Project 1: Batch LDLt factorization for batch linear systems

Assuming a large number of $d \times d$ ($d \leq 1024$) symmetric and positive definite matrices $(A_n)_{1 \leq n \leq N}$, students have to optimize LDLt factorization to solve various linear systems associated to $(A_n)_{1 \leq n \leq N}$ at the same time.

For any kernel `myK` that batch computes problems of dimension d, we launch it using the syntax `myK<<<numBlocks, threadsPerBlock>>>(...);` with `threadsPerBlock` is multiple of d but smaller than 1024 and `numBlocks` is an arbitrary sufficiently big number. Explain why the indices

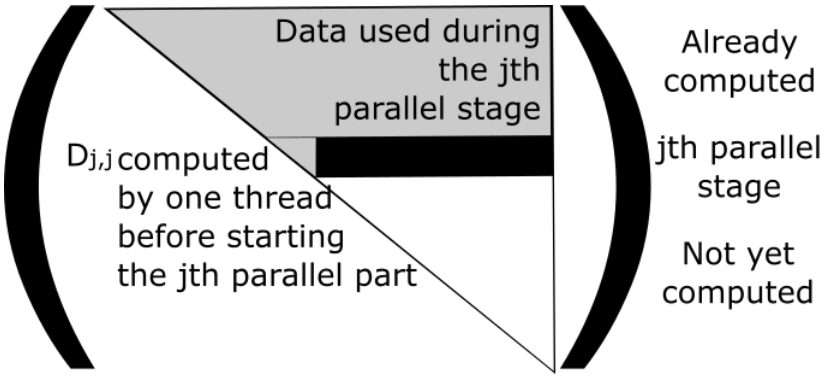
```
int tidx = threadIdx.x%d;
int Qt = (threadIdx.x-tidx)/d;
int gbx = Qt + blockIdx.x*(blockDim.x/d);
```

are important in the definition of `myK`.

Let us start with $d \leq 64$. This exercise is the continuation of the LDLt exercise studied in the Chapter 2 of the lecture notes. Here, we develop another version that involves a bigger number of threads. Indeed, calling this new kernel is performed in the `main` function by

```
LDLt_max_k<<<NB,d*minTB,minTB*((d*d+d)/2+d)*sizeof(float)>>>(AGPU,
YGPU, d);
```

This new version requires `d` collaborative threads per linear system that perform a row after row computation. In fact, as shown on the figure below,



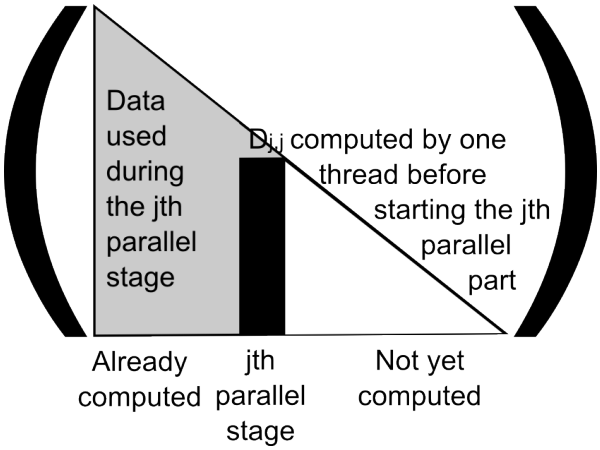
for a fixed value of j, the different coefficients $\{L_{i,j}\}_{j+1 \leq i \leq d}$ can be computed by at most $d-j$ independent threads. Thus, $\{L_{i,1}\}_{2 \leq i \leq d}$ involves the biggest number of possible independent threads equal to $d-1$. In this collaborative version, we use the maximum $d-1$ 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 $L_{i,j}$.

