Step by step implementation and optimization of simulations in quantitative finance

Lokman Abbas-Turki

UPMC, LPMA

10-12 October 2017
Plan

Introduction

Local volatility challenges GPUs

Monte Carlo and OpenACC parallelization

Monte Carlo and CUDA parallelization

PDE formulation & Crank-Nicolson scheme

PDE simulation and CUDA parallelization
Plan

Introduction

Local volatility challenges GPUs

Monte Carlo and OpenACC parallelization

Monte Carlo and CUDA parallelization

PDE formulation & Crank-Nicolson scheme

PDE simulation and CUDA parallelization
Introduction

What make bankers change their mind?

- The CVA (Credit Valuation Adjustment) or XVA (X=C, D, F, K, M Valuation Adjustment) are tipping point applications,
- FRTB (Fundamental Review of the Trading Book) is the other important application with a deadline in 2019.
- Electronic trading and deep learning.
- Lloyds Blankfein declared about Goldman Sachs: “We are a technology firm”.

HPC in banks

- From distribution to parallelization.
- From small to big nodes.
- The efficiency of GPUs becomes undeniable.
- Use the .net C, C++ and C#.

Remaining challenges and fears

- Code management.
- Possible conflicts within quant teams.
- Can we extend the results for toy models to more general models?
Plan

Introduction

Local volatility challenges GPUs

Monte Carlo and OpenACC parallelization

Monte Carlo and CUDA parallelization

PDE formulation & Crank-Nicolson scheme

PDE simulation and CUDA parallelization
Local volatility from implied volatility

First array: \( r_g \)

\[
dS_t = S_t r_g(t) dt + S_t \sigma_{loc}(S_t, t) dW_t, \quad S_0 = x_0.
\]

- \( S \) is the stock price process where \( x_0 \) is the spot price
- \( W \) is a Brownian motion with \( W_0 = 0 \)
- \( r_g \) is the risk-free rate, assumed piecewise constant
- \( \sigma_{loc}(x, t) \) is a local volatility function: \( \mathbb{R}^* \times \mathbb{R}_+ \rightarrow \mathbb{R}_+ \)

Dupire Equation

Given a family \( C(K, T)_{K,T} \) of call prices with strike \( K \) and maturity \( T \)

\[
\sigma^2_{loc}(K, T) = 2 \frac{\partial C/\partial T + K r_g(T) \partial C/\partial K}{K^2 (\partial^2 C/\partial K^2)}
\]

From implied to local

Using the Black & Scholes implied volatility \( \sigma^{imp}(x, t) \), Andersen and Brotherton-Ratcliffe (1997) showed that

\[
\sigma^2_{loc}(x, t) = \frac{2 \frac{\partial \sigma^{imp}}{\partial t} + \sigma^{imp} \frac{t}{x} + 2x r_g(t) \frac{\partial \sigma^{imp}}{\partial x}}{x^2 \left[ \frac{\partial^2 \sigma^{imp}}{\partial x^2} - d_+ \sqrt{t} \left( \frac{\partial \sigma^{imp}}{\partial x} \right)^2 + \frac{1}{\sigma^{imp}} \left( \frac{1}{x \sqrt{t}} + d_+ \frac{\partial \sigma^{imp}}{\partial x} \right) \right]^2}
\]

with \( d_+ = \frac{1}{2} \sigma^{imp} \sqrt{t} + \left[ \log(x_0/x) + \int_0^t r_g(u) du \right] / \left[ \sigma^{imp} \sqrt{t} \right] \).
SVI to simulate implied volatility

+2 arrays: $T_g$, $K_g$

For $(x, t) \in K_g \times T_g$, we assume that the observed implied volatility

$$\sigma_{imp}(x, t) = \sqrt{\frac{w(x, t)}{t}}$$

where the cumulative variance is parameterized by

$$w(x, t) = \frac{\theta_t}{2} \left[ \frac{1 + \rho \varphi(\theta_t) \left( \log \left( \frac{x}{x_0} \right) - \int_0^t r_g(u)du \right) + \sqrt{\left[ \varphi(\theta_t) \left( \log \left( \frac{x}{x_0} \right) - \int_0^t r_g(u)du \right) + \rho \right]^2 + (1 - \rho^2)}}{\eta \theta_t^\gamma (1 + \theta_t)^{1-\gamma}} \right]$$

with $\varphi(\theta_t) = \frac{\eta}{\theta_t^\gamma (1 + \theta_t)^{1-\gamma}}$ and $\theta_t = a^2 \left( t + b(1 - e^{-\lambda t}) \right)$.

