Sorbonne Université

Massive parallel programming on GPU devices for Big Data

Nested Monte Carlo, batch sort, regression and Machine Learning

The presentation of the results on March 31st

The source code must be readable, sufficiently commented.
The source code and slides must be sent before March 31st
Presentation duration = 4 minutes for results + 3 minutes for code + 3 minutes of questions

1 Batch merge small

This subject starts with the same algorithm of merge path, presented in [3], implemented on only one block. The students are then asked to translate it into the batch merge of small arrays.

We start with the merge path algorithm. Let A and B be two ordered arrays (increasing order), we want to merge them in an M sorted array. The merge of A and B is based on a path that starts at the top-left corner of the $|A| \times |B|$ grid and arrives at the down-right corner. The Sequential Merge Path is given by Algorithm 1 and an example is provided in Figure 1.

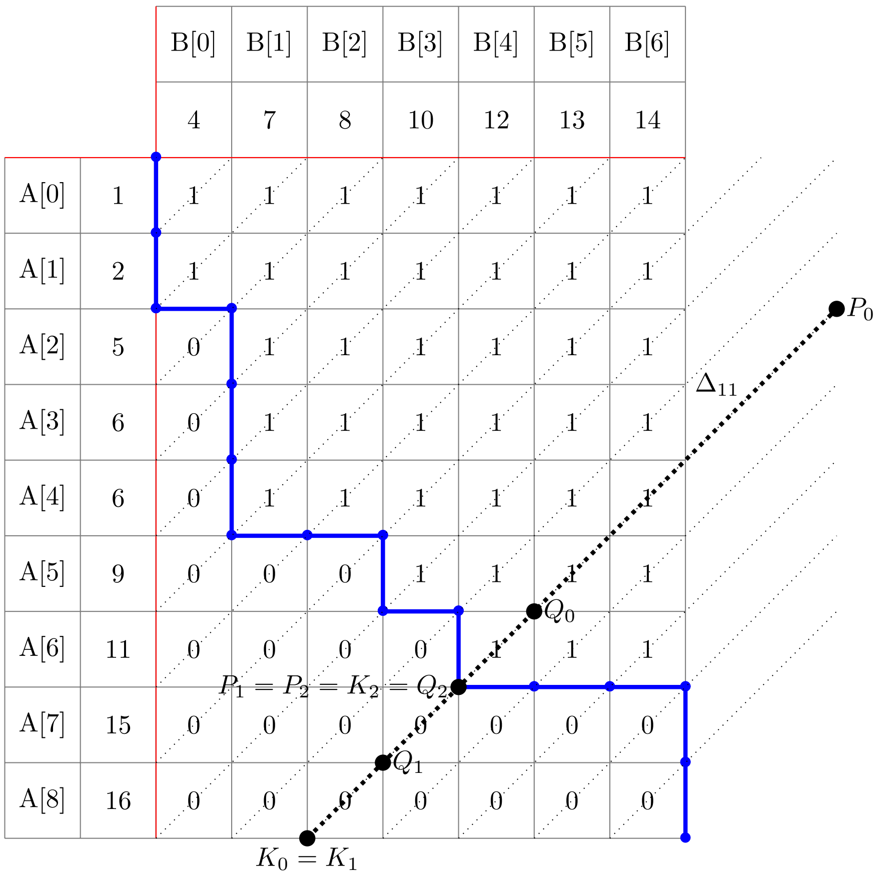


Figure 1: An example of Merge Path procedure

Algorithm 1 Sequential Merge Path

```
Require: $A$ and $B$ are two sorted arrays
Ensure: $M$ is the merged array of $A$ and $B$ with $|M| = |A| + |B|$
  procedure MergePath $(A, B, M)$
    $j = 0$ and $i = 0$
    while $i + j < |M|$ do
      if $i \geq |A|$ then
        M[i+j]=B[j]
        $j = j + 1$                                    ▷ The path goes right
      else if $j \geq |B|$ or $A[i] < B[j]$ then
        M[i+j]=A[i]                                    ▷ The path goes down
        $i = i + 1$
      else
        M[i+j]=B[j]
        $j = j + 1$                                    ▷ The path goes right
      end if
    end while
  end procedure
```

