High-Performance Computing in financial mathematics : Predicting european call option prices under the exponential O.U. model

Tristan Chenaille, MAIN5 Engineering Student, 2024

Below is the table of contents :

Abstract

Let us recall the basics. A European (EU) *call option* is a financial derivative that gives the holder the right (but not the obligation) to buy an underlying asset (like a stock, commodity, or currency) at a predetermined price (called the *strike price* K) on a pre-agreed date (called the *maturity* T).
The *premium* is the cost of purchasing the call option, paid upfront by the buyer to the seller. Here, we focus on the specific case where the premium is null in order to make things as clear as possible.

The *payoff* of a call option depends on the relationship between the spot price of the underlying asset at maturity (S_T) and the strike price (K) as follows :

$$\text{Payoff} \;=\; \max(S_T - K\,,\,0)$$

Indeed, if $S_T > K$, the option is said to be "in the money", and the holder exercises the option to buy the asset at the lower strike price K. The payoff is $S_T - K$.
Otherwise, if $S_T \leq K$, the option is "out of the money", and the holder lets the option expire without exercising it. The payoff is 0.

The *price of a call option* (referred to as C) is the discounted risk-neutral expectation of its payoff at maturity. Here, we will set the risk-free rate r to 0 to keep things clear. Thus, we have :

$$C \;=\; \exp(-r\,T)\;\mathbb{E}(\text{Payoff}) \;=\; \exp(0)\;\mathbb{E}(\max(S_T - K\,,\,0)) \;=\; \mathbb{E}[(S_T - K)_+]$$

Nowadays, numerous mean-reverting stochastic volatility models are utilized in financial mathematics to describe asset price dynamics. Here, we focus on the exponential Ornstein–Uhlenbeck (O.U.) volatility model, in which the volatility process follows an Ornstein–Uhlenbeck-type dynamics. Specifically, this model is characterized by the following equations:

$$\begin{aligned} dS_t \;&=\; S_t\;(r\,dt + \exp(Y_t)\,dW_t) \;=\; S_t\;\exp(Y_t)\,dW_t \\ dY_t \;&=\; \alpha\,(m - Y_t)\,dt + \beta\,d\hat{Z}_t \\ \hat{Z}_t \;&=\; \rho\,W_t + \sqrt{1-\rho^2}\,Z_t \end{aligned}$$

Where:

- $S_0 = 1$ and $Y_0 = \log(0.1)$ are the spot values
- r is the risk-free interest rate (recall that we assumed $r = 0$, which simplifies the dSt formula)
- α is the mean reversion rate of the volatility
- m is the long-term log-volatility
- β is the volatility of the log-volatility
- W_t and Z_t are independent Brownian motions (also known as Wiener processes)
- $\hat{Z}_t$ is another Brownian motion depending on both W_t and Z_t through ρ
- ρ is the correlation coefficient between $\hat{Z}_t$ and W_t
- t is the time variable ranging from start ($t = 0$) to maturity ($t = T$).

For further details on this exponential O.U. model, we refer the reader to this research paper.

In this project, the goal is to predict the price of a call option for any maturity, strike, and a range of model parameters α, β, m, ρ, and Y_0. To achieve this, we will start by writing code to perform these predictions thanks to the *Monte-Carlo* technique. Then, we will parallelise this code with `CUDA` to speed the execution up on an `NVIDIA GPU`. Once this is done, we will nest the Monte-Carlo simulations and store the results in a `.csv` file to create a dataset. Finally, we will use this file as a training set for a neural network that will learn and generalize the relationship between option prices and the various model parameters across different maturities and strikes.

Monte-Carlo methods : concept and relevance

Monte-Carlo methods, also known as Monte-Carlo experiments, are a type of algorithms relying on repeated random samplings to estimate mathematical functions and mimic the operations of complex systems. The method derives its name from the Monte Carlo Casino in Monaco, as mathematician Stanislaw Ulam, one of its primary developers, was inspired by his uncle's gambling habits.

Rather than distracting the reader with mathematical details, let us demonstrate the principle using a simple example. Let us consider the *Salagou* Lake in southern France. As shown on the map below, it has a very irregular shoreline :

Therefore, measuring its exact area would be a very difficult task. It would imply mapping every curve and inlet along the boundary, plus some very complex computations afterwards. This would require way too much time and resources.

However, the Monte Carlo method offers a simple yet highly efficient alternative. Imagine the picture above as a poster stuck to the wall of your room. It represents the lake that is 5 kilometers long and 2 kilometers wide, so the poster's dimensions would be 5 meters by 2 meters. Now imagine randomly (without aiming) throwing a lot of darts at this poster and keeping track of two counts :

- C_1, the number of darts that landed onto the poster
- C_2, the number of darts that landed onto the part of the poster representing the lake's surface

The more darts you throw, the better the ratio (C_2/C_1) will approximate the proportion of space the lake takes on the poster. Because of the law of large numbers - which is very intuitive - you know for sure that the approximation will be irrelevant for a dozen of throws. For a thousand, it will be quite good however. And for a million, it will be very accurate.

Instead of making you throw a million darts, I asked my computer to run the experiment (by simulating only a poster and a lake) and to stop when its C_1 (which corresponds to his throw count because no wall means he cannot miss the poster) reaches $1\,000\,000$. At the moment he stopped, his C_2 count was $748\,253$. So his (C_2/C_1) ratio was $(748\,253/1\,000\,000)$, which means the lake takes approximately $74,8253\,\%$ of the poster's space. However, the area represented on the poster has a true scale of 5 kilometers $\times$ 2 kilometers $=$ 10 square kilometers. $74,8253\,\%$ of 10 square kilometers is $7,48253$ square kilometers. Consequently, our Monte-Carlo simulation for a million throws estimates the area of the Salagou Lake as $7,48253$ square kilometers. Google tells us its real superficy is $7,5$ square kilometers, so the approximation here is really good since it differs from the real value by only $\sim 0,234\,\%$.

In general, the Monte-Carlo technique can be used for problems that involve high-dimensional integration, optimisation or stochastic processes. Its probabilistic approach allows for providing approximate solutions to problems that are otherwise intractable or too complex to analyze mathematically.

In the context of this paper, the complex stochastic differential equations of the exponential O.U. model lack closed-form solutions. This makes the mathematical problem too difficult to solve analytically, hence the cruciality of the Monte-Carlo method, which can be used to compute nice estimations when applied on a discretised Euler scheme that has small enough time steps.

Sequential `C` implementation of Monte-Carlo for one case

Although the principle remains the same, the estimation of a call option price is somewhat more elusive than that of a surface area. This is why we will first implement the algorithm sequentially (focusing on one single case) before addressing its parallelisation.

A trivial instance of the Euler discretisation scheme

Let the Euler scheme update the asset price S and the log-volatility Y at each time step as follows :

$$S_{t+\Delta t} \;=\; S_t \;+\; r\, S_t\, \Delta t \;+\; \exp(Y_t)\, S_t\, \sqrt{\Delta t}\, G_1 \;=\; S_t \;+\; \exp(Y_t)\, S_t\, \sqrt{\Delta t}\, G_1$$
$$Y_{t+\Delta t} \;=\; Y_t \;+\; \alpha\,(m - Y_t)\,\Delta t \;+\; \beta\,\sqrt{\Delta t}\;(\rho\, G_1 \;+\; \sqrt{1-\rho^2}\, G_2)$$

Where:

- Δt is the time step of the Euler discretisation scheme, say there are 1000 steps
- G_1 and G_2 are independent standard normal variables
- r is the risk-free interest rate (recall that we assumed $r = 0$, which simplifies the $S_{t+\Delta t}$ formula)
- S_t, Y_t, α, m, β, ρ are the variables introduced in the previous **Abstract** section.

To maintain clarity, we make the following assumptions regarding the current particular case :

- $\alpha = 1$, indicating that the volatility reverts to a mean at a moderate speed, which is close to the average trend on financial markets.
- $m = 0.1$, so $\exp(m) \approx 1.105$, reflecting market conditions where volatility is expected to stay consistently low on average (e.g., stable markets or low-risk assets).
- $\beta = 0.1$, implying that the volatility itself doesn't experience wild swings, which is realistic for many financial assets. This choice helps maintain stability in Monte-Carlo simulations and reflects the empirical observation that volatility fluctuations are typically smaller than price ones.
- $q = -0.5$. This negativity emphasizes the leverage effect causing volatility to increase during price drops. This is a pretty usual value for financial markets.