From the paragraph above make sure that you understand what is written in the following code

```
// Perform the LDLt factorization
 for(i=n; i>0; i--){
  if(tidx==0){
   for(k=n; k>i; k--){
    sA[nt+n2-i*(i+1)/2] -= sA[nt+n2-k*(k+1)/2]*
            sA[nt+n2-k*(k+1)/2+k-i]*
            sA[nt+n2-k*(k+1)/2+k-i];
   }
  }
  __syncthreads();
  if(tidx<i-1){
   sA[nt+n2-i*(i+1)/2+tidx+1] /= sA[nt+n2-i*(i+1)/2];
   for(k=n; k>i; k--){
    sA[nt+n2-i*(i+1)/2+tidx+1] -= sA[nt+n2-k*(k+1)/2]*
             sA[nt+n2-k*(k+1)/2+k-i]*
             sA[nt+n2-k*(k+1)/2+tidx+1+k-i]/
              sA[nt+n2-i*(i+1)/2];
   }
  }
  __syncthreads();
 }
```

Define this new kernel `LDLt_max_k`.

Now, write another kernel `LDLt_max_col_k` that performs the same computations as `LDLt_max_k` but on columns instead of rows as shown on the figure below.



For $d \leq 64$, compare the execution time of `LDLt_max_col_k` and of `LDLt_max_k`. For $64 < d \leq 1024$ explain and perform the needed adaptations then compare both kernels.

Project 2: Batch merge path sort

The goal of this subject is to implement a batch version of the merging sort presented in [1].

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

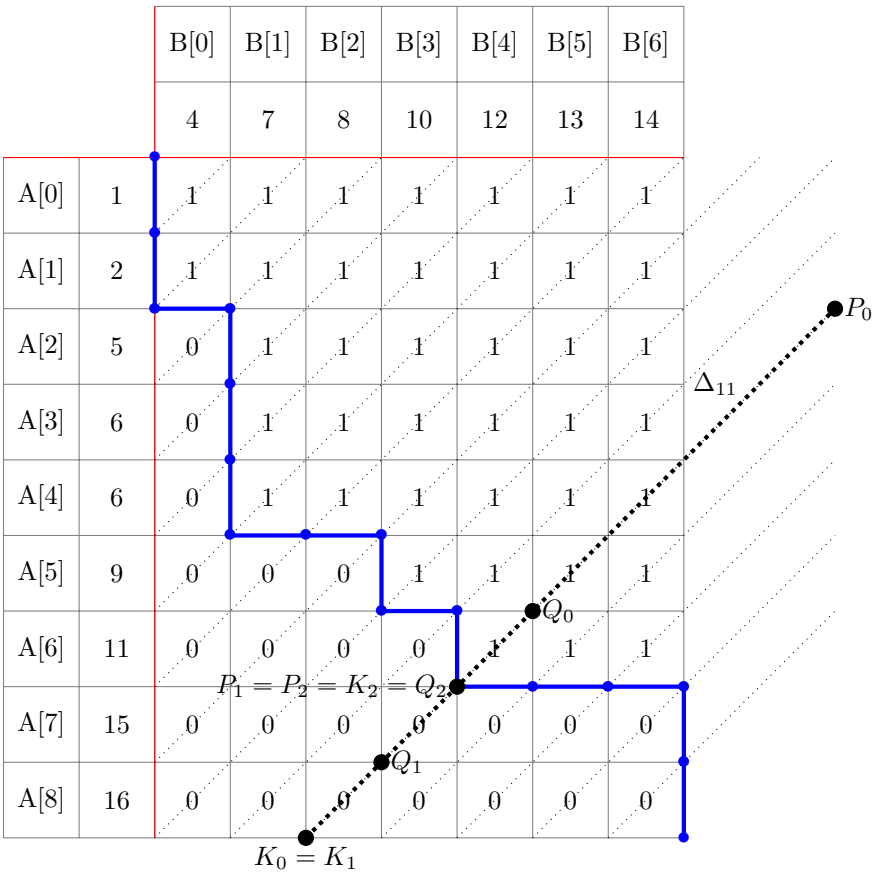


Figure 1: An example of merge path procedure

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.

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.

Algorithm 2 Merge path (Indexes 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

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 at least one diagonal. It finds the intersection of the merge path and the diagonal Δ_k with a binary search described in Algorithm 2.

Using the merge path algorithm, write a code that batch sorts various arrays $\{M_i\}_{1 \leq i \leq N}$ at the same time. These arrays have the same size d. The students have to associate only one block per array. They should start with $d \leq 1024$ then generalize the procedure for larger d.