Algorithm 2 Merge Path (Indices of n threads are 0 to $n - 1$)

```
Require: $A$ and $B$ are two sorted arrays
Ensure: $M$ is the merged array of $A$ and $B$ with $|M| = |A| + |B|$
  for each thread in parallel do
    i=index of the thread
    if $i > |A|$ then
      $K = (i - |A|, |A|)$                              ▷ Low point of diagonal
      $P = (|A|, i - |A|)$                              ▷ High point of diagonal
    else
      $K = (0, i)$
      $P = (i, 0)$
    end if
    while True do
      $offset = abs(K_y - P_y)/2$
      $Q = (K_x + offset, K_y - offset)$
      if $Q_y \geq 0$ and $Q_x \leq |B|$ and
         $(Q_y = |A|$ or $Q_x = 0$ or $A[Q_y] > B[Q_x - 1])$ then
        if $Q_x = |B|$ or $Q_y = 0$ or $A[Q_y - 1] \leq B[Q_x]$ then
          if $Q_y < |A|$ and $(Q_x = |B|$ or $A[Q_y] \leq B[Q_x])$ then
            $M[i] = A[Q_y]$                              ▷ Merge in $M$
          else
            $M[i] = B[Q_x]$
          end if
          Break
        else
          $K = (Q_x + 1, Q_y - 1)$
        end if
      else
        $P = (Q_x - 1, Q_y + 1)$
      end if
    end while
  end for
```

Each point of the grid has a coordinate $(i, j) \in [\![0, |A|]\!] \times [\![0, |B|]\!]$. The merge path starts from the point $(i, j) = (0, 0)$ on the left top corner of the grid. If $A[i] < B[j]$ the path goes down else it goes right. The array $[\![0, |A|-1]\!] \times [\![0, |B|-1]\!]$ of boolean values $A[i] < B[j]$ is not important in the algorithm. However, it shows clearly that the merge path is a frontier between ones and zeros.

To parallelize the algorithm, the grid has to be extended to the maximum size equal to $\max(|A|, |B|) \times \max(|A|, |B|)$. We denote K_0 and P_0 respectively the low point and the high point of the ascending diagonals Δ_k. On GPU, each thread $k \in [\![0, |A| + |B| - 1]\!]$ is responsible of one diagonal. It finds the intersection of the merge path and the diagonal Δ_k with a binary search described in Algorithm 2.

1. For $|A| + |B| \leq 1024$, write a kernel `mergeSmall_k` that merges A and B using only one block of threads. (10 points)

In this part, we assume that we have a large number $N(\geq 1e3)$ of arrays $\{A_i\}_{1\leq i\leq N}$ and $\{B_i\}_{1\leq i\leq N}$ with $|A_i| + |B_i| = d \leq 1024$ for each i. Using some changes on `mergeSmall_k`, we would like to write `mergeSmallBatch_k` that merges two by two, for each i, A_i and B_i.

Given a fixed common size $d \leq 1024$, `mergeSmallBatch_k` is launched using the syntax

```
mergeSmallBatch_k<<<numBlocks, threadsPerBlock>>>(...);
```

with `threadsPerBlock` is multiple of d but smaller than 1024 and `numBlocks` is an arbitrary sufficiently big number.

Try to figure out how the indices
```
int Qt = threadIdx.x/d;
int tidx = threadIdx.x - Qt*d;
int gbx = Qt + blockIdx.x*(blockDim.x/d);
```
are important in the definition of `mergeSmallBatch_k`.

