Explicite scheme, implicit and Crank-Nicolson

GPU programming

Laure Michaud - Tim Houdayé

May 4, 2022

Summary

1. Explicit Euler scheme
   - PDE_diff_k1 - A bad solution with a lot of access to the global memory
   - PDE_diff_k2 - A better solution using shared memory
   - Comparison of the two solutions
2. Implicit Euler scheme
3. Crank-Nicolson
4. Find the optimal values of x_0 and σ_0

Explicit scheme - A bad solution with a lot of access to the global memory

For this first question, we implement a way to compute option price using GPU with a lot of access to the memory. In order to put a price on the options, we use the explicit scheme technique and resolve for a fixed σ the equation:

$$u_{i,j} = p_u u_{i+1,j+1} + p_m u_{i+1,j} + p_d u_{i+1,j-1} \tag{1}$$

with i the space of time and j the space of x. In our kernel, the *for* loop will increment the time space, each threads will be an increment of the x space, and each block an increment of the sigma space.
We implement the following steps in PDE_diff_k1:

1. Compute all the coefficients μ, σ,p_u, p_m and p_d to solve the equation (1)
2. Initialize a **static allocation** of the shared memory with size NTPB in a variable named **result**
3. Process the **boundary conditions** separately by identifying the threadIdx.x equal to 0 and NTPB-1 where the price is equal respectively to p_{min} and p_{max}. Otherwise, we compute the price using equation (1).
4. **Synchronize** all threads and overwrite the data of **pt_GPU** with those of **result**.

Explicit scheme - A better solution using shared memory

For this question, we wanted to put the *for* loop on dates into the kernel. In order to optimize the access to the memory we also use **dynamic allocation**. We implement the following steps in PDE_diff_k2:

1. Initialize the **dynamic memory** of size *NTPB*sizeof(float)* in variable using the command *extern __shared__ float result_block[];* **result_block**, and compute the coefficients as in the first kernel.
2. Compute the price of the option with equation (1) using data in **pt_GPU**, and stack them in **result_block**. Then, wait for all the the thread to **synchronize**.
3. Start the *for* loop on dates from 1 to N, using the counter **i_date**
   - Initialize a variable **result_thread** which is a **static allocation** of the shared memory using the command *__shared__ float result_thread[NTPB];*. This variable will stock the result of the price computation before overwriting the results of **result_block**
   - Compute the price of the option for the time **i_date** and stock this result in **result_thread**
   - Wait for all the threads to end, and overwrite the result of **result_block** with those of **result_thread**
   - Increment **i_date** of 1
4. Once the *for* loop on date finish, we **synchronize** the threads, and overwrite the data of **pt_GPU** with those of **result_block**

Comparison of the two solutions

Comparison of the kernel 1 and the kernel 2		
Parameters	Price Explicit	Black & Scholes
$S_0 = 100, \sigma = 0.2$	3.751771	3.753428
$S_0 = 100, \sigma = 0.3$	7.228672	7.217876
$S_0 = 141.4214, \sigma = 0.3$	1.010791	1.012503

Comparison of the kernel's time execution	
Solutions	Time executions
Global memory (kernel 1)	45.498302 ms
Shared memory (kernel 2)	4.853760 ms

For this question, we try to compute option price using an implicit scheme methods. For a fixed σ, we resolved the following equations:

$$u_{i+1,j} = q_u u_{i,j+1} + q_m u_{i,j} + q_d u_{i,j-1} \tag{2}$$

This system can be written in a matrix form such as $\mathrm{T}\mathrm{X} = \mathrm{Y}$. With T a tri-diagonal matrix, Y the matrix of u_{i+1} and X the matrix of u_i.

