GPU Acceleration for Computational Finance

Chuan-Hsiang Han[1]

and

Shi-Ti Yu[2]

Abstract:

Recent progress of graphics processing unit (GPU) computing with applications in science and technology has demonstrated tremendous impact over the last decade. However, financial applications by GPU computing are less discussed and may cause an obstacle toward the development of financial technology, an emerging and disruptive field focusing on the efficiency improvement of our current financial system. This paper aims to raise the attention of GPU computing in finance by first empirically investigate the performance of three basic computational methods including solving a linear system, Fast Fourier transform, and Monte Carlo simulation. Then a fast calibration of the wing model to implied volatilities is explored with a set of traded futures and option data in high frequency. At least 60% executing time reduction on this calibration is obtained under the Matlab computational environment. This finding enables the disclosure of an instant market change so that a real-time surveillance for financial markets can be established for either trading or risk management purpose.

[1] Department of Quantitative Finance, National Tsing-Hua University, Hsinchu, Taiwan 30013, R.O.C., Email: chhan@mx.nthu.edu.tw. Work Supported by NSC 102-2115-M-007-002-
Acknowledgment: Nvidia-NTHU Joint Lab on Computational Finance

[2] Department of Quantitative Finance, National Tsing-Hua University, Hsinchu, Taiwan 30013, R.O.C., Email: styu@mx.nthu.edu.tw

Section 1: Introduction

The central processing unit (CPU) contains multiple and powerful cores. Each CPU core is optimally designed for serial processing. In contrast, the graphic processing unit (GPU) may consist of hundreds or thousands of cores. These cores are highly structured for parallel processing. See a pictorial comparison of the structure between CPU and GPU in Figure 1.

In comparison to the longer history of CPU, GPU is a new and revolutionary device to accelerate computational performance. First manufactured by Nvidia in 1999, GPU was designed for computer graphics rendering. After Nvidia launched GPU's low-end programming language - Compute Unified Device Architecture (CUDA) in 2006, scientists and engineers have found that many heavy computational tasks can be significantly improved by GPUs.

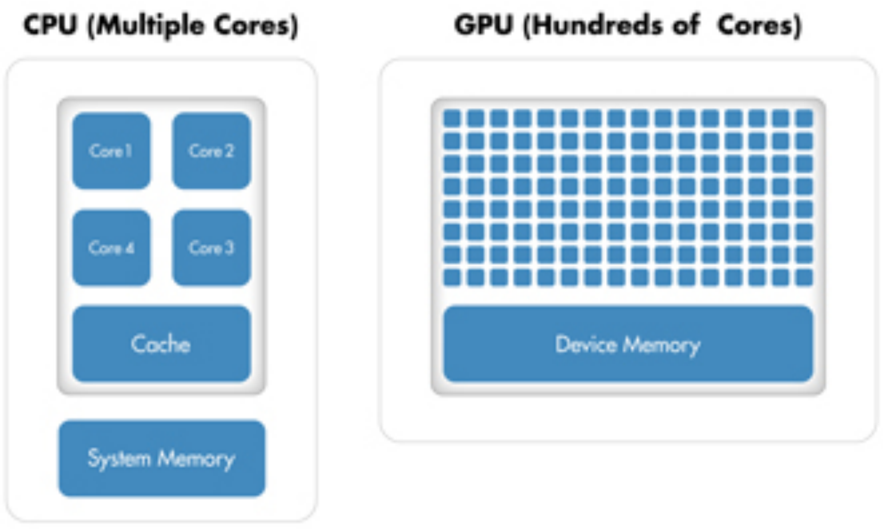


Figure 1. Comparison of the number of cores on a CPU system and a GPU system. (Resource: Nvidia)

GPU computing refers to the use of CPU together with GPU for fast computation by offloading parallel portions of the numerical algorithm to the GPU, while serial portions of the algorithm to the CPU. When a computational task is massively paralleled, the cooperative GPU computing may become an accelerator to solely CPU computing subjected to memory access to passing messages. That is, the time spend on transferring data between the CPU system memory and the GPU shared memory is crucial to the efficiency of GPU computing. Thus, GPU computing can be of high performance when numerical algorithms satisfy two criteria:

(1) massive parallelization – a large number of instructions can be executed (upon many sets of data) independently.