2. Write the kernel `mergeSmallBatch_k` that batch merges two by two $\{A_i\}_{1\leq i\leq N}$ and $\{B_i\}_{1\leq i\leq N}$. (4 points)

3. Study the execution time with respect to d. What can be done to optimize further the code? (3 points + 3 points)

2 Monte Carlo simulation of Heston model

The Heston model for asset pricing has been widely examined in the literature. Under this model, the dynamics of the asset price S_t and the variance v_t are governed by the following system of stochastic differential equations:

$$dS_t = rS_t dt + \sqrt{v_t} S_t d\hat{Z}_t \quad (1)$$
$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t \quad (2)$$
$$\hat{Z}_t = \rho W_t + \sqrt{1-\rho^2} Z_t \quad (3)$$

where:

- the spot values $S_0 = 1$ and $v_0 = 0.1$,
- r is the risk-free interest rate, we assume $r = 0$,
- κ is the mean reversion rate of the volatility,
- θ is the long-term volatility,
- σ is the volatility of volatility,
- W_t and Z_t are independent Brownian motions

In this project, we aim to compare three distinct methods for simulating an at-the-money call option (where "at-the-money" here means $K = S_0 = 1$) at maturity $T = 1$ under the Heston model. The option has a payoff given by $f(x) = (x - K)_+$, and, thus, we want to simulate with Monte Carlo the expectation $E[f(S_T)] = E[(S_1 - 1)_+]$. This comparison will focus on the efficiency and accuracy of each simulation method in pricing the call option within the stochastic volatility framework of the Heston model.

We begin with the Euler discretization scheme, which updates the asset price S_t and the volatility v_t at each time step as follows:

$$S_{t+\Delta t} = S_t + rS_t\Delta t + \sqrt{v_t}S_t\sqrt{\Delta t}(\rho G_1 + \sqrt{1-\rho^2}G_2) \tag{4}$$

$$v_{t+\Delta t} = g\left(v_t + \kappa(\theta - v_t)\Delta t + \sigma\sqrt{v_t}\sqrt{\Delta t}G_1\right) \tag{5}$$

where G_1 and G_2 are independent standard normal random variables, and the function g is either taken to be equal to $(\cdot)_+$ or to $|\cdot|$.

1. Assuming $\kappa = 0.5$, $\theta = 0.1$, $\sigma = 0.3$ and using a discretization $\Delta t = 1/1000$, write down a Monte Carlo simulation code using Euler discretization to approximate $E[(S_1 - 1)_+]$. (8 points)

As an alternative to the Euler scheme, we are going to implement a version of the exact simulation procedure presented in the paper [1]. The steps that we choose differ a little bit as we take the discretization $\Delta t = 1/1000$ and we do the following:

step 1. Within a for loop on time steps, we define

$$d = 2\kappa\theta/\sigma^2, \quad \lambda = \frac{2\kappa e^{-\kappa\Delta t}v_t}{\sigma^2(1-e^{-\kappa\Delta t})}, \quad N = \mathcal{P}(\lambda), \quad \text{with } \mathcal{P} \text{ simulated by } \texttt{curand\_poisson}$$

and denoting $\mathcal{G}(\alpha)$ the standard gamma distribution whose simulation is presented in [6]

$$v_{t+\Delta t} = \frac{\sigma^2(1-e^{-\kappa\Delta t})}{2\kappa}\mathcal{G}(d+N).$$

step 2. The integral $\int_0^1 v_s ds$ is stored in a variable `vI` set to zero before the for loop on time steps. Then, in the for loop we update `vI+=`$0.5 * (v_t + v_{t+\Delta t})\Delta t$.

step 3. Once we finish the for loop, we compute $\int_0^1 \sqrt{v_s}dW_s$, using the expression

$$\int_0^1 \sqrt{v_s}dW_s = \frac{1}{\sigma}\left(v_1 - v_0 - \kappa\theta + \kappa\texttt{vI}\right)$$

Then we compute

$$m = -0.5\texttt{vI} + \rho\int_0^1 \sqrt{v_s}dW_s, \quad \Sigma^2 = (1-\rho^2)\texttt{vI}$$

and we set $S_1 = \exp(m + \Sigma G)$ where G is a standard normal random variable independent from (G_1, G_2).