Project 3: Parallel Cyclic and Householder reductions

Parallel cyclic reduction (PCR) is an alternative method to Thomas algorithm for the resolution of tridiagonal linear systems. It is stable for diagonally dominant matrices or symmetric and positive definite matrices and its implementation on GPUs is presented in [3]. First, the students have to implement and compare Thomas algorithm to PCR for tridiagonal systems. We refer to [2] and to Project 5 for a fair description of Thomas algorithm. Using Thomas method, write a kernel that solves various tridiagonal systems ($d \leq 1024$) at the same time, one system per block. Do the same thing for PCR then compare both methods.

Detailed in [2], Householder reduction is a stable method that provides a tridiagonal decomposition of any symmetric matrix. Assuming a large number of $d \times d$ ($d \leq 1024$) symmetric and positive definite matrices $(A_n)_{1 \leq n \leq N}$, the students have to combine Householder and PCR to batch solve linear systems associated to $(A_n)_{1 \leq n \leq N}$. Thus, the combination of Householder and PCR can be used as an alternative to LDLt presented in Project 2.

For any kernel `myK` that batch computes problems of dimension d, we launch it using the syntax `myK<<<numBlocks, threadsPerBlock>>>(...);` with `threadsPerBlock` is multiple of d but smaller than 1024 and `numBlocks` is an arbitrary sufficiently big number. Explain why the indices

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

are important in the definition of `myK`.

Let us start with the implementation of PCR applied to the following tridiagonal system

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{d-1} \\ & & & & a_d & b_d \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_d \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{pmatrix}. \tag{1}$$

One way of solving this system is to use the Cyclic Reduction (CR) that involves a forward phase of elimination then a backward phase to recover the whole solution. CR is sketched in the following figure.

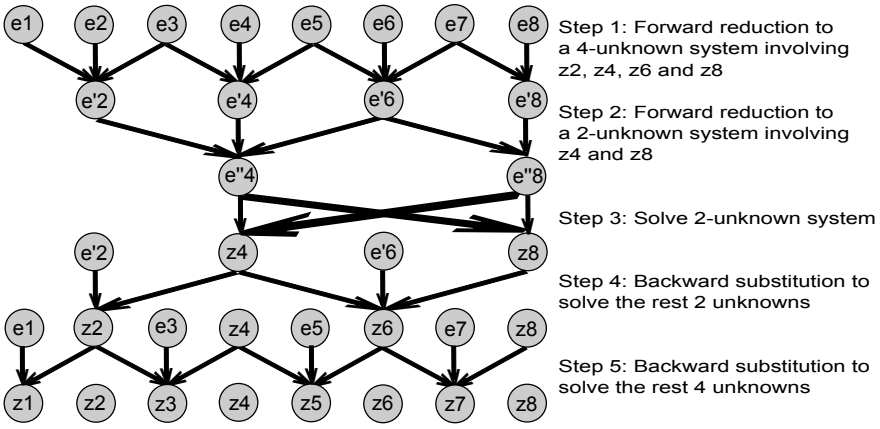


Figure 2: CR when $d = 8$.

However CR is not perfect for GPUs and a better alternative is to use PCR. Indeed, as shown on the figure bellow, PCR involves only a forward phase that consumes better the GPU resources

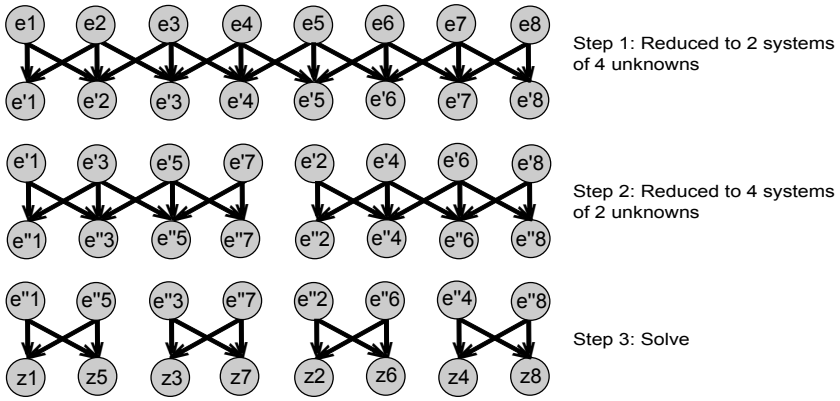


Figure 3: PCR when $d = 8$.