All parameters of (1) are discussed in Gatheral & Jacquier (2013)

Interpolation on $[0, \max(K_g)] \times [0, \max(T_g)]$

We compute some values of $\sigma_{imp}(x, t)$ on the grid $\in K_g \times T_g$. Then, we interpolate these values to obtain $\tilde{\sigma}(x, t)$ defined on $[0, \max(K_g)] \times [0, \max(T_g)]$ and we assume

$$\sigma_{imp}(x, t) \approx \tilde{\sigma}(x, t) \text{ when } (x, t) \in [0, \max(K_g)] \times [0, \max(T_g)]$$
Local volatility challenges GPUs

Bicubic interpolation for implied volatility

Let $k, q$ with $(x, t) \in \mathbb{K}_g[k], \mathbb{K}_g[k+1] \times \mathbb{T}_g[q], \mathbb{T}_g[q+1]$

$$\bar{\sigma}(x, t) = \sum_{i=0}^{3} \sum_{j=0}^{3} C_g(k, q, i, j) l^i u^j$$

where $l = \frac{t - T_g[q]}{T_g[q+1] - T_g[q]}$ and $u = \frac{x - K_g[k]}{K_g[k+1] - K_g[k]}$

Fourth array: $C_g$

$C_g(k, q, i, j) = C_g[k * (n_t - 1) * 16 + q * 16 + i * 4 + j]$ with $(k, q, i, j) \in \{0, ..., n_k - 1\} \times \{0, ..., n_t - 1\} \times \{0, ..., 3\} \times \{0, ..., 3\}$ and $n_k, n_t$ are the size of $\mathbb{K}_g$ and $\mathbb{T}_g$

Approximated local volatility

$$\sigma_{loc}^2(x, t) \approx \min(\max(\tilde{\sigma}^2(x, t), 0.0001), 0.5)$$

where $\tilde{\sigma}^2(x, t) = \frac{2 \frac{\partial \tilde{\sigma}}{\partial t} + \frac{\tilde{\sigma}}{t} + 2 x r_g(t) \frac{\partial \tilde{\sigma}}{\partial x}}{x^2 \left[ \frac{\partial^2 \tilde{\sigma}}{\partial x^2} - d_+ \sqrt{t} \left( \frac{\partial \tilde{\sigma}}{\partial x} \right)^2 + \frac{1}{\tilde{\sigma}} \left( \frac{1}{x \sqrt{t}} + d_+ \frac{\partial \tilde{\sigma}}{\partial x} \right)^2 \right]}$

with $d_+ = \frac{1}{2} \tilde{\sigma} \sqrt{t} + \left[ \log(x_0/x) + \int_{0}^{t} r_g(u) du \right] / [\tilde{\sigma} \sqrt{t}]$
Plan

Introduction

Local volatility challenges GPUs

Monte Carlo and OpenACC parallelization

Monte Carlo and CUDA parallelization

PDE formulation & Crank-Nicolson scheme

PDE simulation and CUDA parallelization
Pricing bullet option with Monte Carlo (MC)

Price for \( t \in [0, T] \)

\[
F_t = e^{-\int_t^T r_g(u)du} E \left( (S_T - K) + 1\{I_T \in [P_1, P_2]\} \mid \mathcal{F}_t \right) \text{ with } I_t = \sum_{T_i \leq t} 1\{S_{T_i} < B\}
\]

- \( K, T \) are respectively the contract’s strike and maturity
- \( 0 < T_1 < \ldots < T_M < T \) is a predetermined schedule
- The barrier \( B \) should be crossed \( I_T \) times with \( \{P_1, \ldots, P_2\} \subset \{0, \ldots, M\} \)

\((S_t, I_t)\) is Markov

\[
F(t, x, j) = e^{-\int_t^T r_g(u)du} E(X \mid S_t = x, I_t = j), \quad X = (S_T - K) + 1\{I_T \in [P_1, P_2]\}
\]

MC procedure

Simulate \( F(0, x_0, 0) = E(X) \) using a family \( \{X_i\}_{i \leq n} \) of i.i.d \~\( X \)

- Strong law of large numbers:

\[
P \left( \lim_{n \to +\infty} \frac{X_1 + X_2 + \ldots + X_n}{n} = E(X) \right) = 1
\]

