_l$</td>
<td>Rel. tol. of local solver</td>
<td>$1.0 \times 10^{-3}$</td>
</tr>
<tr>
<td>$\tau_g$</td>
<td>Tol. of global solver</td>
<td>0.01</td>
</tr>
<tr>
<td>$N_{gmax}$</td>
<td>Max iter. of global solver</td>
<td>400</td>
</tr>
</tbody>
</table>

Table IV. This table shows the parameters of the global and local calibration routines used to produce the benchmark results shown in Table V.

<table>
<thead>
<tr>
<th>#terms</th>
<th>Solver</th>
<th>Serial time (N)</th>
<th>Xcl CPU time (N)</th>
<th>Xcl GPU time (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>SLSQP</td>
<td>733 (448)</td>
<td>24.8 (448)</td>
<td>3.7 (448)</td>
</tr>
<tr>
<td></td>
<td>LBFGSB</td>
<td>903 (454)</td>
<td>32.8 (459)</td>
<td>6.3 (487)</td>
</tr>
<tr>
<td></td>
<td>TNC</td>
<td>4,109 (810)</td>
<td>107.0 (678)</td>
<td>45.3 (1,209)</td>
</tr>
<tr>
<td>1,024</td>
<td>SLSQP</td>
<td>1,515 (454)</td>
<td>49.2 (448)</td>
<td>5.7 (448)</td>
</tr>
<tr>
<td></td>
<td>LBFGSB</td>
<td>2,598 (497)</td>
<td>76.8 (478)</td>
<td>7.0 (453)</td>
</tr>
<tr>
<td></td>
<td>TNC</td>
<td>7,750 (783)</td>
<td>331.3 (860)</td>
<td>26.0 (689)</td>
</tr>
<tr>
<td>2,048</td>
<td>SLSQP</td>
<td>3,093 (444)</td>
<td>94.6 (444)</td>
<td>9.4 (444)</td>
</tr>
<tr>
<td></td>
<td>LBFGSB</td>
<td>4,047 (459)</td>
<td>126.9 (458)</td>
<td>13.0 (461)</td>
</tr>
<tr>
<td></td>
<td>TNC</td>
<td>18,123 (838)</td>
<td>357.9 (459)</td>
<td>35.6 (627)</td>
</tr>
<tr>
<td>4,096</td>
<td>SLSQP</td>
<td>6,189 (441)</td>
<td>183.8 (441)</td>
<td>16.4 (441)</td>
</tr>
<tr>
<td></td>
<td>LBFGSB</td>
<td>7,451 (447)</td>
<td>332.6 (486)</td>
<td>31.1 (490)</td>
</tr>
<tr>
<td></td>
<td>TNC</td>
<td>21,349 (627)</td>
<td>805.3 (663)</td>
<td>241.7 (1,339)</td>
</tr>
</tbody>
</table>

Table V. This table compares the overall time (in seconds) to calibrate the Heston model across the local solvers and sequential and parallel implementations. The number of iterations $N$ is shown in parentheses.

We find that the SLSQP solver not only requires the least number of iterations to converge, but this number stays relatively stable with varying data and choice of parallel implementation. Using the Xcelerit platform (with 4096 Fourier-Cosine terms), the overall time for calibrating the sequential implementation of the Heston model with the SLSQP solver is reduced from 6,189 seconds to 183.8 and 17.8 seconds on the CPU and GPU respectively. On the other hand, the TNC solver requires the most iterations to achieve convergence and the number of iterations is sensitive to the ordering of the RMSE evaluation, and hence the parallel implementation. This sensitivity explains the high
variations in timings between varying numbers of Fourier-Cosine terms and between the serial and parallel implementations for the TNC solver.

7. EXTENSION TO OTHER MODELS

In this paper we employed the Heston stochastic volatility model for all experiments. However, there are many other stochastic volatility models in use in practice, such as the Bates Jump-Diffusion model [14], the Variance Gamma model [15], or the CGMY model [16]. Details of these models are provided in Appendix B. As outlined in [8], these models can be calibrated using the same approach as presented here for Heston, but they require changes in the implementation of the ErrorFunction itself.

With the Xcelerit SDK, as the processing stages are implemented as C++ classes, other models can be implemented by use of polymorphism. That is, different concrete implementations of the IFourierCosine class in Figure 1 are provided for each model and allocated using a factory method. With this approach, demonstrated in Appendix A, new models can be added easily, benefiting from GPU acceleration and fast multi-core CPU execution as described for the Heston model above.

8. CONCLUSIONS

Financial markets change precipitously and on-demand pricing and risk models must be constantly recalibrated to reduce model risk. This paper describes the acceleration of stochastic volatility model calibration on multi-core CPUs and GPUs using the Xcelerit platform. By adopting a simple programming model, the Xcelerit platform enables the application developer to write sequential, high-level C++ code, without concern for low-level high performance computing frameworks such as OpenMP, OpenCL, CUDA, or SIMD intrinsics. This platform provides the portability, flexibility and modularity required to facilitate rapid in-house model development and productionization. Speedups of up to 29.5x and 292.8x are respectively achieved on an Intel Xeon CPU and NVIDIA Tesla K40 GPU, compared to a sequential CPU implementation. The Xcelerit platform implementation is further shown to be equivalent in performance to a low-level CUDA version. Overall, we are able to reduce the entire calibration process time of the sequential implementation from 6,189 seconds to 183.8 and 17.8 seconds on the CPU and GPU respectively without requiring the developer to reimplement in low-level high performance computing frameworks.

A. EXTENSIBLE IMPLEMENTATION

To provide extensibility in modeling capability, the flow graph should be constructed without knowledge of the model specific parameters and functionality. The abstract factory design pattern can be used to provide a generic interface to the flow graph while hiding details of the model. Following such an approach, a factory method creates the concrete model specific objects.