Additionally, we want to estimate the specific case $\mathbb{E}[(S_1 - 1)_+]$ of $\mathbb{E}[(S_T - K)_+]$, meaning :

- The maturity T equals 1, so the size of each of the 1000 time steps Δt is $(1/1000)$
- The strike price K equals 1 as well.

To summarize, we have :

- $T = 1$ and $\Delta t = \frac{1}{1000}$
- $\alpha = 1$ and $\beta = 0.1$
- $m = 0.1$
- $\rho = -0.5$
- $K = 1$
- $S_0 = 1$ and $Y_0 = \log(0.1)$

Implementation of `v1-0`

First off, we define a function `generate_normal()` to generate G_1 and G_2. This function uses the original *Box-Muller* transform, which tends to be more optimal than its *Marsaglia* polar version because it involves no rejection, and therefore no additional iterations :

```
// Box-Muller method for generating standard normal random variables
float generate_normal() {
    float u1, u2;
    // Generate two independent uniform random variables in [0,1]
    u1 = ((float) rand() + 1.0f / (RAND_MAX) + 1.0f);
    u2 = ((float) rand() + 1.0f / (RAND_MAX) + 1.0f);
    // Apply the Box-Muller transform
    return sqrtf(-2.0f * logf(u1)) * cosf(2.0f * M_PI * u2);  }
```

On to the `main()` now, we define the model parameters and the simulation variables :

```
// Model parameters
float alpha = 1.0f, m = 0.1f, beta = 0.1f, rho = -0.5f;
float S0 = 1.0f, Y0 = logf(0.1f);
float K = 1.0f, T = 1.0f;
float dt = 0.001f;
int steps = (int)(T/dt) ;
// Simulation variables
float S, Y, payoff, sum_payoff = 0.0f;
```

Then, we set up the Monte-Carlo simulation loop :

```
// Monte-Carlo loop with N different simulations (aka trajectories)
for (int i = 0; i < N; i++) {

    // Initial conditions
    S = S0;
    Y = Y0;

    // Simulation on the Euler scheme for j from 0 to last_step
    for (int j = 0; j < steps; j++) {
        // Generation of two normal standard random variables
        float G1 = generate_normal();
        float G2 = generate_normal();
        // Updating St and Yt
        S = S + expf(Y) * S * sqrtf(dt) * G1;
        Y = Y + alpha * (m - Y) * dt + beta * sqrtf(dt)
        * (rho * G1 + sqrtf(1.0f - rho * rho) * G2);
    }

    // Evaluating the payoff of the finished trajectory
    payoff = ( S-K > 0.0f ? S-K: 0.0f );

    // Updating the sum of simulated payoffs
    sum_payoff += payoff;
    }
```

Please note that this Monte-Carlo simulation code loop is part of the `main()`, and is not defined as a function because of the constant parameters we consider for now. The only user-defined variable is `N`, which gets retrieved by a simple `scanf()`. To gain better understanding, download the project's code. To conclude, we print the estimated payoff :

```
float option_price = sum_payoff / N; // <--- Averaging
printf("Estimated price of the call option : %f\n", option_price);
```

Optimisation : implementation of `v1-1`

Some improvements to this `v1-0` code can be done.

First off, since the models parameters are fixed, there is no need to compute operations on these at each iteration. It is useless and can become costly for a large number of iterations. To fix this, one can start by rewriting the Euler scheme formulas by substituting parameters with the assumed values :

$$S_{t+\Delta t} = S_t + \exp(Y_t)\, S_t \sqrt{\Delta t}\, G_1 = S_t\,(1 + \exp(Y_t)\sqrt{\Delta t}\, G_1)$$

and

$$\begin{aligned}
Y_{t+\Delta t} &= Y_t + \alpha\,(m - Y_t)\,\Delta t + \sqrt{\Delta t}\,(\rho\, G_1 + \sqrt{1-\rho^2}\, G_2) \\
&= Y_t + 1\,(0.1 - Y_t)\,(1/1000) + 0.1\sqrt{1/1000}\,(-0.5\, G_1 + \sqrt{1-(-0.5)^2}\, G_2) \\
&= Y_t + (0.1 - Y_t)\,0.001 + 0.1\sqrt{1/1000}\,(-0.5\, G_1 + \sqrt{0.75}\, G_2) \\
&= Y_t - 0.001\, Y_t + 0.0001 + 0.1\sqrt{1/1000}\,(-0.5\, G_1 + \sqrt{0.75}\, G_2) \\
&= 0.999\, Y_t + 0.0001 + 0.1\sqrt{1/1000}\,(-0.5\, G_1 + \sqrt{0.75}\, G_2) \\
&= 0.999\, Y_t + 0.0001 - 0.1\sqrt{1/1000}\,0.5\, G_1 + 0.1\sqrt{1/1000}\sqrt{0.75}\, G_2 \\
&= C_1\, Y_t + C_2 - C_3\, G_1 + C_4\, G_2
\end{aligned}$$

Where:

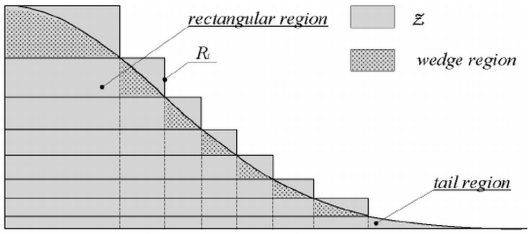
- $C_1 = 0.999$
- $C_2 = 0.0001$
- $C_3 = 0.1\sqrt{1/1000}\,0.5$
- $C_4 = 0.1\sqrt{1/1000}\sqrt{0.75}$

We now have to update the code consequently. Here are the new equations to update S_t and Y_t at each time step inside the MC loop :

```
// Precomputing constants
float sqrtdt = sqrtf(dt); float C1 = 0.999f; float C2 = 0.0001f;
float C3 = 0.1f*sqrtf(1.0f/1000.0f)*0.5f;
float C4 = 0.1f*sqrtf(1.0f/1000.0f)*sqrtf(0.75f);

// Simulation on the Euler scheme for j ranging from 0 to max_step
for (int j = 0; j < steps; j++) {
    // Generation of two normal standard random variables
    float G1 = generate_normal();
    float G2 = generate_normal();
    // Update of St and Yt
    S = S * (1.0f + expf(Y) * sqrtdt * G1);
    Y = C1 * Y + C2 - C3 * G1 + C4 * G2 ;
}
```

To further optimise the code, we now turn our attention to the generation of random standard normal variables. In each of the N Monte Carlo simulations, the Euler scheme is iterated 1000 times, with each iteration requiring two calls to the `generate_normal()` function. Each call to this function involves 4 multiplications and 2 divisions on `float` precision floating-point numbers. Consequently, the total computational cost amounts to $N \times 2 \times 4 = 8000N$ multiplications and $N \times 2 \times 2 = 4000N$ divisions. Such a computational burden may cause a bottleneck.

Fortunately, our current *Box-Muller-Transform* generation algorithm happens to have a far better optimised alternative for a large number of generations. It is called the *Ziggurat* algorithm. This method derives its name from the bounding of the probability density function using rectangles, resembling the structure of ancient mesopotamian ziggurats.



The *Ziggurat* algorithm for generating random standard normal variables is based on bounding one side of the density function (due to symmetry) with rectangles. These rectangles are precomputed and stored in a table, which also indicates whether each rectangle lies entirely under the density function or intersects it. A random point is then selected within a rectangle.

- If the rectangle lies entirely below the density function, the value is accepted without any computation.
- Otherwise, if the rectangle intersects the density function (rare due to most rectangles being below it), additional computation determines if the value falls within the valid region (the "wedge" zone depicted right above). If so, it is accepted; otherwise, it is rejected.

While it requires memory for storing the rectangles table, it significantly reduces computational cost.

A highly optimised version of the *Ziggurat* is implemented in the `GNU Scientific Library (GSL)`, which is native to most `UNIX` distributions. It can be installed on Windows using the `vcpkg` tool.