- Central limit theorem: Denoting \( \epsilon_n = E(X) - \frac{X_1 + X_2 + \ldots + X_n}{n} \)

\[
\frac{\sqrt{n}}{\sigma} \epsilon_n \to G \sim \mathcal{N}(0, 1), \sigma \text{ is the standard deviation of } X
\]

- There is a 95% chance of having: \( \epsilon_n \leq 1.96 \frac{\sigma}{\sqrt{n}} \)
Discritization set $0 = t_0 < t_1 < \ldots < t_{N_t} = T$ finer than $0 < T_1 < \ldots < T_M < T$ with $\delta t = \sqrt{t_{k+1} - t_k}$

Iterating for path-dependant contract ($x_0 = 50$)

For each $t_k$, $k = 0, \ldots, N_t - 1$:

1. Random number generation (RNG) of independent Normal variables $G_i$
2. Stock price actualization $S^i_{t_{k+1}} = S^i_{t_k} (1 + r_g(t_k) \delta t \delta t + \sigma_{loc}(S^i_{t_k}, t_k) \delta t G_i)$
3. If $t_{k+1} = T_l$ with $l \in \{1, \ldots, M\}$, $l^i_{T_l} = l^i_{T_{l-1}} + (S^i_{T_l} < B)$

At $t_{N_t}$ Compute the payoff $X^i$ then average
P. L’Ecuyer CMRG on GPU to generate uniformly distributed random variables

General Form of linear RNGs

Without loss of generality:

\[ X_n = (AX_{n-1} + C) \mod(m) = (A : C) \begin{pmatrix} X_{n-1} \\ \vdots \\ 1 \end{pmatrix} \mod(m) \]  

(3)

Parallel-RNG from Period Splitting of One RNG

Pierre L’Ecuyer proposed a very efficient RNG (1996) which is a CMRG on 32 bits: Combination of two Multiple Recursive Generator (MRG) with lag = 3 for each MRG.

* Very long period \( \sim 2^{185} \)

\[ x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \mod(m) \]

Pre-computations

We launch as many parallel RNGs as the number of paths

Use

We prefer local variables to store the RNG’s state vector
Some OpenACC pragmas: `#pragma acc clause`

**global variables**

clause = `declare device_resident`, applied to $r_g$, $K_g$, $T_g$ and $C_g$. These arrays are known by all GPU functions.

**copying to GPU**

clause = `enter data copyin` copies the values to the GPU.

**embarrassingly parallel**

clause = `parallel loop present (+reduction(operation,arguments))` making parallel the execution of the loop without data movement.

`reduction(operation,arguments)` reduces all the private copies of `arguments` into one final result using `operation`.
Plan

Introduction

Local volatility challenges GPUs

Monte Carlo and OpenACC parallelization

Monte Carlo and CUDA parallelization

PDE formulation & Crank-Nicolson scheme

PDE simulation and CUDA parallelization
GPU architecture

Hardware software equivalence
- Streaming processor: Executes threads
- Streaming multiprocessor: Executes blocks

Built-in variables
Known within functions executed on GPU: threadIdx.x, blockIdx.x, blockDim.x, gridDim.x

Global memory
GPU RAM, allocated thanks to `cudaMalloc`. Transfer values from GPU/CPU to CPU/GPU thanks to `cudaMemcpy`

Constant memory
Read only, declared globally. Data transfer with `cudaMemcpyToSymbol`

Shared
Declared in the kernel
Function declaration and calling

Standard C functions

The same as for C or C++ programming

Device functions

- Called by the GPU and executed on the GPU
- Declared as
  ```
  __device__ void myDivFun (...) { ...; }
  __device__ float myDivFun (...) { ...; }
  ```
- Called simply by `myDivFun(...)` but only within other device functions or kernels

Kernel functions

- Called by the CPU and executed on the GPU
- Declared as `__global__ void myKernel (...) { ...; }
- Called standardly by
  ```
  myKernel<<<numBlocks, threadsPerBlock>>>(...);
  ```
where
  - `numBlocks` should take into account the number of multiprocessors
  - `threadsPerBlock` should be equal to 128, 256, 512 or 1024
- Dynamic parallelism: kernels can be called within kernels by the GPU and executed on the GPU
**Adaptation steps**