(2) Memory accessibility – the overall computational speedup is subjected to the amount of data transfer between CPU system memory and GPU shared memory because memory access is slow.

In addition to the nature of algorithms, writing computer programs in CUDA can still be challenging and it often requires a fine-tuning procedure to optimize numerical performance for specific applications and GPU configuration. Professional developers are indeed able to gain extraordinary speedup using CUDA[3] codes for their GPU computing.

The structure of this paper is the following. In Section 2, we adopt the commercial software Matlab as a tested computational environment[4]. The general concept of Matlab GPU computing is introduced. In Section 3, Matlab GPU commands are illustrated and explained on how they can easily improve typical numerical examples from computational finance. These examples include two deterministic schemes such as a linear solver and the fast Fourier transform, and one stochastic scheme - Monte Carlo simulations. In Section 4, a real-time calculation of implied volatilities and the wing model calibration are implemented for a set of traded futures and option prices in high frequency from TAIFEX[5]. Numerical comparisons between CPU and GPU computing are provided. Lastly we make a conclusion.

[3] A good source for GPU computing by CUDA can be found on http://www.nvidia.com/object/cuda_home_new.html

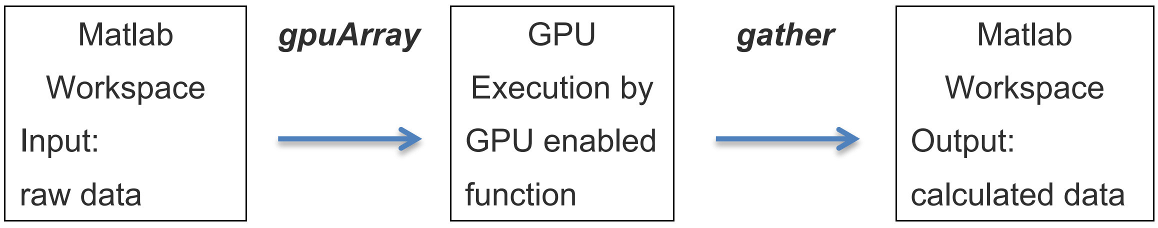
[4] Other public open-sourced platforms such as R or Python can also be used for GPU computing.

[5] TAIFEX is an abbreviation of Taiwan Futures Exchange.

Section 2: GPU Computing in Matlab

In 2010, the feature of GPU computing was added into Matlab's parallel computing toolbox by a joint force of Mathworks and Nvidia. Build-in GPU enabled functions[6] allow developers to take advantage on the powerful GPU computing simply by Matlab, a high-end programming language. There already exist many successful application areas in science and engineering using this computing framework[7].

When Matlab's GPU enabled functions are executed on the GPU, data must be transferred from Matlab workspace to GPU device memory. The command ***gpuArray*** provides a specific array type for such data transfer, then GPU enabled functions can process these data. The command ***gather*** returns those calculated results, which are stored in GPU, back to Matlab workspace. The procedure of GPU computing in Maltab is as follows:


Matlab Workspace Input: raw data
gpuArray
GPU Execution by GPU enabled function
gather
Matlab Workspace Output: calculated data

Note that when input raw data is large, for example a large matrix, users need to check whether these data exceed GPU's memory limit or not. By running

[6] A number of Build-in Matlab GPU enabled functions can be found on http://www.mathworks.com/help/distcomp/establish-arrays-on-a-gpu.html and http://www.mathworks.com/help/distcomp/run-built-in-functions-on-a-gpu.html

[7] A good source to learn about Matlab GPU computing can be found on http://www.mathworks.com/discovery/matlab-gpu.html

gpuDevice, information such as name, total memory, and available memory can be retrieved from the GPU device.