The following code listings are used to demonstrate the implementation of the abstract factory design pattern. In Listing 1, the interface class IFourierCosine is defined. This class specifies a contract for the child class to implement the setupParameters method. Listing 2 defines the templated child class FourierCosine which allows for run-time model specialization through the template type Model. Listing 3 defines the SVCalibration class method setModel which is the factory for the concrete actor. This method initializes the actor handle given the model’s name. The model specific calculation of the Characteristic function is not shown here but is implemented in model classes such as HestonModel and BatesModel etc.

Clearly the code provided here is just a starting point for the construction of a more extensive list of stochastic volatility models. Finally in Listing 4, the code demonstrates how the model agnostic flow graph is constructed using the actor handle produced by the factory method SVCalibration::setModel.
Listing 1: Source listing for demonstrating implementation of the interface class (abstract base class) `IFourierCosine`.

```cpp
class IFourierCosine : public xcl::Actor {
public:
  // to allow connecting any FourierCosine in flowgraph
  xcl::SeqInput<int> optIndex, termIndex;
  xcl::Output<double> result;

  // Define a pure virtual function which is called
  // by the child class in order to update parameters at
  // each optimization step
  virtual void setupParameters(const double* p) = 0;
};
```

Listing 2: Source listing for demonstrating implementation of the concrete class `FourierCosine`.

```cpp
template <typename Model>
class FourierCosine : public IFourierCosine {
public:
  Model model_; 
  
  FourierCosine(const Model& model) : model_(model) {
    registerMembers();
  }

  void setupParameters(const double* p) {
    model_.setupParameters(p);
  }

  __actor__ void run() const {
    result[0] = model_(optIndex[0], termIndex[0]);
  }
};
```
Listing 3: Source listing for demonstrating how the factory method constructs the concrete SV model objects.

```cpp
void SVCalibration::setModel(const std::string& model)
{
    if (a_ != NULL)
        delete a_;  
    model_ = model;

    // Initialize the actor based on the model
    if (model == "Heston")
    {
        // Create a Heston model object and initialize it
        HestonModel mod(T_, K_, Types_, s0_, r0_, q0_, numOptions_, bIncNLCnstr_);
        // Create actor (makes a copy of mod)
        a_ = new FourierCosine<HestonModel>(mod);
        param_size = 5;
    }
    elseif (model == Bates)
    {
        // Create a Bates model object and initialize it
        BatesModel mod(T_, K_, Types_, s0_, r0_, q0_, numOptions_, bIncNLCnstr_);
        // Create actor (makes a copy of mod)
        a_ = new FourierCosine<BatesModel>(mod);
        param_size = 8;
    }
    else
    {
        throw std::runtime_error("Unsupported model");
    }
}
```

Listing 4: Source listing for demonstrating how to construct the flow graph for calculating the RMSE.

```cpp
double SVCalibration::error_func(const double* p0, int length)
{
    assert(length >= param_size);
    // Adjust parameters in the actor
    a_->setupParameters(p0);

    // Construct actors and flowgraph
    xcl::SequenceSource<int,2> src(xcl::Range<int>(0, numOptions_),
                                  xcl::Range<int>(0, numTerms_));
    // Reduce across series terms
    xcl::PartialSumReduce<double> partialSumReduce(numTerms_);
    xcl::Flowgraph f;
    f += src.output1 >> a_->optIndex,
        src.output2 >> a_->termIndex,
        a_->result >> partialSumReduce,
        partialSumReduce >> errorSink_->_V_hat;
    // Call the run method of the flowgraph to compute MSE.
    f.run();
    double mse = errorSink_->_getResult();
    // Return the RMSE
    return sqrt(mse/numOptions_);
}
```
B. STOCHASTIC VOLATILITY MODEL DESCRIPTIONS

The stochastic volatility models also implemented are briefly listed here. This section is provided to briefly describe the model parameters and is intended as a useful reference for other stochastic volatility models.

B.1. Bates Model

The Bates Jump-Diffusion model [14] is specified as the following set of coupled stochastic differential equations

\[
\frac{dS_t}{S_t} = \mu dt + \sqrt{V_t} dW_1^t + (Y - 1)S_t dN_t, \\
\frac{dV_t}{V_t} = \kappa (\theta - V_t) dt + \sigma \sqrt{V_t} dW_2^t,
\]

(7)  
(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 \(W_1^t\) and \(W_2^t\). A key characteristic of the model, which originates from the embedded Heston stochastic volatility diffusion model, is that the Wiener processes are correlated \(dW_1^t \cdot dW_2^t = \rho dt\). 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.

B.2. Variance Gamma Model

Following [16], 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 \(L\):

\[
S(t) = S(0) \exp(m t + L(t; \theta, \nu, \sigma) + \omega(t)), 
\]

(9)

where \(m\) is the mean rate of return on the stock under the statistical probability measure and the Martingale correction term is \(\omega(t) = \frac{1}{\nu} \ln(1 - \theta \nu - \sigma^2 \nu^2/2)\). The parameters \(\theta, \nu, \sigma\) only indirectly reflect the skewness and kurtosis of the return distribution. \(\theta\) by itself determines the overall scale of the volatility. The form of the characteristic function is provided in [16].

B.3. CGMY Model

The CGMY model [17] 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

\[
C = \frac{1}{\nu}, \\
G = \left( \frac{\nu^2 \nu^2 + \sigma^2 \nu^2}{4} - \frac{\theta \nu}{2} \right)^{-1}, \\
M = \left( \frac{\nu^2 \nu^2 + \sigma^2 \nu^2}{4} + \frac{\theta \nu}{2} \right)^{-1}
\]

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 [17].
ACKNOWLEDGEMENT

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