**Extension**
Change it from MC.cpp and rng.cpp to MC.cu and rng.cu

**__constant__**
Declare $T_g$, $K_g$, $r_g$ and $C_g$ using constant memory

**Copying**
Copy the values $T_g$, $K_g$, $r_g$, $C_g$ and $CMRG$ on the GPU

**__device__**
Except main, MC, VarMalloc, FreeVar and parameters, define all other functions in MC.cu using **__device__**

**Parallel**
Declare MC function as a kernel using **__global__** + Replace the Monte Carlo loop using threadIdx.x, blockDim.x, blockIdx.x in MC kernel

**Reduction**
Perform the reduction using **__shared__** memory in kernel MC and atomicAdd function
float Tim;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

/****************************************************************************

To compute the execution time of this part of the code

****************************************************************************/

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&Tim, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
Plan

Introduction

Local volatility challenges GPUs

Monte Carlo and OpenACC parallelization

Monte Carlo and CUDA parallelization

PDE formulation & Crank-Nicolson scheme

PDE simulation and CUDA parallelization
PDE formulation & Crank-Nicolson scheme

The price
\[ F(t, x, j) = e^{-\int_t^{T_g} r(u) du} E(X \mid S_t = x, I_t = j), \quad X = (S_T - K)^{+}1_{\{I_T \in [P_1, P_2]\}} \]

For \( u(t, x, j) = e^{\int_t^{T_g} r(u) du} F(t, e^x, j) \), we equivalently solve \( u \) PDE given by
\[ \frac{1}{2} \sigma^2_{loc}(x, t) \frac{\partial^2 u}{\partial x^2}(t, x, j) + \mu(x, t) \frac{\partial u}{\partial x}(t, x, j) = -\frac{\partial u}{\partial t}(t, x, j) \quad (5) \]

\[ \mu(x, t) = r_g(t) - \frac{\sigma^2_{loc}(x, t)}{2}, \quad (x, t, j) \in [0, \max(K_g)] \times ]T_k, T_{k+1}[ \times [0, P_2] \]
with \( T_0 = 0 \) and \( T_{M+1} = T \)

Final and boundary conditions
\[ u(T, x, j) = \max(e^x - K, 0) \] for any \((x, j)\) and heuristically
\[ u(t, \log[\min(K_g)], j) = p_{\min} = 0 \] and
\[ u(t, \log[\max(K_g)], j) = p_{\max} = \max(K_g) - K \]

Crank-Nicolson scheme
Denoting \( u_{k,i,j} = u(t_k, x_i, j), \sigma = \sigma_{loc}(x_i, t_k) \) and \( \mu = \mu(x_i, t_k) \)
\[ q_u u_{k+1,i,j} + q_m u_{k,i,j} + q_d u_{k-1,i,j} = p_u u_{k+1,i+1,j} + p_m u_{k+1,i,j} + p_d u_{k+1,i-1,j} \]
\[ q_u = -\frac{\sigma^2 \Delta t}{4\Delta x^2} - \frac{\mu \Delta t}{4\Delta x}, \quad q_m = 1 + \frac{\sigma^2 \Delta t}{2\Delta x^2}, \quad q_d = -\frac{\sigma^2 \Delta t}{4\Delta x^2} + \frac{\mu \Delta t}{4\Delta x} \]
\[ p_u = \frac{\sigma^2 \Delta t}{4\Delta x^2} + \frac{\mu \Delta t}{4\Delta x}, \quad p_m = 1 - \frac{\sigma^2 \Delta t}{2\Delta x^2}, \quad p_d = \frac{\sigma^2 \Delta t}{4\Delta x^2} - \frac{\mu \Delta t}{4\Delta x} \]
PDE formulation & Crank-Nicolson scheme

PDE, \( u(t, x, j) \) with respect to \( j \), barrier condition

\[
\begin{align*}
\frac{u_t(x, j)}{u_t(x, j)} &= \mathbb{E} \left( (S_t - K)_+ \mathbb{1} \left\{ \sum_{i=1}^M 1\{S_{T_i} < B\} \in [P_1, P_2] \right\} \right) \left| S_t = x, \sum_{T_i \leq t} 1\{S_{T_i} < B\} = j \right. \\
& \quad \left. \in [T_M, T] \right.
\end{align*}
\]

\[ t \in [T_M, T] \]

\[
\begin{align*}
\frac{u_t(x, j)}{u_t(x, j)} &= \mathbb{E}[(S_T - K)_+ | S_t = x]1_{\{j = P_2\}} \\
& \quad + \mathbb{E}[(S_T - K)_+ | S_t = x]1_{\{j = P_1 - 1\}} \\
& \quad + \mathbb{E}[(S_T - K)_+ | S_t = x]1_{\{j \in [P_1, P_2 - 1]\}}
\end{align*}
\]