The advantage of Matlab GPU programming is that users can easily utilize GPU computing by adding just few more commands to their original Matlab codes. Disadvantages include that only limited Matlab functions are GPU-enabled and the computing efficiency of Matlab GPU is less than those codes written in CUDA. When one wants to execute a whole function on GPU, the Matlab function ***arrayfun***[8] is designed for this purpose.

In next section, several standard examples arising form computational finance are discussed. Only gpyArray is demonstrated for a crystal comparison between Matlab CPU commands and GPU commands.

Section 3: Basic Examples from Computational Finance

Three popular computational methods used in quantitative finance include but not limited to (1) a linear solver to the numerical partial differential equation, (2) Fourier transform method, and (3) Monte Carlo simulations. Next we demonstrate Matlab GPU computing on examples associated with these methods and their speedup performance over Matlab CPU computing.

Example 1: Solving a linear equation

Solutions of linear equations often represent the first order approximations to many problems. In computational finance, linear equations may emerge from numerical partial differential solutions (PDEs), optimization, regression, etc. Here we specifically address the method of numerical PDEs (Lamberton and Lapeyre (2011)). According to stochastic financial theory, prices of some financial derivatives can be described by solutions of PDEs.

[8] Its usage can be found on http://www.mathworks.com/help/distcomp/arrayfun.html

The implicit finite difference scheme is known as an accurate and stable method for solving pricing PDEs. This scheme induces linear equations with certain structure. Solutions of those linear equations are discrete approximation to solutions of the corresponding pricing PDEs.

Given an invertible matrix A and a vector b with the same dimension, the solution of linear equation Ax=b can be obtained by this command line

```
>>x=A\b;            % on CPU
```

in Matlab and this computation is executed by CPU. To take advantage of GPU computing in Matlab, users only have to create GPUArrays by transferring matrix A and vector b to GPUs but still use the same command line like in CPU. Here is what Matlab user can do for solving the linear equation in GPU:

```
>> gA = gpuArray(A); gb = gpuArray(b); gx=gA\gb;       %on GPU
>>x=gather(gx)     %on CPU
```

It shows that by simply change of the array type of inputs MATLAB users are able to implement GPU computing for their applications.

Figure 2 demonstrates speedup performance of GPU computing over traditional CPU computing given the same random matrix A and random vector b with various dimensions shown on the x-axis. When the dimension n=1000, we set

```
>>n=1000;
>> A = rand(n); b = rand(n, 1);
```

Then one can use the previous command lines to solve for the linear equation by either CPU computing or GPU computing. We record their computing times. Speedup ratios on various dimensions from n=1000 to 8000 are shown below.

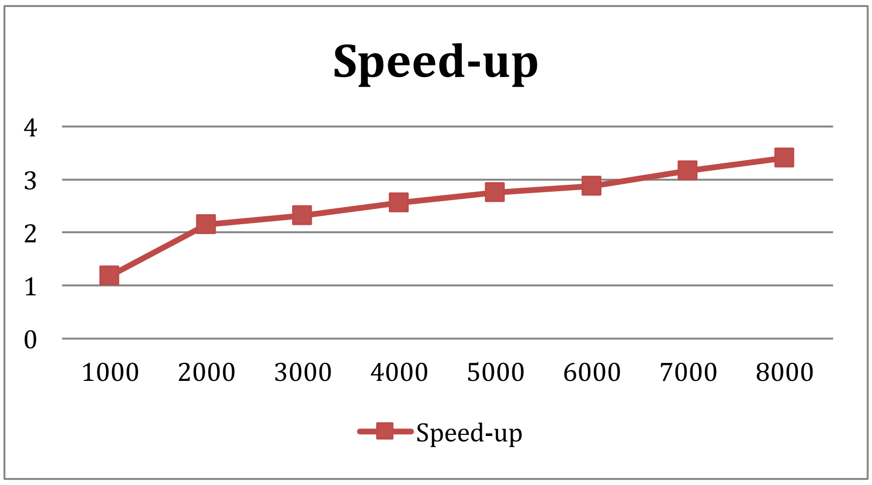


Figure 2: GPU Speedup performance when the random vector size n ranges from 1000 to 8000.