2. Define a device function that simulates the standard gamma distribution $\mathcal{G}(\alpha)$ presented in [6]. Make sure to deal with both situations $\alpha \geq 1$ and $\alpha < 1$. Then write down the code for the exact Monte Carlo simulation that is based on this standard gamma distribution. (5 points + 3 points)

3. For many values of $\kappa \in [0.1, 10]$, $\theta \in [0.01, 0.5]$ and $\sigma \in [0.1, 1]$ such that $20\kappa\theta > \sigma^2$, compare the execution time of the Euler discretization scheme to the exact simulation procedure. Compare the execution time of the previous two methods to the almost exact scheme

$$S_{t+\Delta t} = S_t + k_0 + k_1 v_t + k_2 v_{t+\Delta t} + \sqrt{(1-\rho^2)v_t}\sqrt{\Delta t}(\rho G_1 + \sqrt{1-\rho^2}G_2)$$

with

$$k_0 = \left(-\frac{\rho}{\sigma}\kappa\theta\right)\Delta t$$

$$k_1 = \left(\frac{\rho\kappa}{\sigma} - 0.5\right)\Delta t - \frac{\rho}{\sigma}$$

$$k_2 = \frac{\rho}{\sigma}$$

presented in [8] where v_t is simulated as in the exact scheme. (2 points + 2 points)

3 Nested Monte Carlo for Log-Variance-Gamma model

In the Variance Gamma model, the asset price is modeled as the exponential of a Variance Gamma process. This Lévy process can be represented via Brownian subordination. The variance Gamma process is described by the following dynamics:

$$X_{\mathrm{VG}}(t;\kappa,\sigma,\theta) := \theta Z_t + \sigma W(Z_t),$$

An asset Y is governed by the exponential VG dynamics if

$$Y_t = Y_0 e^{X_{\mathrm{VG}}(t)},$$

There are three free parameters in total, they are κ, θ, and σ. The initial values of the asset price Y_0 is set to one.

The price of the call option is the expectation of the payoff in the risk free measure:

$$C(T, K, \kappa, \theta, \sigma) = \mathbb{E}[(Y_T - K)^+].$$

First, the students will be asked to generate a dataset of prices $\hat{C}$ obtained by Monte Carlo simulation such that $\hat{C}(T, K, \kappa, \theta, \sigma) = \hat{\mathbb{E}}[(Y_T - K)^+]$, where $\hat{\mathbb{E}}$ denotes the empirical expectation. The parameters are stored with the $\hat{C}$ and its confidence interval in the csv files whose names contain strike and maturity. Second, the students will train a neural network $f(T, K, \kappa, \theta, \sigma) \approx \hat{C}(T, K, \kappa, \theta, \sigma)$ to approximate the Monte Carlo price. Last but not least, the students should study the monotonicity of f prices with respect to T and K and the convexity with respect to K.

Following [5], Y can be simulated by

$$Y_t = \exp\left(wt + \sum_{t_i \le t} \Delta X_{t_i}\right)$$

$$\Delta X_{t_i} = \sigma N_i \sqrt{\kappa \Delta S_i} + \theta\kappa\Delta S_i$$

$$\Delta S_i \sim \frac{x^{\Delta t/\kappa - 1}}{\Gamma(\Delta t/\kappa)} e^{-x} 1_{x>0}, \quad N_i \sim \mathcal{N}(0,1)$$

with $w = \ln\left(1 - \theta\kappa - \kappa\sigma^2/2\right)/\kappa$ being the adjustment term for martingale. The simulation can be performed using the following algorithms (from [2])

ALGORITHM 6.11 Simulating a variance gamma process on a fixed time grid

Simulation of $(X(t_1), \ldots, X(t_n))$ for fixed times $t_1, \ldots, t_n$: a discretized trajectory of the variance gamma process with parameters σ, θ, κ.