One has to include the required libraries :

```
// all others #include ...
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

Then, the initialization has to be done at the beginning of the `main()` :

```
// Initializing the GSL random number generator
gsl_rng *rng = gsl_rng_alloc(gsl_rng_mt19937);
gsl_rng_set(rng, (unsigned long)time(NULL));
```

Let us call the `GSL` ziggurat for G_1 and G_2 inside the Euler scheme loop :

```
float G1 = gsl_ran_gaussian_ziggurat(rng, 1.0);
float G2 = gsl_ran_gaussian_ziggurat(rng, 1.0);
```

And finally, right after the Monte-Carlo loop, the `GSL` random number generator has to be freed :

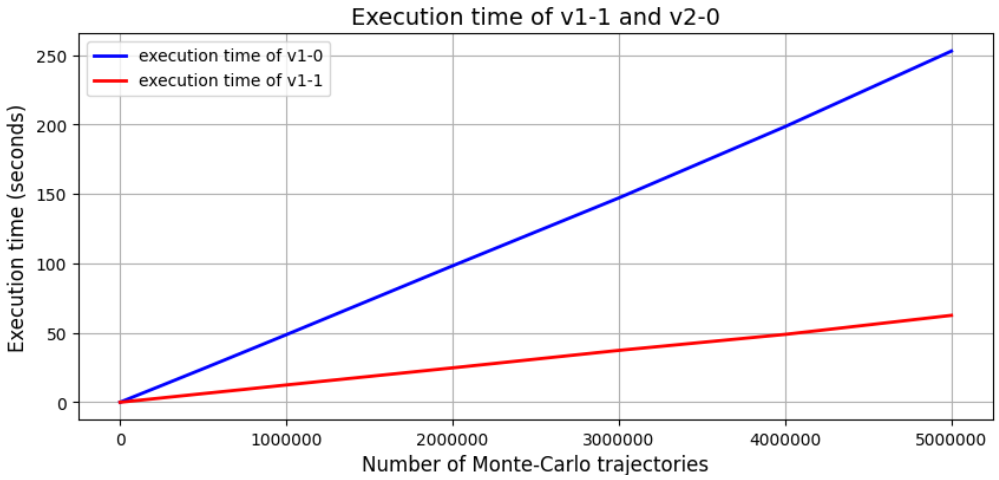
```
// Freeing the GSL random number generator
gsl_rng_free(rng);
```

The optimised code implementing both $C_{\{1,2,3\}}$ and Ziggurat optimisations is named `v1-1.c` .

Performance comparison between `v1-0` and `v1-1`

Let us now check if the improvements have a significant impact on performance.

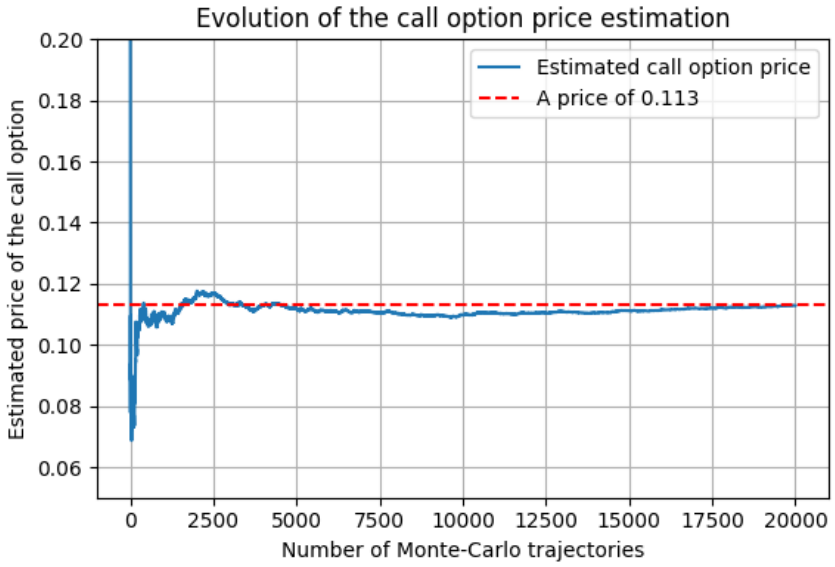
Here is a benchmark made on an `E-core` of an `Intel i9-13980HX CPU` :



Neither `v1-0` nor `v1-1` is parallelised, hence the quasi-linearity of their execution times. Notably, `v1-1` is approximately 4 times faster than `v1-0`, a substantial improvement, particularly for a large number of trajectories. For example, when performing 5 000 000 Monte-Carlo simulations, `v1-0` requires 253 seconds to execute, whereas `v1-1` completes the task in only 62 seconds. These optimisations in the sequential code, though seemingly unnecessary for a future CUDA implementation, are crucial as they provide an optimised baseline for accurately evaluating the future parallelisation speedup.

Monte-Carlo convergence

One can check that the code works properly by displaying the evolution of Monte-Carlo's estimation :



This behavior is characteristic of the evolution curve of an MC estimation. The estimated call option price converges to approximately 0.113, a reference value that will help us ensure the results remain consistent across parallel implementations. Including additional trajectories would allow for a more precise approximation of the lower decimal places of the true call option price.

Parallelised CUDA implementation of Monte-Carlo for one case

With the sequential implementation now optimised, our focus can shift to parallelisation. Given that all Monte-Carlo trajectories are independent simulations and that their number is substantial, NVIDIA CUDA is the most suitable strategy. This is because GPUs possess significantly more cores than CPUs. GPU cores are individually less powerful, but there is no consequence to it as calculating a Monte-Carlo trajectory is not a computationally-intensive process.

Implementation of v2-0

First off, we adapt the #include section as follows :

- We retain basic libraries to ensure maximum compatibility on Windows distributions.
- We include two CUDA libraries : the runtime one, which allows to display the current GPU specifications, and the cuRAND one, which is about *RNG* (Random Number Generation).
- We drop GSL dependencies : even though gsl_ran_gaussian_ziggurat() function was highly efficient in v1-1, cuRAND embeds curand_normal2() which is deeply optimised for NVIDIA GPUs.

Here is the updated code :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>
```

Followingly, we add a CUDA function that prints a detailed message upon catching an error :

```
// Function that catches the CUDA error
void testCUDA(cudaError_t error, const char* file, int line) {
    if (error != cudaSuccess) {
        printf("There is an error in file %s at line %d : %s\n",
        file, line, cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }
}

// Macro to automatically input the current file name and line
#define testCUDA(error) (testCUDA(error, __FILE__ , __LINE__))
```

Subsequently, we implement a testing CUDA function to print the main-GPU properties :

```
void printMainGpuProperties() {
    int deviceID; cudaGetDevice(&deviceID);
    cudaDeviceProp prop;cudaGetDeviceProperties(&prop, deviceID);
    // Printing (skipping detailed prop fields for clarity)
    printf("\n===== Main CUDA GPU device %d : %s =====\n", /* ... */);
    printf("Total global memory : %lu bytes\n", /* ... */);
    printf("Max threads per block : %d\n", /* ... */);
    printf("Max threads per multiprocessor : %d\n", /* ... */);
    printf("Number of streaming multiprocessors : %d\n", /* ... */);
    printf("Max grid size : (%d, %d, %d)\n", /* ... */);
    printf("Warp size : %d\n", /* ... */);
    printf("Maximum simultaneous threads : %d\n", /* ... */);
    printf("============================\n");
}
```

Then, we define a Monte-Carlo *kernel* (a function running on the GPU) to replace the sequential code :

```
__global__ void monte_carlo_kernel(float *results, int N, int steps,
    float sqrtdt, float S0, float Y0, float K, float C1, float C2,
    float C3, float C4) {

    // Determining global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x ;

    // Setting up the random number generator for the current thread
    curandState state;
    curand_init((unsigned long long)clock64() + idx, idx, 0, &state);

    // The current thread processes all of its assigned trajectories
    if (idx < N) {

        // Variables and initial conditions
        float S = S0, Y = Y0, G1, G2 ; float2 randPair ;

        // Perform the simulation
        for (int j = 0; j < steps; j++) {
            randPair = curand_normal2(&localState);
            G1 = randPair.x; G2 = randPair.y;
            S = S * (1.0f + expf(Y) * sqrtdt * G1);
            Y = C1 * Y + C2 - C3 * G1 + C4 * G2 ;
        }

        // Compute payoff and store in global memory
        results[idx] = ( S-K > 0.0f ? S-K: 0.0f );
    }
}
```

Let us examine two key points here :

- The variable `idx` serves as the unique identifier for each thread, ranging from 0 to `totalThreads`-1. Consequently, if `N` is not perfectly divisible by `totalThreads`, some threads will have indices exceeding `N`-1. Taking these indices into account would lead to computing unwanted trajectories and result in out-of-bounds errors when attempting to store in the `results[]` array. To prevent this, the `if` condition ensures `idx` $<$ `N`.
- The function `curand_init()` initializes an RNG state for each thread. By incorporating `+idx` into the seed and using `idx` as the sequence parameter, it ensures unique initial states for each thread. Every `curand_normal2(&state)` call updates the internal state of the RNG for the next generation. Furthermore, the use of `clock64()` in the seed guarantees variation across program executions. All in all, we have a reliable pseudo-random RNG.