Example 2: Fast Fourier Transform

Fourier transform method can be used to characterize option prices under various financial models (Kienitz and Wetterau (2012)). Fast Fourier transform (FFT) is applied and becomes a major computational method in finance.

Let's introduce basic instructions on CPU and GPU both in Matlab by considering a random vector with the size n and its FFT. Next line shows how Matlab CPU is programmed:

```
>>n=2^16; T = rand(n,1); F = fft(T);   %on CPU
```

To perform the same operation on the GPU, recall that one has to use the command gpuArray to transfer data from the MATLAB workspace to GPU device memory. Then fft, a GPU enabled function, can be executed. One can use the command gather to transfer the result stored on GPU back to CPU for further serial operation.

```
>> gT = gpuArray(T); gF = fft(gT);      %on GPU

>> F=gather(gF);   %on CPU
```

It is worth noting that in this particular example the running time of FFT on GPU might be less than the time to transfer data between CPU and GPU. This means data transfer can possibly degrade the whole performance on GPU computing.

Figure 3 shows the speedup performance for pricing European options under Heston model by FFT (Carr and Madan (1999)). A comparison of execution times for CPU and GPU computing is implemented under a CPU (Intel Core i5-3230M Processor, 2.6 GHz) and a GPU (Geforce GT635M). It is clear to see that when the discretization size n, shown on the x-axis, is small, the overall performance of GPU is worse than CPU.

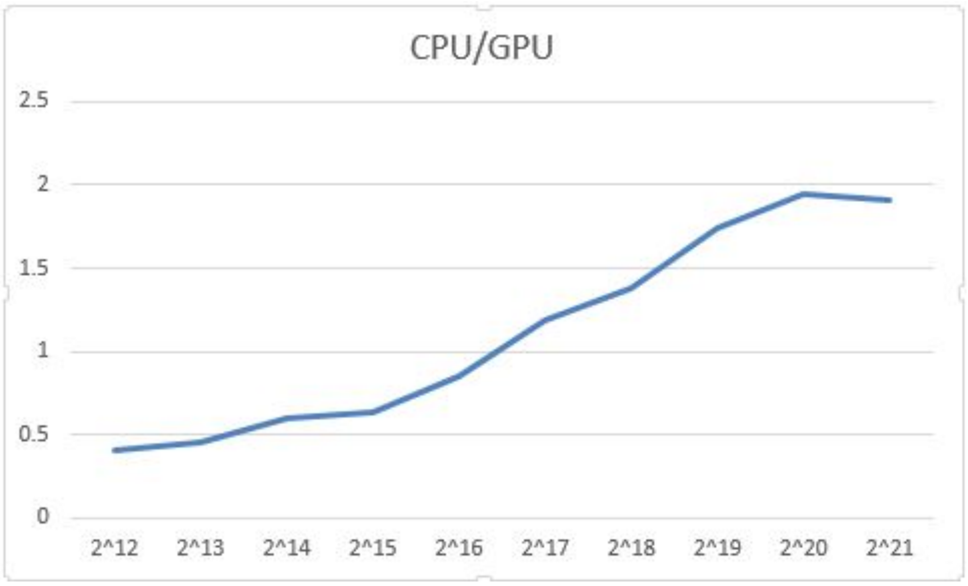


Figure 3: GPU Speedup performance when the discretization size n ranges from 2^12 to 2^21.

Example 3: Monte Carlo Simulation

The previous two examples, solving a linear equation and FFT, involve deterministic numerical methods. In this section stochastic computation, namely Monte Carlo simulation, is considered.

Basic Monte Carlo method calculates the arithmetic average of a large number of random samples drawn from independently identical distributions. Its independent property of large samples fits well to the massive parallelization criteria of GPU computing. It is often to see huge numerical performance by GPU computing on Monte Carlo simulation.