Using the index `P = (d / 2 + (d % 2))*(tidx % 2) + (int)tidx / 2;` explain why we can resolve systems of any size. Your explanations can be based on the operations sketched below.

$$
\begin{pmatrix}
b_1 & c_1 & & & & & & \\
a_2 & b_2 & c_2 & & & & & \\
 & a_3 & b_3 & c_3 & & & & \\
 & & a_4 & b_4 & c_4 & & & \\
 & & & a_5 & b_5 & c_5 & & \\
 & & & & a_6 & b_6 & c_6 & \\
 & & & & & a_7 & b_7 &
\end{pmatrix}
\xrightarrow{(R)}
\begin{pmatrix}
b'_1 & 0 & c'_1 & & & & & \\
0 & b'_2 & 0 & c'_2 & & & & \\
a'_3 & 0 & b'_3 & 0 & c'_3 & & & \\
 & a'_4 & 0 & b'_4 & 0 & c'_4 & & \\
 & & a'_5 & 0 & b'_5 & 0 & c'_5 & \\
 & & & a'_6 & 0 & b'_6 & 0 & \\
 & & & & a'_7 & 0 & b'_7 &
\end{pmatrix}
$$

$$
\xrightarrow{(P)}
\begin{pmatrix}
b'_1 & c'_1 & & & & & & \\
a'_3 & b'_3 & c'_3 & & & & & \\
 & a'_5 & b'_5 & c'_5 & & & & \\
 & & a'_7 & b'_7 & 0 & & & \\
 & & & 0 & b'_2 & c'_2 & & \\
 & & & & a'_4 & b'_4 & c'_4 & \\
 & & & & & a'_6 & b'_6 &
\end{pmatrix}
\xrightarrow{(R)}
\begin{pmatrix}
b''_1 & 0 & c''_1 & & & & & \\
0 & b''_3 & 0 & c''_3 & & & & \\
a''_5 & 0 & b''_5 & 0 & & & & \\
 & a''_7 & 0 & b''_7 & 0 & & & \\
 & & & 0 & b''_2 & 0 & c''_2 & \\
 & & & & 0 & b''_4 & 0 & \\
 & & & & a''_6 & 0 & b''_6 &
\end{pmatrix}
$$

$$
\xrightarrow{(P)}
\begin{pmatrix}
b''_1 & c''_1 & & & & & & \\
a''_5 & b''_5 & 0 & & & & & \\
 & 0 & b''_3 & c''_3 & & & & \\
 & & a''_7 & b''_7 & 0 & & & \\
 & & & 0 & b''_2 & c''_2 & & \\
 & & & & a''_6 & b''_6 & 0 & \\
 & & & & & 0 & b''_4 &
\end{pmatrix}
$$

Write a code that batch resolve tridiagonal systems of any size $d \leq 1024$ using the PCR algorithm.

Without going through the details of Householder tridiagonalization that can be found in [2], the basic ingredient is the Householder matrix H whose expression, for some vector u different from the zero vector, is given by

$$H = I - uu^t/b, \quad b = u^t u/2. \tag{2}$$

Considering one matrix A among $(A_n)_{1\leq n\leq N}$, the idea then is to choose the right vectors u_d,..., u_3 associated to H_d,..., H_3. The product of these matrices yields to the orthogonal matrix $Q = H_d H_{d-1}...H_3$ and their successive applications on A provide: $U = Q^t AQ = H_3^t...H_d^t AH_d...H_3$.

The first stage is to compute the tridiagonal form U through successive zeroing of the columns of matrix $A = (a_{i,j})_{i,j=1,...,d}$. This stage is processed at each step $i = d, ..., 3$ beginning by the vector

$$u_i^t = \left(a_{i,1}, ..., a_{i,i-1} \pm \sqrt{\sigma}, 0, ..., 0\right), \quad \sigma = \sqrt{a_{i,1}^2 + ... + a_{i,i-1}^2}, \tag{3}$$

then calculating the intermediary variables

$$b_i = \frac{u_i^t u_i}{2}, \quad , p_i = \frac{U_{i+1} u_i}{b_i} \quad , B_i = \frac{u_i^t p_i}{2b_i} \quad , q_i = p_i - B_i u_i \tag{4}$$

which allow us to set

$$U = U_3 \text{ with } U_i = U_{i+1} - q_i u_i^t - u_i q_i^t \text{ and } U_{d+1} = A. \tag{5}$$