\[
\begin{align*}
\frac{u_t(x, j)}{u_t(x, j)} &= \mathbb{E}[u_{T_{M-k}}(S_{T_{M-k}}, P_2)1\{S_{T_{M-k}} \geq B\} | S_t = x]1_{\{j = P_2\}} \\
& \quad + \mathbb{E}[u_{T_{M-k}}(S_{T_{M-k}}, p^1_k)1\{S_{T_{M-k}} < B\} | S_t = x]1_{\{j = p^1_k - 1\}} \\
& \quad + \mathbb{E}\left[u_{T_{M-k}}(S_{T_{M-k}}, j)1\{S_{T_{M-k}} \geq B\} | S_t = x\right]1_{\{j \in [p^1_k, P_2 - 1]\}} \\
& \quad + u_{T_{M-k}}(S_{T_{M-k}}, j + 1)1\{S_{T_{M-k}} < B\}
\end{align*}
\]

\[ k = M - 1, \ldots, 1 \]

\[ (T_0 = 0) \]

with \( p^1_k = \max(P_1 - k, 0) \)
$M = 10$

$P_1 = 3$

$P_2 = 8$
Plan

Introduction

Local volatility challenges GPUs

Monte Carlo and OpenACC parallelization

Monte Carlo and CUDA parallelization

PDE formulation & Crank-Nicolson scheme

PDE simulation and CUDA parallelization
Solve tridiagonal system for the implicit part

\[
T = \begin{pmatrix}
  d_1 & c_1 & & & & & \\
  a_2 & d_2 & c_2 & & & & \\
  & a_3 & d_3 & & & & \\
  & & & \ddots & & & \\
  & & & & 0 & & \\
  & & & & & a_n & d_n \\
\end{pmatrix}.
\]

Cyclic Reduction

Step 1: Forward reduction to a 4-unknown system involving \(z_2, z_4, z_6\) and \(z_8\)

Step 2: Forward reduction to a 2-unknown system involving \(z_4\) and \(z_8\)

Step 3: Solve 2-unknown system

Step 4: Backward substitution to solve the rest 2 unknowns

Step 5: Backward substitution to solve the rest 4 unknowns
\[
\begin{pmatrix}
  d_1 & c_1 \\
  c_1 & d_2 & c_2 \\
  c_2 & d_3 & c_3 \\
  c_3 & d_4 & c_4 \\
  c_4 & d_5 & c_5 \\
  c_5 & d_6 & c_6 \\
  c_6 & d_7
\end{pmatrix}
\]

\[
\begin{pmatrix}
  d'_1 & 0 & c'_2 \\
  0 & d'_2 & 0 & c'_3 \\
  c'_2 & 0 & d'_3 & 0 & c'_4 \\
  c'_3 & 0 & d'_4 & 0 & c'_5 \\
  c'_4 & 0 & d'_5 & 0 & c'_6 \\
  c'_5 & 0 & d'_6 & 0 & c'_7 \\
  c'_6 & 0 & d'_7
\end{pmatrix}
\]

\[
\begin{pmatrix}
  d''_1 & c''_2 \\
  c''_2 & d''_3 & c''_4 \\
  c''_3 & d''_4 & c''_5 \\
  c''_4 & d''_6 & c''_7 \\
  c''_5 & d''_7
\end{pmatrix}
\]
Adaptation steps

Extension
Change it from PDE.cpp to PDE.cu

__constant__
Declare $T_g$, $K_g$, $r_g$ and $C_g$ using constant memory

Copying
Copy the values $T_g$, $K_g$, $r_g$, $C_g$ on the GPU

__device__
Except for main, PDE_Diff, Out2In, VarMalloc, FreeVar and parameters, define all other functions in PDE.cu using __device__

Registers
Compare the CPU version of PCR to the GPU version given to you. Remark the large use of shared memory and registers.

Parallel
Declare PDE_Diff and Out2In functions as kernels using __global__ + Replace loops related to $S_t$ by threadIdx.x and loops related to $I_t$ by blockIdx.x in PDE_Diff and Out2In kernels as well as in Exp device function.
```c
float Tim;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

/*********************
To compute the execution time of this part of the code
***************************/
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&Tim, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```
References