Two case studies are conducted for running Monte Carlo simulation on both CPU and GPU computing. The first case concerns an estimation problem for joint default probability, associated with a risk problem. The second example concerns about the vanilla option pricing problem, associated with a pricing problem. Both the risk and pricing problems constitute a large component in the field of quantitative finance.

Case 1: Estimating Joint Default Probability under Multivariate Normal

We consider the estimation of joint default probability:

$$p = E\left[I\left(\vec{X} < \vec{D}\right)\right] = E[\prod_{i=1}^{n} I(X_i < D_i)],$$

in which the defaultable asset vector $\vec{X} = (X_1, X_2, \dots, X_n)' \in \mathcal{R}^{n\times 1}$ is assumed centered normally distributed $\vec{X} \sim \mathcal{N}\left(\vec{0}, \Sigma\right)$ with dimension n and its default threshold vector is denoted by $\vec{D} = (D_1, D_2, \dots, D_n)' \in \mathcal{R}^{n\times 1}$. For simplicity, in the Matlab experiment below we further assume that $\vec{D} = d\times \underbrace{(1, \dots 1)'}_{n\times 1}$. More general distributions and relevant (efficient) importance sampling can be found on author's work.

Matlab codes for this case study can be found below for CPU and GPU computing.

```
%Parameters and variables
d=-1; rho=0.25;
n=5;

Nrepl = 750000;          %total number of simulation
```

Matlab CPU Computing	**Matlab GPU Computing**

Sigma=rho*ones(n,n)
+(1-rho)*eye(n);
T = chol(Sigma);
X_MC = randn(Nrepl,size(T,1))
* T;
MC = prod(1*(X_MC <
d*ones(Nrepl,n)),2);
P_MC = mean(MC);
SE_MC = std(MC) / sqrt(Nrepl);
	Sigma=rho*gpuArray.ones(n,n)
	+(1-rho)*gpuArray.eye(n);
	T = chol(Sigma);
	X_MC = gpuArray.randn(Nrepl,size(T,1))
	* T;
	MC = prod(1*(X_MC <
	d*gpuArray.ones(Nrepl,n)),2);
	P_MC_g = mean(MC);
	SE_MC_g = std(MC) / sqrt(Nrepl);
	P_MC = gather(P_MC_g);
	SE_MC = gather(SE_MC_g);

Figure 4 shows the speedup performance for this case study over various dimensions. GPU computing performs efficiently than CPU computing. We should remark that Maltab's command mvncdf.m provides the same calculation but it is not a GPU enabled function. This function is limited to dimension 25 but not our Monte Carlo simulation shown above.

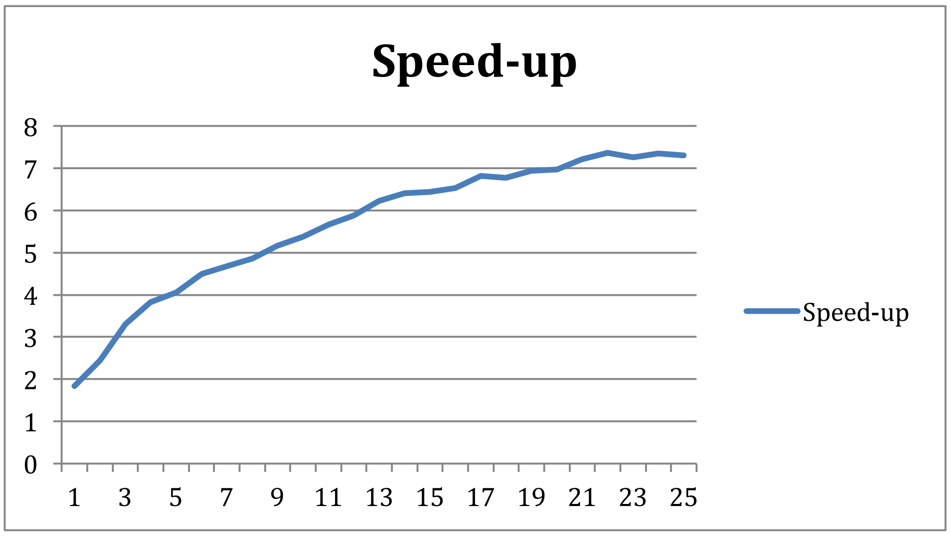


Figure 4: GPU Speedup performance when the dimension size ranges from 1 to 25.