- *Simulate, using Algorithms 6.7 and 6.8, n independent gamma variables $\Delta S_1, \ldots, \Delta S_n$ with parameters $\frac{t_1}{\kappa}, \frac{t_2 - t_1}{\kappa}, \ldots, \frac{t_n - t_{n-1}}{\kappa}$. Set $\Delta S_i = \kappa \Delta S_i$ for all i.*
- *Simulate n i.i.d. $N(0,1)$ random variables $N_1, \ldots, N_n$. Set $\Delta X_i = \sigma N_i \sqrt{\Delta S_i} + \theta \Delta S_i$ for all i.*

The discretized trajectory is $X(t_i) = \sum_{k=1}^{i} \Delta X_i$.

ALGORITHM 6.7 Johnk's generator of gamma variables, $a \leq 1$

```
REPEAT
    Generate i.i.d. uniform $[0,1]$ random variables $U$, $V$
    Set $X = U^{1/a}$, $Y = V^{1/(1-a)}$
UNTIL $X + Y \leq 1$
Generate an exponential random variable $E$
RETURN $\frac{XE}{X+Y}$
```

ALGORITHM 6.8 Best's generator of gamma variables, $a \geq 1$

```
Set $b = a - 1$, $c = 3a - \frac{3}{4}$
REPEAT
    Generate i.i.d. uniform $[0,1]$ random variables $U$, $V$
    Set $W = U(1-U)$, $Y = \sqrt{\frac{c}{W}}(U - \frac{1}{2})$, $X = b + Y$
    If $X < 0$ go to REPEAT
    Set $Z = 64W^3V^3$
UNTIL $\log(Z) \leq 2(b\log(\frac{X}{b}) - Y)$
RETURN $X$
```

1. Write down the Nested Monte Carlo simulation code that generates the required dataset. (12 points)
2. Train a neural network f that approximates the prices. (4 points)
3. Study the monotonicity of f with respect to T and K, as well as its convexity with respect to K. How can it be improved? (2 points + 2 points)

4 Nested Monte Carlo for Asian options and training

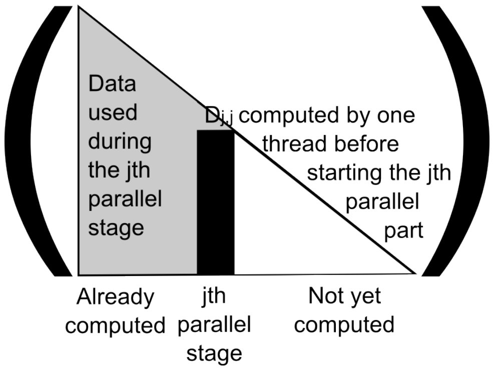
Let F be the price of an Asian option whose value is given by $F(t,x,y) = E(X \big| S_t = x, I_t = y)$, $X = (S_T - I_T)_+$ where the processes I and S are defined by

$$I_t = \frac{1}{t}\int_0^t S_s ds, \quad dS_t = S_t \sigma dW_t \text{ and } S_0 = x_0.$$

- K, T are respectively the contract's strike and maturity. We assume $K = x_0 = 100$ and $T = 1$.
- W is a Brownian motion
- $\sigma = 0.2$ is the volatility.

1. Change the code given to you in order to make possible the parallel simulation on GPUs of Asian options for many choices of starting values of S_t and I_t at any time t in the grid $\{\Delta_t, 2\Delta_t, ..., 1\}$ with $\Delta_t = 1/100$. (9 points)
2. Save the obtained values in CSV files and subsequently train a neural network on these data points to approximate the function F. (4 points)

In the following, we want to compare the results of 2. to monomial regression using LDLt factorization given to you. As shown on the figure


Data used during the jth parallel stage
D_{j,j} computed by one thread before starting the jth parallel part
Already computed
jth parallel stage
Not yet computed

the parallel implementation of LDLt is performed as follows: for a fixed value of j, the different coefficients $\{L_{i,j}\}_{j+1\leq i\leq d}$ can be computed by at most $d-j$ independent threads. Thus, $\{L_{i,1}\}_{2\leq i\leq d}$ involves the biggest number of possible independent threads equal to $d-1$. In this collaborative version, we use the maximum $d-1$ threads $+1$ additional thread that is involved in the copy from global to shared and in the solution of the system after factorization. This makes d threads for the collaborative version and one of these threads is also involved in the computation $D_{j,j}$ which needs a synchronization before calculating $L_{i,j}$.