Then, here is how the kernel is called from the `main()` function :

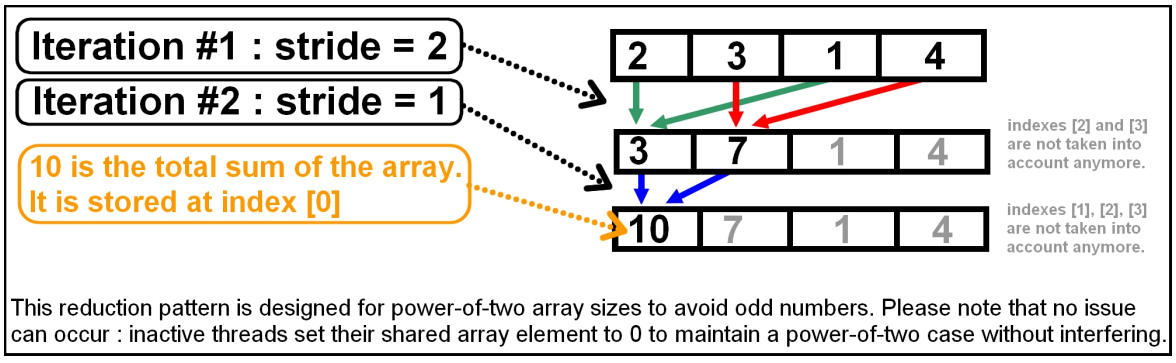
```
// Launching the Monte-Carlo CUDA kernel
const int BLOCK_SIZE_MC = 32;
const int GRID_SIZE_MC = (N + BLOCK_SIZE_MC - 1) / BLOCK_SIZE_MC;
monte_carlo_kernel<<<GRID_SIZE_MC, BLOCK_SIZE_MC>>>(/*parameters*/);
```

The kernel is launched with a CUDA configuration defined by `BLOCK_SIZE_MC` and `GRID_SIZE_MC`.
In this model, each thread computes exactly one trajectory, which is easy since they are independent.
Block size (`BLOCK_SIZE_MC`) is set to the `warp` size (32). $32 \times n$ sizes feature neglectable perfs changes.
To ensure all N Monte-Carlo trajectories are covered, even when N is not a multiple of `BLOCK_SIZE_MC`, `GRID_SIZE_MC` is computed with a formula rounding up the division to the nearest greater integer.
For a large N, generating far more threads and blocks than the maximum that can run simultaneously poses no issue, as CUDA efficiently manages idle threads/blocks through its optimized queuing system.

Additionally, we define a reduction kernel to reduce bottlenecks by minimizing global memory access. For highly demanding simulations ($N > 10^9$), this reduction can be implemented inside the Monte-Carlo kernel to take advantage of the variables that are still loaded in the cache at MC runtime :

```
// Reduction done outside of the MC kernel for clarity
__global__ void reduction_kernel(float* input, float* output, int n) {

    // Declaring a shared memory array for the block
    extern __shared__ float sharedData[];

    // Calculating thread ID and block ID
    int idx = threadIdx.x;
    int globalIndex = blockIdx.x * blockDim.x + idx;

    // Loading data into shared memory
    sharedData[idx] = (globalIndex < n) ? input[globalIndex] : 0.0f;
    __syncthreads();

    // Performing reduction within the block
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
        if (idx < stride) {
            sharedData[idx] += sharedData[idx + stride];
        }
        __syncthreads();
    }

    // Writing the block result to the output array
    if (idx == 0) { output[blockIdx.x] = sharedData[0]; }
}
```

- A dynamically-allocated shared memory array (`sharedData[]`) is declared for each block.
- Each thread loads one element from global memory (`input`) into its block shared memory. If a thread's global index exceeds the size of the array, it initializes its contribution to `0.0f`. The `__syncthreads()` call ensures all threads complete this step before moving on to the next one.
- In the reduction loop, the `stride` is initially set to half the block size, and is halved in each iteration until it reaches 1. At each of these steps, only threads with indices less than the `stride` are active and sum their value in `sharedData[idx]` with the value of their "partner" at `sharedData[idx+stride]`. This process reduces the size of the array being processed by half in each iteration, with synchronization barriers preventing race conditions. Consequently, when the reduction has completed, the valid final sum for the block is stored in `sharedData[0]`. To better understand, here is a simple diagram that represents this stride-based reduction model:



- Threads with index 0 write the final sum of their block to the global output array.

Then, here is how the kernel is called from the `main()` function :

```
// Setting up the launch configuration of the Reduction CUDA kernel
const int BLOCK_SIZE_R = 512;
const int GRID_SIZE_R = (N + BLOCK_SIZE_R - 1) / BLOCK_SIZE_R;

// Launching the Reduction CUDA kernel
reduction_kernel<<<GRID_SIZE_R, BLOCK_SIZE_R,
BLOCK_SIZE_R * sizeof(float)>>>(d_results, d_output, N);

// Copying the reduced results array to host
cudaMemcpy(h_output, d_output, GRID_SIZE_R * sizeof(float),
cudaMemcpyDeviceToHost);
```

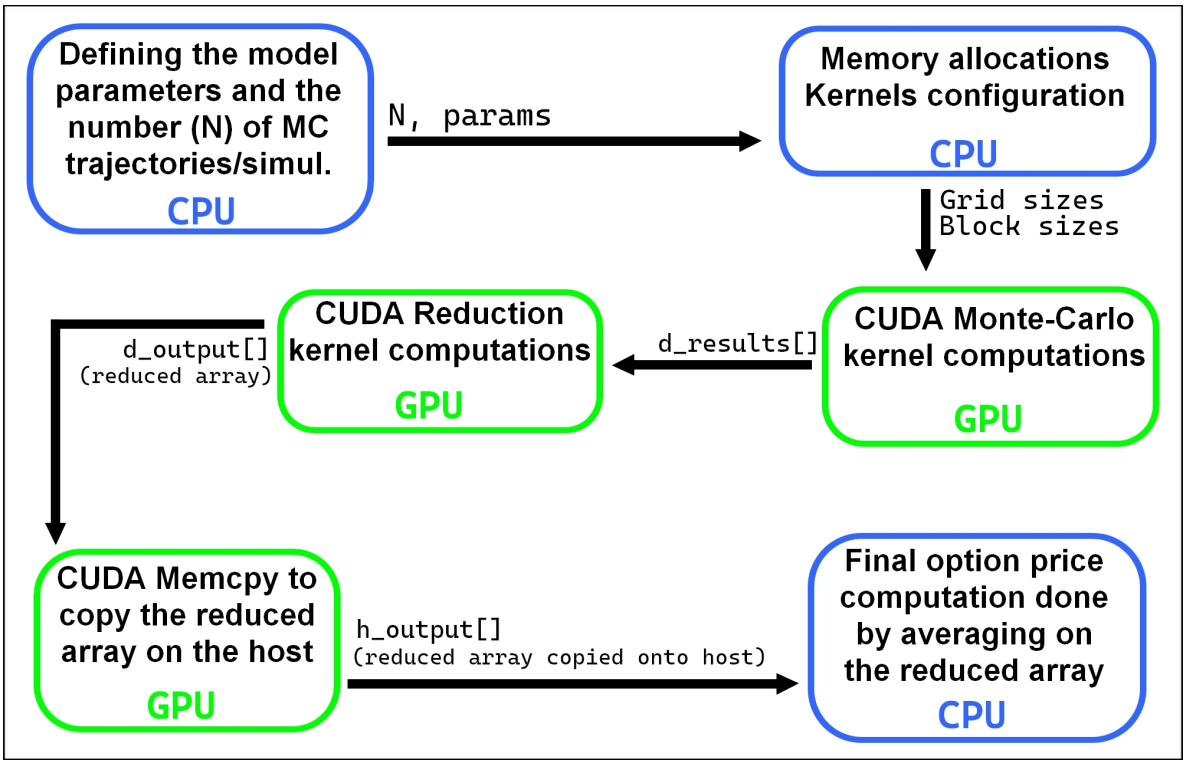
A brief explanation of this code follows.