Case 2: European Option Pricing by the Basic Monte Carlo Method

Consider the estimation of an European call option price: $p = E[e^{-rT}(S_T - K)^+ | S_0 = S0]$. The stock price process S_t is governed by the Black-Scholes model: $dS_t = rS_t dt + \sigma S_t dW_t$ with the initial price $S_0 = S0$. Parameters r and σ denote the risk-free interest rate and the volatility, respectively. Variables T and K denotes the time to maturity and the strike price of the European option, respectively.

The basic Monte Carlo (BMC) estimation under GPU computing is shown below.

```
NSteps=100;         %time domain discretization size for geometric Brownian
motion
Nrepl=100000;       %number of simulation for Monte Carlo

%model parameters and variables
T=1;        %time to maturity
r=0.05;         %risk-free interest rate
sigma=0.3;          %volatility
S0=50;          %initial stock price
K=55;           %call strike price

% stock price simulations
dt=T/NSteps;
nudt=(r-0.5*sigma^2)*dt;
sqdt=sqrt(dt);
sidt=sigma*sqdt;
RandMat=gpuArray.randn(Nrepl, NSteps);
Increments=[nudt+sidt*RandMat];
LogPaths=cumsum([log(S0)*gpuArray.ones(Nrepl,1), Increments],2);
SPaths=exp(LogPaths);

%samples of European call payoff
SPaths(:,1)=[];       %get ride of starting points
CashFlows=exp(-r*T).*max(0,SPaths(:,NSteps)-K);         %samples of discounted
payoffs

%calculate sample mean and standard error
price=mean(CashFlows)             %sample mean
```

```
var=cov(CashFlows)/Nrepl;
std=sqrt(var);              %standard error
```

Removing gpuArray from those red marked commands induces the CPU programming. An additional variance reduction technique termed martingale control variate (MCV) method (Fouque and Han (2007)), though details are skipped here, can be applied to dramatically increase the accuracy of estimation. However MCV takes more time to compute than BMC. The combination of MCV with GPU shows a great potential to increase the accuracy (MCV vs. BMC) and reduce the computing time (GPU vs. CPU).

Table 1 records numerical performance and runtimes under different estimation methods: BMC and MCV under different computing framework: CPU and GPU. It can be observed that the combination of MCV algorithm with GPU computing performs best. The run time of MCV on GPU is about the run time of BMC on CPU but the former is much accurate than the later. This can be understood by "standard error reduction" from the reduced variance by MCV and enlarged sample size by GPU.

Table 1: Execution time and numerical results of CPU and GPU Computing. Numerics in parenthesis indicate standard errors. Hardware Configuration: CPU: Core i7 950 (4-core 3.06 GHz), GPU: NVIDIA GeForce GTX 690 (3072 CUDA core, 915 MHz)

	BMC	Time	MCV	Time
CPU	168.9960 (2.9736)	0.125410 s	167.2062 (0.1281)	0.405542 s
GPU	167.8563 (3.0303)	0.090116 s	167.1935 (0.1280)	0.185130 s

A similar conclusion under a more complex stochastic volatility model can be found in Han and Lin (2014), in which CUDA was implemented for GPU. The synergy of an efficient numerical algorithm and a parallel computing structure demonstrate a dramatic computational performance.

Section 4: Wing Model Calibration to Implied Volatilities

The implied volatility of an option price is a quantity that uniquely solves a one-dimensional nonlinear root-finding problem. It is defined by the Black-Scholes type formula (Hull (2010)) in which all model parameters, except that the volatility, and the corresponding option price are assumed fixed. Once a set of implied volatilities with the same expiration date is solved, a graph of these volatilities as a function of strike prices becomes a curve, known as the implied volatility curve. A wing model provides a parametric approach to best fit an implied volatility curve. A simple form of the wing model utilizes a quadratic polynomial as a function of strike prices to regress implied volatilities.