3. Explain why the indices
```
int Qt = threadIdx.x/d;
int tidx = threadIdx.x - Qt*d;
int gbx = Qt + blockIdx.x*(blockDim.x/d);
```
are important in the definition of LDLt function given to you.

Replace the neural network regression in 2. by a monomial regression at any time step t in the grid $\{\Delta_t, 2\Delta_t, ..., 1\}$ with $\Delta_t = 1/100$. Study the monotonicity of the two regressions with respect to t. How to improve the results? (4 points + 2 points + 1 point).

5 Nested Monte Carlo for uncertain volatility model

Let F be the price of a call spread $F(t,x) = E(g(S^*_T)\big|S^*_t = x)$ with $g(y) = (y-K_1)_+ - (y-K_2)_+$ and $90 = K_1 < K_2 = 110$ where the process S^* is defined on the grid $t_k \in \{1/N, 2/N, ..., 1\}$ with $N = 100$, by

$$S^*_{t_k} = S^*_{t_{k-1}}[1+\sigma^*(t_{k-1}, S^*_{t_{k-1}})\Delta W_{t_k}] \text{ and } S^*_{t_0} = 100$$

and its randomized start family process $\{S^j\}_{j=0,...,N-1}$ is given by

$$S^j_{t_{k+1}} = \begin{cases} S^j_{t_k}[1+\sigma^*(t_k, S^j_{t_k})\Delta W_{t_{k+1}}] & \text{if } N \geq k > j \geq 0 \\ S^j_{t_k}[1+\sigma^j(t_k)\Delta W_{t_{k+1}}] & \text{if } 1 \leq k \leq j \leq N-1 \end{cases}, \text{ with } S^j_{t_0} = 100.$$

- T is the contract's maturity assumed to be equal to one.
- $\{\Delta W_{t_k}\}_{k=1,...,N} = \{W_{t_k} - W_{t_{k-1}}\}_{k=1,...,N}$ are independent Brownian motion increments.
- $\{\sigma^*(t_k, x)\}_{k=0,...,N-1}$ are unknown coefficients that take their values in $\{\sigma_{min}, \sigma_{max}\}^N$ with $\sigma_{min} = 0.1$ and $\sigma_{max} = 0.3$.
- $\{\sigma^j(t_k)\}_{k=1,...,j}$ are independent and Bernoulli distributed variables with $P(\sigma^j(t_k) = \sigma_{min}) = P(\sigma^j(t_k) = \sigma_{max}) = 1/2$.

One can thus show that

$$F(t_j,x)=\max\left(E\left[g(S^j_T)\big|S^j_{t_j}=x,\sigma^j(t_j)=\sigma_{min}\right],E\left[g(S^j_T)\big|S^j_{t_j}=x,\sigma^j(t_j)=\sigma_{max}\right]\right)$$

which, through preconditionning, provides us

$$F(t_j,x)=\max\left(E\left[F(t_{j+1},S^j_{t_{j+1}})\big|S^j_{t_j}=x,\sigma^j(t_j)=\sigma_{min}\right],E\left[F(t_{j+1},S^j_{t_{j+1}})\big|S^j_{t_j}=x,\sigma^j(t_j)=\sigma_{max}\right]\right). \tag{6}$$

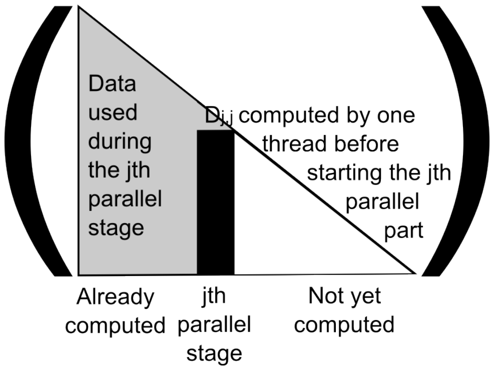
1. Given any Borel function f with $f(S^j_{t_{j+1}})$ integrable, write down a Nested Monte Carlo routine on GPU that is able to compute $E\left[f(S^j_{t_{j+1}})\big|S^j_{t_j},\sigma^j(t_j)\right]$. (10 points)