- `GRID_SIZE_R` is defined with the same logic as in the Monte-Carlo kernel, while `BLOCK_SIZE_R` is set to 512, which is what works best on a `NVIDIA RTX 4070 Laptop GPU`.
- The reduction kernel is called with `d_results` as the input array (which has been determined earlier by the Monte-Carlo kernel) and `d_output` as the reduced output array.
- `cudaMemcpy` then copies the `d_output` array from the `GPU` to the `h_output` array on the `CPU`.

Once `d_output` has been copied to `h_output`, the final option price is computed on the host :

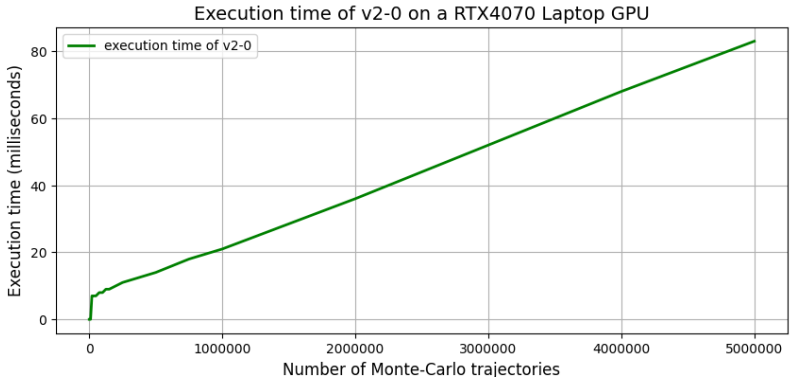
```
// Computing final option price by averaging
    float sum_payoff = 0.0f;
    for (int i = 0; i < GRID_SIZE_R; i++) {
        sum_payoff += h_output[i];
    }
    float option_price = sum_payoff / N;
```

Ultimately, the temporal diagram below briefly illustrates the steps of how `v2-0` operates :



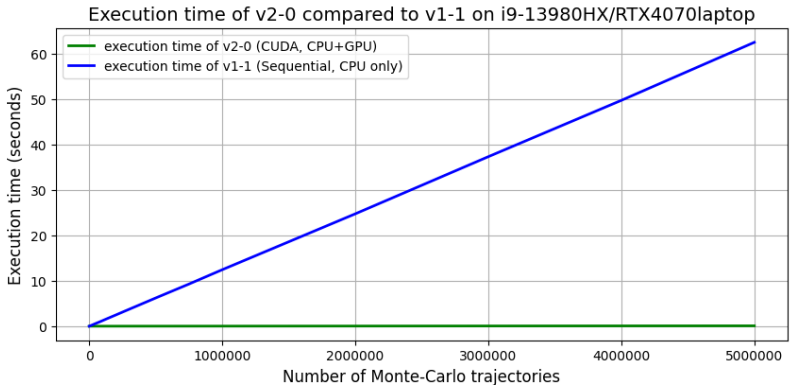
Performance comparison between `v1-1` and `v2-0`

Below is the execution time of our parallelised program :



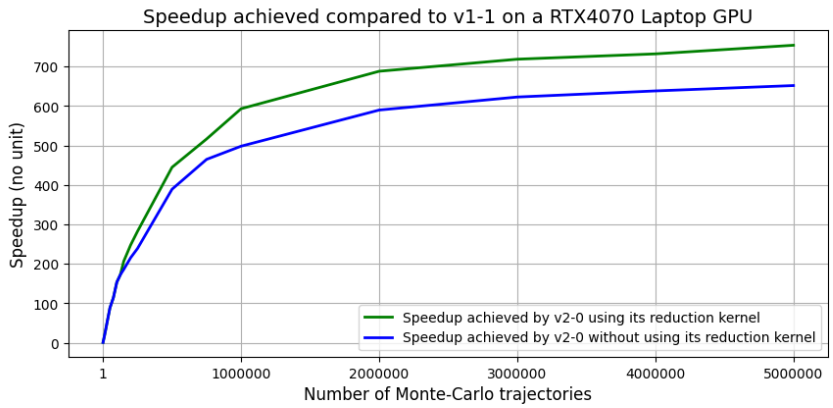
One can notice that `CUDA` causes only a $\sim$ 6ms initialization overhead, while enabling the `v2-0` to have a smooth quasi-linear execution time. This is a good indicator of successful parallelisation.

Let's now plot the execution times of the `v1-1` and `v2-0` on the same graph :



The `v2-0` is so much faster than `v1-1` that it looks like its execution-time curve lays on $y = 0$.

Rather than using logarithmic scales, let's simply plot the speedup directly :



This curve illustrates a smooth progression in speedup, ranging from 1 to 750, following a pseudo-logarithmic trend. As N increases, the speedup improves until reaching a point (beyond $N = 5\,000\,000$, not shown here) where bottlenecks inevitably emerge. It can be observed that the reduction kernel significantly improves speedup for N values exceeding $\sim 150\,000$.

Parallelised `CUDA` implementation of Monte-Carlo for many cases

While `v1-0`, `v1-1`, and `v2-0` assist in determining the value of a call option price for specific parameters, we now aim to develop an implementation that evaluates this price across a wide range of parameter combinations. Here are the parameters we intend to vary :

- Y_0 is a spot value
- α is the mean reversion rate of the volatility
- m is the long-term log-volatility
- ρ is the correlation coefficient between some Brownian motions
- K is the strike price
- T is the maturity (reached when time t equals T)
- ν^2 is the parameter such that the volatility of the log-volatility can be rewritten as follows :

  $\beta \;=\; \sqrt{2\,\alpha\,\nu^2}\;\;(1 - \exp(m))$

  It is preferable to use ν^2 over β because the latter is dependent of α, m and ν^2 itself, which might cause entanglement issues for the upcoming machine learning model.

Adapting the Euler discretisation scheme

Using varying parameters requires generalisation in the scheme. Let's rewrite it :

$$S_{t+\Delta t} \;=\; S_t \;+\; \exp(Y_t)\,S_t\,\sqrt{\Delta t}\,G_1$$

and

$$\begin{aligned}
Y_{t+\Delta t} \;&=\; Y_t \;+\; \alpha\,(m - Y_t)\,\Delta t \;+\; \beta\,\sqrt{\Delta t}\;(\rho\,G_1 \;+\; \sqrt{1-\rho^2}\,G_2) \\
&=\; Y_t \;+\; (\alpha\,m \;-\; \alpha\,Y_t)\,\Delta t \;+\; \beta\,\sqrt{\Delta t}\;\rho\,G_1 \;+\; \beta\,\sqrt{\Delta t}\,\sqrt{1-\rho^2}\,G_2 \\
&=\; Y_t \;+\; \alpha\,m\,\Delta t \;\;-\; \alpha\,Y_t\,\Delta t \;+\; \beta\,\sqrt{\Delta t}\;\rho\,G_1 \;+\; \beta\,\sqrt{\Delta t}\,\sqrt{1-\rho^2}\,G_2 \\
&=\; Y_t\;(1 \;-\; \Delta t\,\alpha) \;+\; \alpha\,m\,\Delta t \;\;+\; \beta\,\sqrt{\Delta t}\;\rho\,G_1 \;+\; \beta\,\sqrt{\Delta t}\,\sqrt{1-\rho^2}\,G_2 \\
&=\; Y_t\,C_1 \;+\; C_2 \;+\; C_3\,G_1 \;+\; C_4\,G_2
\end{aligned}$$

Where:

- $C_1 \;=\; 1 \;-\; \Delta t\,\alpha$
- $C_2 \;=\; \Delta t\,\alpha\,m$
- $C_3 \;=\; \beta\,\sqrt{\Delta t}\;\rho$
- $C_4 \;=\; \beta\,\sqrt{\Delta t}\,\sqrt{1-\rho^2}$

Note that we did not assign a constant variable to $\sqrt{\Delta t}$: for varying T values, directly passing Δt and $\sqrt{\Delta t}$ to the MC kernel is more convenient and avoids some repeated operations.