In contrast to other some model calibration models such as stochastic volatility or jump diffusion (Kienitz and Wetterau (2012)), the wing model is less sophisticated, lack of dynamic consistency, and probably less precise but it can be calculated in a rather short time. Hence the wing model remains a popular approach for traders and risk controller to mark to the market. It can be used for real-time surveillance to watch over intraday change of financial dynamics. One application among many is to establish a price stability system for futures and options exchange (Han (2017)).

With the drastic increase of trading frequency in major exchanges for futures and options, there is a need for fast calculation of wing model calibration to implied volatility curves in order to retrieve the real-time market information. The GPU computing is equipped to accelerate the whole calibration process in two aspects. They include (1) fast calculation of implied volatilities, and (2) regression with a quadratic polynomial. The reason for the first aspect is that implied volatilities can be calculated independently. This property is a good match to the parallel structure of GPU. As for the second aspect, it turns out the Matlab function for regression is granted as a GPU enabled function. In virtue of advantages of these two aspects, one can expect an increase of efficiency by using GPU computing.

Tested data includes all near month call prices of Taiwan Index Option (TXO) traded in TAIFEX on 8/25/2015 and their associated index futures price (TX) with the same expiration as the underlying. Those asset prices are given in every second. (The annualized riskless rate (r) is given as 0.5%.) Data liquidity is a typical concern, so deep out-of-the-money options whose absolute value of delta is larger than 0.1 are excluded. We compare execution time calculated by CPU and GPU, and observe whether GPU indeed can drastically reduce computation time.

As mentioned in Section 2, under the Matlab experimental environment, time series data of futures and option's prices have to be relocated to GPUarray[9]. Since the typical Matlab solver blsimpv.m for implied volatility calculation is not a GPU enabled function, the bisection[10] method is introduced to calculate implied volatilities. As for the wing model calibration to implied volatility curves, if the number of updated implied volatility is less than 6, we use the previously estimated parameters[11] of wing model.

Two implied volatility curves with their fitted wing models are shown in Figure 5. These are results calibrated at two different time points, i.e. 11:10:26 and 11:10:57 respectively.

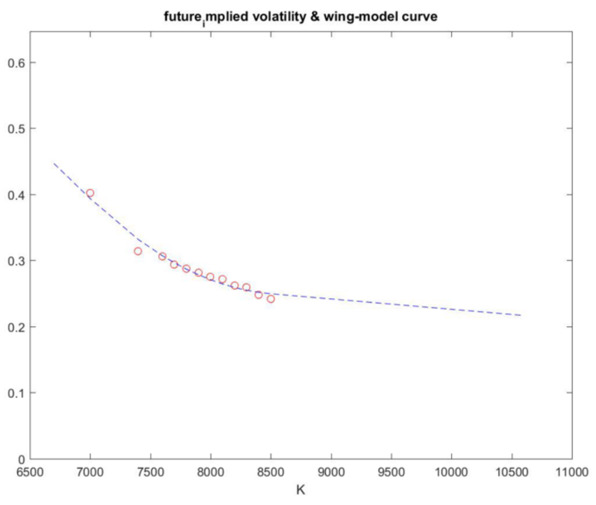
[9] Stock = gpuArray(stock)； call_data = gpuArray(call_data)

[10] If we use Newton's method instead, we will gain more significant effect of acceleration. However, the bisection method is more stable.

[11] The function polyfit in matlab can fit the data in quadratic form and it is also GPU-enabled function which can directly be used by GPU to accelerate.