We want now to replace the pointwise evaluation

$$h(t_j,S^j_{t_j})=\max\left(E\left[f(S^j_{t_{j+1}})\big|S^j_{t_j},\sigma^j(t_j)=\sigma_{min}\right],E\left[f(S^j_{t_{j+1}})\big|S^j_{t_j},\sigma^j(t_j)=\sigma_{max}\right]\right)$$

into a functional one using a monomial regression that projects the ralizations $\{h(t_j,S^{j,m}_{t_j})\}_{m=1,...,M}$ on monomials of of $\{S^{j,m}_{t_j}\}_{m=1,...,M}$. The monomial regression involves the LDLt factorization given to you.

As shown on the figure


Data used during the jth parallel stage
D_{j,j} computed by one thread before starting the jth parallel part
Already computed
jth parallel stage
Not yet computed

the parallel implementation of LDLt is performed as follows: for a fixed value of j, the different coefficients $\{L_{i,j}\}_{j+1\le i\le d}$ can be computed by at most $d-j$ independent threads. Thus, $\{L_{i,1}\}_{2\le i\le d}$ involves the biggest number of possible independent threads equal to $d-1$. In this collaborative version, we use the maximum $d-1$ threads $+1$ additional thread that is involved in the copy from global to shared and in the solution of the system after factorization. This makes d threads for the collaborative version and one of these threads is also involved in the computation $D_{j,j}$ which needs a synchronization before calculating $L_{i,j}$.

2. Explain why the indices

```
int Qt = threadIdx.x/d;
int tidx = threadIdx.x - Qt*d;
int gbx = Qt + blockIdx.x*(blockDim.x/d);
```

   are important in the definition of LDLt function given to you.

   Implement a monomial regression associated to any pointwise values of h as above. (4 points)

3. Propose a complete backward algorithm that computes, at each time step $t_j\in\{1/N,2/N,...,(N-1)/N\}$, the pointwise Monte Carlo approximation of $F(t_j,\cdot)$ and replaces it with a regression which is then replaced by induction (6) to obtain the pointwise Monte Carlo approximation of $F(t_{j-1},\cdot)$ and so on. (3 points)

   What do you propose to approximate first and second order derivatives: $\partial_x F(t_{j-1},x)$ and $\partial^2_x F(t_{j-1},x)$? (3 points)

References

[1] Mark Broadie and Özgür Kaya. Exact simulation of stochastic volatility and other affine jump diffusion processes. *Operations research*, 54(2):217–231, 2006.

[2] R. Cont and P. Tankov (2003): *Financial modelling with jump processes.* Chapman and Hall/CRC.

[3] O. Green, R. McColl and D. A. Bader GPU Merge Path - A GPU Merging Algorithm. *26th ACM International Conference on Supercomputing (ICS),* San Servolo Island, Venice, Italy, June 25-29, 2012.

[4] J. P. Fouque, G. Papanicolaou and K. R. Sircar, Mean-reverting stochastic volatility. *International Journal of theoretical and applied finance*, 3(01), 101–142, 2000.

[5] D. B. Madan, P. P. Carr and E. C. Chang (1998): The variance gamma process and option pricing. *Review of Finance*, 2(1), 79–105.

[6] G. Marsaglia and T. Wai-Wan. A simple method for generating gamma variables. *ACM Transactions on Mathematical Software*, 26(3):363–372, 2000.

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery (2002): *Numerical Recipes in C++: The Art of Scientific Computing.* Cambridge University Press.

[8] A. Van Haastrecht and A. Pelsser (2010) Efficient, almost exact simulation of the Heston stochastic volatility model. *International Journal of theoretical and applied finance*, 13(01), 1–43.

[9] Y. Zhang, J. Cohen and J. D. Owens (2010): Fast Tridiagonal Solvers on the GPU. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 127–136.