Implementation of v3-0 (MC.cu)

In the header, we declare global GPU memory symbols :

```
__device__ float Y0d[10];
__device__ float md[10];
__device__ float alphad[10];
__device__ float nu2d[10];
__device__ float rhod[10];
__device__ float Kd[16];
```

In the main(), we first define N and Δt, which are fixed :

```
// Defining number of MC trajectories (N) and time steps size (dt)
int N = 256*512;            // This is equal to 131072
float dt = 1.0f/1000.0f ;
float sqrtdt = sqrtf(dt);
```

Subsequently, we establish the parameter space corresponding to our exploration range :

```
// Exploring 10 decreasing "Y0" values from log(0.08) to log(0.4)
float Y0[10] = {logf(0.4f), logf(0.35f), logf(0.31f), logf(0.27f),
                logf(0.23f), logf(0.2f), logf(0.17f), logf(0.14f),
                logf(0.11f), logf(0.08f)};

// Exploring 10 decreasing "m" values from log(0.06) to log(0.34)
float m[10] = {logf(0.34f), logf(0.3f), logf(0.27f), logf(0.24f),
               logf(0.21f), logf(0.18f), logf(0.15f), logf(0.12f),
               logf(0.09f), logf(0.06f)};

// Exploring 10 increasing "alpha" values from 0.1 to 51.2
float alpha[10] = {0.1f, 0.2f, 0.4f, 0.8f, 1.6f, 3.2f, 6.4f, 12.8f,
                   25.6f, 51.2f};

// Exploring 10 increasing "nu2" values from 0.6 to 1.5
float nu2[10] = {0.6f, 0.7f, 0.8f, 0.9f, 1.0f, 1.1f, 1.2f, 1.3f,
                 1.4f, 1.5f};

// Exploring 10 decreasing "rho" values from -0.95 to 0.95
float rho[10] = {0.95f, 0.75f, 0.55f, 0.35f, 0.15f, -0.15f, -0.35f,
                 -0.55f, -0.75f, -0.95f};

// Exploring 16 increasing "T" values from 1/12 to 2
float T[16] = {1.0f/12.0f,  2.0f/12.0f, 3.0f/12.0f, 4.0f/12.0f,
               5.0f/12.0f, 6.0f/12.0f, 7.0f 12.0f, 8.0f/12.0f,
               9.0f/12.0f, 10.0f/12.0f, 11.0f/12.0f, 1.0f,
               1.25f, 1.5f, 1.75f, 2.0f};

// Declaring the 16 "K" values array.
// It will be defined dynamically (inside loops) depending on T.
float K[16];
```

Fixed parameters arrays are then copied into global GPU memory :

```
cudaMemcpyToSymbol(Y0d, Y0, 10*sizeof(float));
cudaMemcpyToSymbol(md, m, 10*sizeof(float));
cudaMemcpyToSymbol(alphad, alpha, 10*sizeof(float));
cudaMemcpyToSymbol(nu2d, nu2, 10*sizeof(float));
cudaMemcpyToSymbol(rhod, rho, 10*sizeof(float));
```

As described in the parameter space definition, the `K[]` array must be computed dynamically based on the values of T, stored in `T[]`. For a specific `T[i]` value, the corresponding `K[]` array can be generated as follows using the `strikeInterval()` function (whose body is not shown here for brevity) :

```
strikeInterval(K, T[i]); //Fills K[] array with 16 values based on T[i]
```

This means that K can take 16 values (stored in the `K[]` array) for each of the 16 possible values of T. While it might initally seem that K could form $16 \times 16 = 256$ combinations across all values of T, it is restricted to 16 values for any specific T. Consequently, the contribution of K to the increase of the parameter space size is only a factor of its own array size. Thus, we have :

$$\begin{aligned}\text{Parameter Space Size} &= \text{size of } Y_0[] \times \text{size of } m[] \times \text{size of } \alpha[] \times \text{size of } \nu^2[] \times \text{size of } \rho[] \times \text{size of } T[] \times \text{size of } K[] \\ &= 10 \times 10 \times 10 \times 10 \times 10 \times 16 \times 16 \\ &= 25\,600\,000\end{aligned}$$

Given the fact that $N = 256 * 512 = 131072$, applying the same parallelization logic as in `v2-0` would result in generating $25\,600\,000 \times 131072 = 3\,355\,443\,200\,000$ threads, leading to significant overhead.

We propose a new approach : the `T[]` array will be iterated over 16 times, reducing the parameter space to $25\,600\,000/16 = 1\,600\,000$ per iteration. For each iteration, a Monte-Carlo kernel will be launched with 1 600 000 threads, with each thread assigned a unique parameter combination to process all N trajectories. One can rewrite this number to find an appropriate `CUDA` launch configuration :

$$\begin{aligned}&1\,600\,000 \;/\; 512 \;=\; 3\,125 \\ \iff\; &\texttt{TOTAL\_THREADS} \;=\; 512 \;\times\; 3\,125 \\ \iff\; &\texttt{BLOCK\_SIZE} = 512 \quad \text{and} \quad \texttt{GRID\_SIZE} = 3\,125\end{aligned}$$

The 512 `BLOCK_SIZE` is large enough to reduce overhead, yet small enough to enable all `GPUs` to run it. We initialize this configuration in the `main()` :

```
int BLOCK_SIZE = 128 * 4    // This is equal to 512
int GRID_SIZE = 625 * 5     // This is equal to 3125
```

Then, we assign pseudo-random states to all threads :

```
// Declaring, allocating and generating random CUDA states
// For readability, the body of init_curand_state_k is not shown here
// Randomness is based on a time+idx based seed+offset
curandState* states;
cudaMalloc(&states, GRID_SIZE*BLOCK_SIZE*sizeof(curandState));
init_curand_state_k <<<GRID_SIZE, BLOCK_SIZE>>> (states);
```

We also have to declare a few more variables on the host :

```
// Declaring local host variables
float KR, mR, alphaR, nu2R, rhoR, Y0R, price, error;

// Declaring a pointer for the Monte-Carlo results array
float *mcResults;

// Declaring the "same" index. It will be used to locate the results
// of a specific parameters combination in the MC results array
int same;

// Declaring a file pointer and a string buffer to construct file names
// We will have to generate 256 files (1 for each T-K combination)
FILE* fpt;
char strg[30];
```

Here is the body of the adapted Monte-Carlo kernel :

```
__global__ void MC_k(float dt, float sqrtdt, int steps, float T,
    int N, curandState* state, float* mcResults){

    // Determining global thread ID and assigning a random state
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    curandState localState = state[idx];

    // Declaring local variables and MC accumulators
    float S, Y, price, G1, G2; float2 randPair;
    float sumPayoffs = 0.0f, sumSquaredPayoffs = 0.0f;

    // Initializing local varying parameters
    float KR = Kd[idx / 100000], Y0R = Y0d[(idx % 100000) / 10000],
    mR = md[(idx % 10000) / 1000], rhoR = rhod[(idx % 1000) / 100],
    alphaR = alphad[(idx % 100) / 10], nu2R = nu2d[idx % 10];

    // Precomputing constants
    float betaR = sqrtf(2.0f*alphaR*nu2R)*(1.0f-expf(mR)) ;
    float C1=1.0f-dt*alphaR, C2=dt*alphaR*mR, C3=betaR*sqrtdt*rhoR,
    C4=betaR*sqrtdt*sqrtf(1.0f-(rhoR*rhoR)) ;

    // Computing for all Monte-Carlo trajectories (N)
    for (int i = 0; i < N; i++) {

        // Initial conditions
        S = 1.0f; Y = Y0R;

        // Computing for every time step (dt)
        for (int j = 0; j < steps ; j++) {

            // Updating values
            randPair = curand_normal2(&localState);
            G1 = randPair.x; G2 = randPair.y;
            S = S * (1.0f + expf(Y) * sqrtdt * G1);
            Y = Y * (1.0f - C1) + C2 + C3 * G1 + C4 * G2 ;

        }

        // Avoiding extreme values of S that
        // would distort the upcoming NN
        if (S < 12.0f) {
            price = S-KR > 0.0f ? S-KR: 0.0f;
            sumPayoffs += price; sumSquaredPayoffs += price * price;
        }
    }

    // Copying results and square results to the mcResults[] array
    mcResults[2 * idx] = sumPayoffs / N;
    mcResults[2 * idx + 1] = sumSquaredPayoffs / N;

    // Copying random state back to global memory
    state[idx] = localState;
}
```