Now that we have the tridiagonal form U, the second stage is to compute the orthogonal matrix Q defined by $Q = H_d H_{d-1}...H_3$. Besides, we remind that a Householder matrix H_i is completely specified by u_i. Consequently, during the first stage, the nonzero components of u_i are stored in the ith row of the memory space allocated for A and u_i/b_i in the ith column. Thus, the computation of Q is performed in the second stage using $Q = Q_d$ and the induction

$$Q_i = H_i Q_{i-1} \text{ for } i = 4, ..., d \text{ with } Q_3 = H_3. \tag{6}$$

By definition, Q_i is an identity matrix in the last i rows and columns and only its elements up to row and column $i-1$ need to be computed. These then overwrite u_i and u_i/b_i stored in A in the first stage.

As far as the first stage is concerned, in addition to the $d \times d$ memory space allocated for A we need $2d + 1$ extra memory space. The latter space is used to store the diagonal and the off-diagonal plus 1 value needed for the synchronization between phases where only one thread can be used and the other phases. Also, since p_i is of size i its components can be stored temporarily in the place of undetermined elements of the off-diagonal. Regarding q_i, it overwrites p_i in the off-diagonal.

Let us now take a look at the multi-threaded parts of the collaborative version. For the second stage, we can use $i-1$ collaborative threads that need synchronization only when the calculation of Q_i is finished. As for the first stage, the computations of p_i and q_i in (4) and the induction performed in (5) are all parallelized using $i-1$ threads. The other parts of this stage are executed using only one thread.

For instance, the code

```
for (j = 0; j <= l; j++) {
    sAds[nt + j*n + i] = sAds[nt + i*n + j] / h; //Store u/b in the ith column of A.
    g = 0.0f; //Form an element of A·u in g.
    for (k = 0; k <= j; k++)
     g += sAds[nt + j*n + k] * sAds[nt + i*n + k];
    for (k = j + 1; k <= l; k++)
     g += sAds[nt + k*n + j] * sAds[nt + i*n + k];
    sAds[nt + n2 + n + j] = g / h; //Form element of p in temporarily unused
}
```

is parallelized using

```
if (tidx <= l){
    h = sAds[nt1 + Qt];
    sAds[nt + tidx*n + i] = sAds[nt + i*n + tidx] / h; //Store u/b in the ith column of A.
    g = 0.0f; //Form an element of A·u in g.
    for (k = 0; k <= tidx; k++)
     g += sAds[nt + tidx*n + k] * sAds[nt + i*n + k];
    for (k = tidx + 1; k <= l; k++)
     g += sAds[nt + k*n + tidx] * sAds[nt + i*n + k];
    sAds[nt + n2 + n + tidx] = g / h; //Form element of p in temporarily unused
}
```

Implement a batch GPU version of the Householder tridiagonalization then use it with PCR for the solution of linear systems associated to symmetric positive definite matrices.

Project 4: Nested Monte Carlo and inference

Based on a nested Monte Carlo, the students have to simulate the price process $F(t,x,j)$ of a bullet option then train a Neural Network (NN) or at least a linear regression how to infer $F(t,x,j)$ for specific values taken by the triplet (t,x,j).

The price of a bullet option $F(t,x,j) = e^{-r(T-t)}E(X\big|S_t = x, I_t = j)$, $X = (S_T - K)_+ 1_{\{I_T \in [P_1,P_2]\}}$ with $I_t = \sum_{T_i \le t} 1_{\{S_{T_i} < B\}}$ and

- K, T are respectively the contract's strike and maturity
- $T_0 = 0 < T_1 < ... < T_M = T = T_{M+1}$ is a predetermined schedule
- barrier B should be bigger than S I_T times $\in \{P_1, ..., P_2\} \subset \{0, ..., M\}$
- r is the risk-free rate and σ is the volatility used in the Black & Scholes model

$$dS_t = S_t r dt + S_t \sigma dW_t, \quad S_0 = x_0.$$

The first step in this work is to make nested the Monte Carlo simulation developed in this course. Using nested Monte Carlo allows to simulate the value of $F(t,x,j)$ for various possible values of (t,x,j) instead of having only the simulation of $F(0,x_0,0)$ in a standard Monte Carlo. In addition to Monte Carlo code already given, the students have also to use the new `RNG.cu` and `RNG.h` files that prepare sufficient number of random number genrators for nested Monte Carlo.