11:10:26 tangency extension

11:10:57 tangency extension



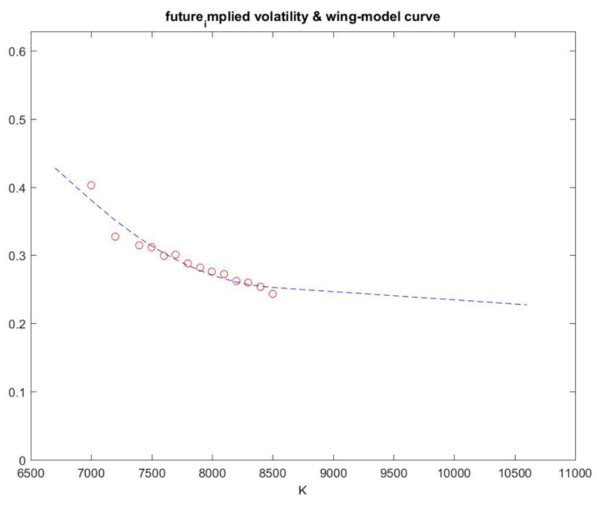


Figure 5: Wing model calibration to implied volatility of TXO data.

In our dataset on 8/25/2015, there are more than 120,000 call TXO option prices in total. Our task is to fit the wing-model every second, so it is about 18,000 times per trading day in TAIFEX. Table 2 recorded Below is the table of GPU parallel computation time before and after acceleration.

Table 2: Comparisons of execution time, numerical results, and hardware configuration of CPU and GPU Computing on implied volatility calculation and the wing model calibration.

	CPU Time	**GPU Time**	**Time Reduction**
Implied Vol Calculation	14.78 seconds in total (0.123ms per option price)	0.53 seconds in total (0.004ms per option price)	96.06%
Wing Model Calibration	875.34 seconds in total (49ms each calibration)	352.82 seconds in total (20ms each calibration)	60.35%
Hardware model	CPU: Intel®Core(TM) i5-6300HQ	GPU: NVIDIA GeForceGTX950M (Total number of cores: 640)	
Software	Matlab	Matlab	

From this table, it is observed that on average for each wing model calibration, 49ms (0.049s = 875.34/18,000 times) and merely 20ms (0.02s = 352.82/18,000 times) are needed on traditional CPU computing and GPU computing frameworks, respectively. The effect of GPU parallel acceleration for fitting implied volatility curve by using the wing model is significant. It reduces about 60% computing time. Since the performance of Matlab on GPU is less than CUDA, we can expect to gain more efficiency if CUDA is implemented.

Conclusion

Engineers, scientists, and financial quants have been successfully employing GPU technology for their domain applications. However GPU's programming language CUDA is low-level and it requires technical knowledge about hardware device. With minimal effort, Matlab users can take advantage of the promising power of GPUs by using gpuArrays and GPU enabled functions to speed up MATLAB operations. We illustrate several typical examples from computational finance and find that Matlab GPU computing can be beneficial. Hand-on codes for Matlab CPU and GPU are provided for comparisons. Moreover, a fast calibration of the wing model to implied volatilities in high frequency is provided. It can play a crucial role on establishing a price stability system for option trading exchanges.

References:

P. Carr and D. Madan. Option evaluation using the fast Fourier transform. Journal of Computational Finance, 24:61-73,1999

J. P. Fouque and C.H. Han. A martingale control variate method for option pricing with stochastic volatility. ESAIM Probability & Statistics 11, 40-54, 2007.

C.H. Han. Research on Instantaneous Pricing Models of Equity Futures and Options in High Frequency. Technical Report (in Chinese). TAIFEX. 2017.

C.H. Han and Y.-T. Lin. Accelerated Variance Reduction Methods on GPU. Proceedings of the 20th IEEE International Conference on Parallel and Distributed Systems, 2014.

J. Hull. Options, futures, and other derivatives, 8th Edition, Pearson/Prentice Hall, 2011.

D. Lamberton and B. Lapeyre. Introduction to Stochastic Calculus Applied to Finance. Second Edition. Chapman Hall/CRC, 2007.

J. Kienitz and D. Wetterau. Financial Modeling: Theory, Implementation and Practice with Matlab source. Wiley Finance. 2012.