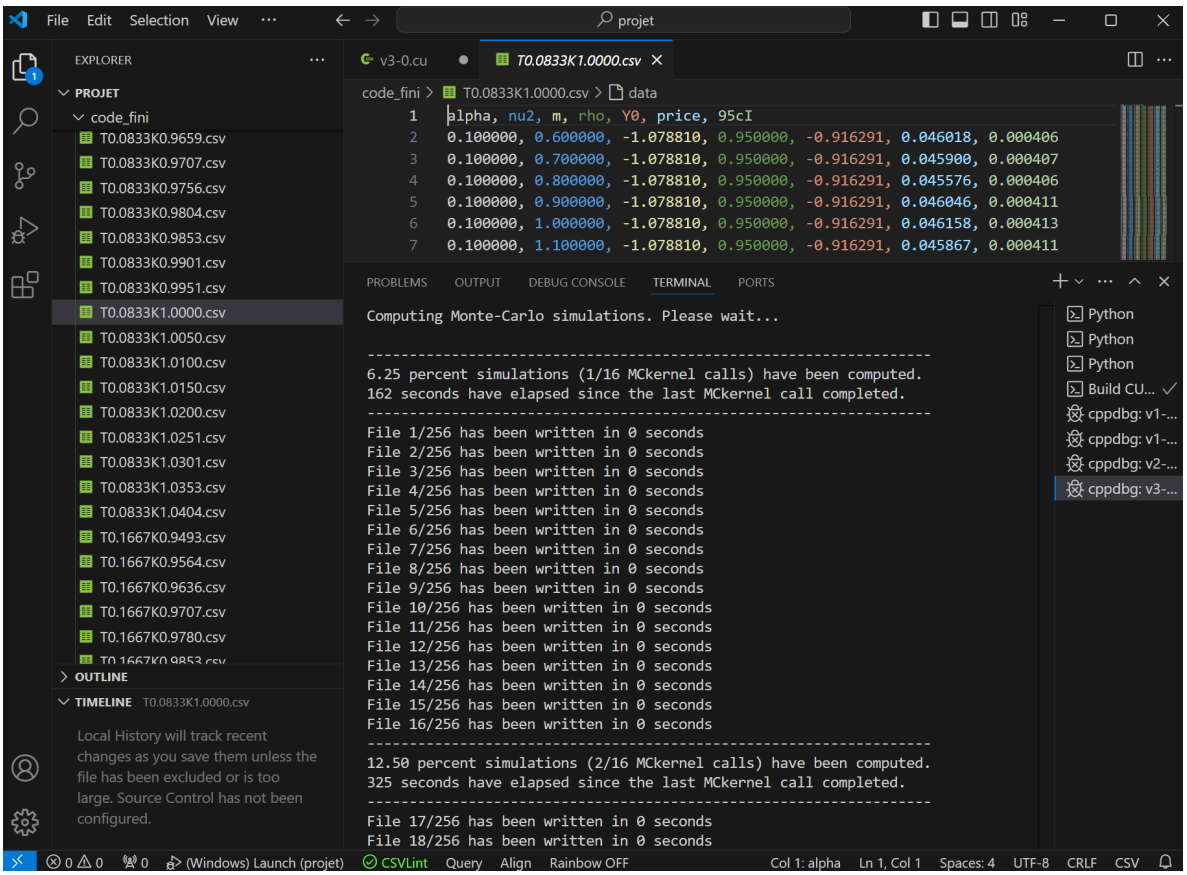
Due to the shift in approach mentioned on page 16, some aspects of this kernel warrant highlighting :

- Recall that this kernel is designed to be called 16 times, once for each value of T.
- The `KR` value of a thread is defined by `KR` = `idx`/100 000. Therefore, the first 100 000 threads will initialize their `KR` value to `Kd[0]`. The next 100 000 threads will initialize it to `Kd[1]`, and so on until the last 100 000 threads to `Kd[15]`.
- The other local parameters of the thread (`Y0R`, `mR`, `rhoR`, `alphaR` and `nu2R`) are initialized by retrieving values from device symbols using indexes determined by modular arithmetic. This ensures that all parameter combinations are explored exactly once.
- The `mcResults[]` array has to be of size $2 \times 1\,600\,000 \times$ `sizeof(float)` because the kernel produces 2 results for each of the 1 600 000 parameters combinations. The first of these two results, stored in `mcResults[2*idx]`, corresponds to the estimated payoff after N trajectories. The second, stored in `mcResults[2*idx+1]`, will be used later on to compute the error.
- Copying random states back to global `GPU` memory prevents threads from initializing their states the same way as their predecessors.

Here is how the MC kernel is called in the `main()` :

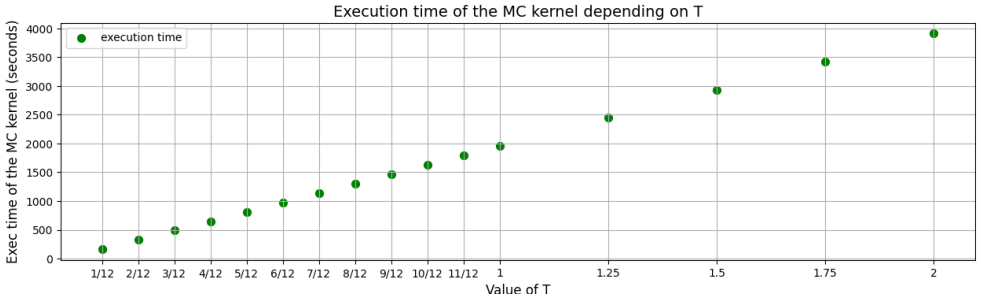
```
// Looping through all 16 possible values of maturity (T)
for(int i=0; i<16; i++){

    // Allocating memory (doing this in the loop saves 175MB on GPU)
    cudaMallocManaged(&mcResults,2*GRID_SIZE*BLOCK_SIZE*sizeof(float));

    // Computing the values of the K array depending on T[i]
    // Then copying it into the GPU device memory
    strikeInterval(K, T[i]);
    cudaMemcpyToSymbol(Kd, K, 16*sizeof(float));

    // Launching the MC kernel for on the T[i] value of T
    MC_k<<<GRID_SIZE,BLOCK_SIZE>>>(dt, sqrtdt, steps, T[i], N, states,
    mcResults); cudaDeviceSynchronize();

    // LOOPING THROUGH THE RESULTS ON THE HOST NOW
    // This loop writes the results in a file ; no heavy computations
    // Looping through all 16 possible values of strike (K)
    for(int j=0; j<16; j++){

        /* WRITING 16 T-K COMBINATIONS FILES :
        T[i]K[0].csv, T[i]K[1].csv, T[i]K[2].csv, ..., T[i]K[15].csv
        SKIPPING FILE WRITING LOGIC FOR SIMPLICITY*/

    }

    // Freeing mcResults[] because the results are saved in the files
    cudaFree(mcResults);

}

// Freeing random thread states, which were still in GPU memory
cudaFree(states);

// Terminating the program
return 0;
```

After executing the code, the results are saved properly in 256 different `.csv` files. Each of these correspond to a fixed T and K combination, and hold the 100 000 unique combinations of α, ν^2, m, ρ and Y_0. Below is a screenshot of an IDE showing that :



When looking at the execution messages printed in the terminal, it is noticeable that :

- File writing on the host always completes in less than second, which is considerably faster than the 162+ seconds of a Monte-Carlo kernel execution. As a result, there is no compelling need to optimize it using reduction kernels or string buffers.
- The execution time of the Monte Carlo kernel varies with T as the number of time steps increases proportionally. For $T = 1/12$, approximately 80 steps are required, while for $T = 2/12$, the steps double. The graph below shows the kernel execution time as a function of T :



As the program invokes the MC kernel once for each of the 16 values of T, the total execution time can be estimated by summing these 16 measurements. This results in a global execution time of approximately 25 000 seconds, equivalent to nearly 7 hours.