Once the nested Monte Carlo code allows to simulate various realizations of $F(T_k,x,j)$ for $(k,x,j) \in \{0,1,...,M\} \times \mathbb{R}_+ \times \{0,1,...,\min(k,P_2)\}$, students should use these simulations to train on GPU a NN capable of generating learned realizations of the price F that have the same distribution as the one simulated by nested Monte Carlo.

Project 5: PDE simulation of bullet options

The students have to implement and compare Thomas algorithm to PCR for tridiagonal systems. Then, they have to simulate a PDE of a bullet option using Crank-Nicolson scheme based on either Thomas or PCR.

We refer to [3] and to Project 3 for a fair description of PCR. Regarding Thomas algorithm, as described in [2], it allows to solve tridiagonal systems:

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{7}$$

using a forward phase

$$c'_1 = \frac{c_1}{b_1},\ y'_1 = \frac{y_1}{b_1},\ c'_i = \frac{c_i}{b_i - a_i c'_{i-1}},\ y'_i = \frac{y_i - a_i y'_{i-1}}{b_i - a_i c'_{i-1}} \text{ when } i = 2, ..., n \tag{8}$$

then a backward one

$$z_n = y'_n,\ z_i = y'_i - c'_i z_{i+1} \text{ when } i = n-1, ..., 1. \tag{9}$$

Using Thomas method, write a kernel that solves various tridiagonal systems ($d \leq 1024$) at the same time, one system per block. Do the same thing for PCR then compare both methods.

Let $u(t, x, j) = e^{r(T-t)} F(t, e^x, j)$ where F is the price of a bullet option $F(t, x, j) = e^{-r(T-t)} E(X \big| S_t = x, I_t = j)$, $X = (S_T - K)_+ 1_{\{I_T \in [P_1, P_2]\}}$ with $I_t = \sum_{T_i \leq t} 1_{\{S_{T_i} < B\}}$ and

- K, T are respectively the contract's strike and maturity
- $T_0 = 0 < T_1 < ... < T_M = T = T_{M+1}$ is a predetermined schedule
- barrier B should be bigger than S I_T times $\in \{P_1, ..., P_2\} \subset \{0, ..., M\}$
- r is the risk-free rate and σ is the volatility used in the Black & Scholes model

$$dS_t = S_t r dt + S_t \sigma dW_t, \quad S_0 = x_0.$$

$u(t, x, j)$ is then the solution of the PDE

$$\frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2}(t, x, j) + \mu \frac{\partial u}{\partial x}(t, x, j) = -\frac{\partial u}{\partial t}(t, x, j)$$

$$\text{with: } \mu = r - \frac{\sigma^2}{2}$$

. The final and boundary conditions are:

$$\begin{aligned} \bullet\ u(T, x, j) &= \max(e^x - K, 0) \text{ for any (x, j)} \\ \bullet\ u(t, \log[K/3], j) &= \texttt{pmin} = 0 \\ \bullet\ u(t, \log[3K], j) &= \texttt{pmax} = 2K \end{aligned}$$

From now on we use notations $u_t(x, j) = u(t, x, j)$ and $u_{k,i} = u(t_k, x_i, j)$. Following Crank Nicolson scheme, we get

$$q_u u_{k,i+1} + q_m u_{k,i} + q_d u_{k,i-1} = p_u u_{k+1,i+1} + p_m u_{k+1,i} + p_d u_{k+1,i-1}$$

$$q_u = -\frac{\sigma^2\Delta t}{4\Delta x^2} - \frac{\mu\Delta t}{4\Delta x}, \quad q_m = 1 + \frac{\sigma^2\Delta t}{2\Delta x^2}, \quad q_d = -\frac{\sigma^2\Delta t}{4\Delta x^2} + \frac{\mu\Delta t}{4\Delta x}$$
$$p_u = \frac{\sigma^2\Delta t}{4\Delta x^2} + \frac{\mu\Delta t}{4\Delta x}, \quad p_m = 1 - \frac{\sigma^2\Delta t}{2\Delta x^2}, \quad p_d = \frac{\sigma^2\Delta t}{4\Delta x^2} - \frac{\mu\Delta t}{4\Delta x}$$