To compute this type of methods, we need to **invert a matrix**. So, we used **Thomas' algorithm** and a parallel cyclic reduction (i.e PCR) technique. We implement the following steps in PDE_diff_k3:

1. Initialize σ and μ coefficients and allocate a dynamic memory of size NTPB in a variable named **sy**.
2. Initialize 4 statics allocations of shared memory of size NTPB for each, named **sa**, **sd**, **sc**, and **sl**. With **sa**, the lower diagonal, **sd** the mid diagonal, and **sc** the upper diagonal of the tri-diagonals matrix T.
3. Compute the coefficients q_d, q_m and q_u, and the prices contain in **pt_GPU** in **sy**. Correct the boundaries conditions of the x space, and synchronize the theads.
4. Start the *for* loop from 0 to N-1 using the counter **i_date**
   - For each threads, stack q_d, q_m and q_u in **sa**, **sd**, **sc** respectively, and add the threadIdx.x counter in **sl**.
   - **Synchronize** the threads and launch the **PCR_d** methods with the following variables **sa**, **sd**, **sc**, **sy**, **sl**, **NTPB**.
   - Compute the boundaries conditions, **synchronize** and increment **i_date**
5. **Synchronize** and overwrite the data of **pt_GPU** with those of **sy**.

In this question, we implement the Crank Nicolson methods to solve PDE. For a fixed σ, we resolved the following equations:

$$p_u u_{i+1,j+1} + p_m u_{i+1,j} + p_d u_{i+1,j-1} = q_u u_{i,j+1} + q_m u_{i,j} + q_d u_{i,j-1} \tag{3}$$

This system can be written in the same matrix form as the implicit scheme methods. We also need to invert a matrix. We implement the following steps in PDE_diff_k4:

1. Initialize all the coefficients: σ, μ, q_u, q_m, q_d, p_u, p_m, p_d, and allocate a dynamic shared memory of size NTPB in a variable named **sy**.
2. Construct **sa**, **sd**, **sc**, **sl** and allocate them, as in the implicit scheme.
3. Compute the left part of equation (3) using **sy** and p_u, p_m, p_d.
4. **Synchronize** and launch the first **PCR_d** method.
5. Start the *for* loop from 1 to N-1 using the counter **i_date**
   - Initialize a variable **result_thread** which is a **static allocation** of the memory.
   - Redo all the previous steps
   - **Synchronize** and increment **i_date** of 1
6. **Synchronize** and overwrite the data of **pt_GPU** with those of **sy**.

Comparison of the two solutions

Comparison of the kernel 3 and the kernel 4			
Parameters	Price Implicit	Price Crank Nicolson	Black & Scholes
$S_0 = 100, \sigma = 0.2$	42.236	38.789	3.753428
$S_0 = 100, \sigma = 0.3$	125.289	121.61	7.217876
$S_0 = 141.4214, \sigma = 0.3$	175.949	144.88	1.012503

Comparison of the kernel's time execution	
Solutions	Time executions
Implicit (kernel 3)	360.748 ms
Crank Nicolson (kernel 4)	373.513 ms

Find the optimal values of x_0 and σ_0

We will share two variables **distance_values** and **dx_values** which will contain respectively the distance $|u(x_0, \sigma_0) - u_0|$ and the values of x_0. The value of σ_0 is given by $\sigma_{min} + d\sigma \times blockIdx$.
We implement the following steps in Optimal_k1:

1. Firstly, we calculate these two vectors for all the thread. We synchronize the threads.
2. Then, we will perform a **while loop (diad division) per block** in order to find the couple $(x_0, u(x_0, \sigma_0))$ which minimises the absolute difference. We synchronize the threads in the while loop at the end of each iteration and also at the end of the while. We fetch the minimum value using the pt_GPU.
3. We perform a last **while loop on the block** in order to find the minimum among all the minimum given by blocks.
4. We add two *printf* at the end to return values of x_0, σ_0

We test our methods for $S_0 = 100$, $\sigma_0 = 0.2$ which corresponds to $u_0 = 3.751771 * e^{rT} = 4.146348$ and find the couple (100.00,0.2).