Noticing that:

$$\sum_{i=1}^{M} \mathbb{1}_{\{S_{T_i}<B\}} = \begin{cases} \sum_{i=1}^{M-1} \mathbb{1}_{\{S_{T_i}<B\}} & \text{if } S_{T_M} \geq B \\ \sum_{i=1}^{M-1} \mathbb{1}_{\{S_{T_i}<B\}} + 1 & \text{if } S_{T_M} < B. \end{cases}$$

Therefore we obtain the following backward induction:

for any $t \in [T_M, T[$, $\qquad u_t(x,j) = \mathbb{E}[(S_T - K)_+ | S_t = x]$

for any $t \in [T_{M-1}, T_M[$,

$$u_t(x,j) = \begin{cases} \mathbb{E}[(S_T - K)_+ \mathbb{1}_{\{S_{T_M} \geq B\}} \mid S_t = x] & \text{if } j = P_2 \\ \mathbb{E}[(S_T - K)_+ \mid S_t = x] & \text{if } j \in [P_1, P_2 - 1] \\ \mathbb{E}[(S_T - K)_+ \mathbb{1}_{\{S_{T_M} < B\}} \mid S_t = x] & \text{if } j = P_1 - 1 \end{cases}$$

for any $t \in [T_{M-k-1}, T_{M-k}[$, $k = M-1, ..., 1$,

$$u_t(x,j) = \begin{cases} \mathbb{E}[u_{T_{M-k}}(S_{T_{M-k}}, P_2)\mathbb{1}_{\{S_{T_{M-k}} \geq B\}} \mid S_t = x] & \text{if } j = P_2 \\ \mathbb{E}[u_{T_{M-k}}(S_{T_{M-k}}, P_k^1)\mathbb{1}_{\{S_{T_{M-k}} < B\}} \mid S_t = x] & \text{if } j = P_k^1 - 1 \\ \mathbb{E}\left[\begin{matrix} u_{T_{M-k}}(S_{T_{M-k}}, j)\mathbb{1}_{\{S_{T_{M-k}} \geq B\}} \\ + u_{T_{M-k}}(S_{T_{M-k}}, j+1)\mathbb{1}_{\{S_{T_{M-k}} < B\}} \end{matrix} \middle| S_t = x \right] & \text{if } j \in [P_k^1, P_2 - 1] \end{cases} \tag{10}$$

with $P_k^1 = max(P_1 - k, 0)$.

The figure below shows an example of how PDE's backward resolution algorithm (with M = 10, $P_1 = 3, P_2 = 8$) is deployed with time on the x-axis and the set of values of I_t in the ordinate. Write the pricing code and compare the execution time when using either Thomas or PCR.

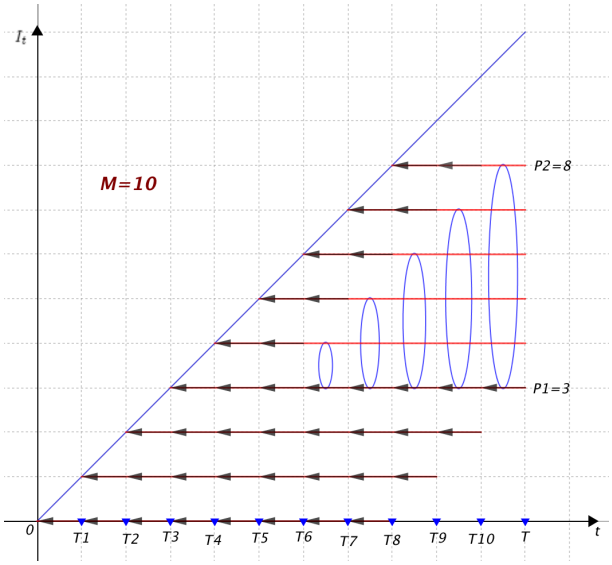


Figure 4: Backward induction scheme

References

[1] 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.

[2] 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.

[3] 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.