Optimisation : implementation of `v3-1`

Profiling the code reveals that exponentiation and random number generation constitute a significant portion of the program's execution time. In this section, we will implement a code optimization prioritising speed over the quality of randomness.

Recall that for a given T, the kernel call involves threads processing $N = 131\,072$ trajectories. Each trajectory consists of $\lfloor T/\Delta t \rfloor$ steps (as the last step to be evaluated must occur before or at maturity, not after). At each of these, 2 standard normal random variables (G_1 and G_2) are generated.

Let's examine this RNG's order of magnitude :

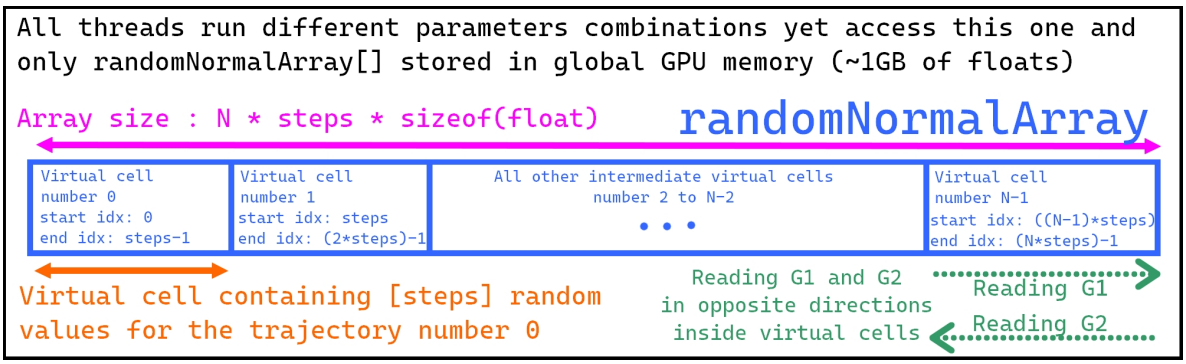
MCkernel call	maturity T	steps per trajectory	random $\{G_1, G_2\}$ pairs generated
call 1/16	1/12	83	10 878 976
call 2/16	2/12	166	21 757 952
call 3/16	3/12	250	32 768 000
call 4/16	4/12	333	43 646 976
call 5/16	5/12	416	54 525 952
call 6/16	6/12	500	65 536 000
call 7/16	7/12	583	76 414 976
call 8/16	8/12	666	87 293 952
call 9/16	9/12	750	98 304 000
call 10/16	10/12	833	109 182 976
call 11/16	11/12	916	120 061 952
call 12/16	1	1000	131 072 000
call 13/16	1, 25	1250	163 840 000
call 14/16	1, 5	1500	196 608 000
call 15/16	1, 75	1750	229 376 000
call 16/16	2	2000	262 699 712
TOTAL (all calls combined)	$\emptyset$	$\emptyset$	1 703 411 712

Nearly two billion $\{G_1, G_2\}$ pairs are generated, which is costly and exceeds what is strictly necessary. Notably, while Monte-Carlo trajectories within the same simulation require distinct random variables (to ensure convergence to theoretical results through the law of large numbers), the simulations themselves differ in parameters and do not inherently require different RNGs.

Therefore, providing every thread with the same large array of random variables would significantly reduce the number of `curand_normal2()` calls while preserving the quality of randomness.

Generating such an array for both the G_1 and G_2 variables is unnecessary : Creating a single array and traversing it in ascending order for G_1 and descending order for G_2 ensures that the combinations are almost never repeated, thereby preserving much of the randomness quality.

The following is a sketch illustrating the concept of this array approach:



Here is a kernel implementation to generate the desired array adapted to our case in no time :

```
__global__ void init_randarray_k(float* randomNormalArray,
    curandState* states) {

    // Initialisating variables
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    curandState localState = states[idx];
    float2 randomPair ;

    // This kernel is designed to be launched with 256*125 threads
    // The generated array has to contain 262144000 values
    // So each thread has to process 262144000/(256*125) = 8192 values
    // Hence the offset below
    int offset = 8192 * idx ;

    // Generating 8192 random values
    for (int i = 0; i < 8192; i+=2) {
        randomPair = curand_normal2(&localState);
        randomNormalArray[offset+i] = randomPair.x;
        randomNormalArray[offset+i+1] = randomPair.y;
    }

    // Copying local random state to global gpu memory
    states[idx] = localState;
}
```

And here is how it is called from the `main()` :

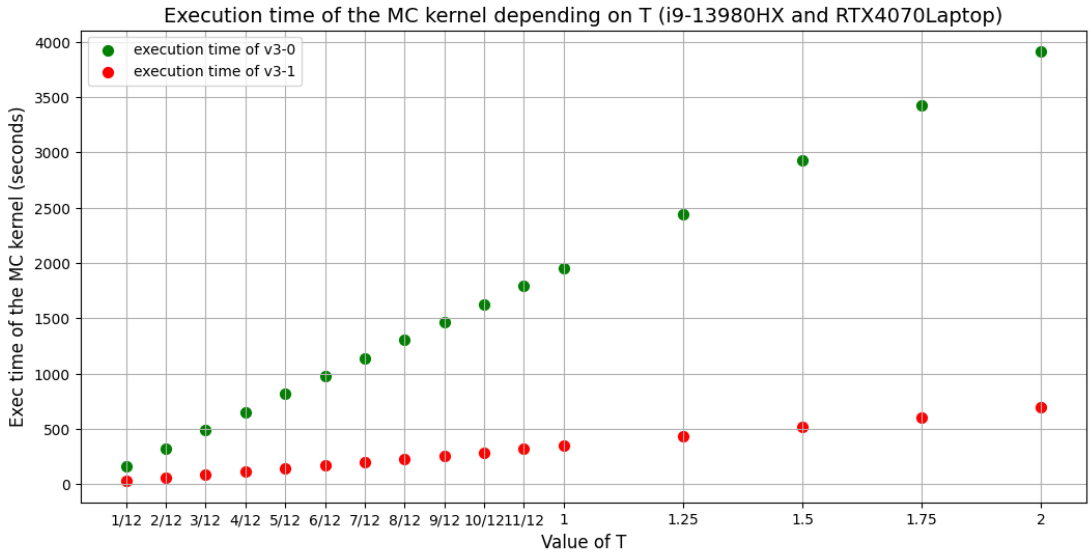
```
// Initialisating a pointer
curandState* states;
cudaMalloc(&states, GRID_SIZE*BLOCK_SIZE*sizeof(curandState));
init_curand_state_k <<<GRID_SIZE, BLOCK_SIZE>>> (states);

// Declaring a pointer for the random array
float* randomNormalArray ;

// Allocating required space in global GPU memory
cudaMalloc(&randomNormalArray, 125*256*8192*sizeof(float));

// Launching the kernell to fill the array as intended
init_randarray_k<<<125, 256>>>(randomNormalArray, states);

// Synchronisating to avoid any premature access to the array
cudaDeviceSynchronize();

// RNG complete : random states can be freed
// The MC kernel does not need them anymore as the RNG is already done
cudaFree(states);
```

The array is initialised only once with the maximum possible steps amount, which means only the last kernel explores it fully. When caring about live-time memory management rather than light overheads, this part shall be moved and adapted inside the `i` loop (iterating over `T[]` values).

The Monte-Carlo now has to be adapted as follows to fit the new approach :

```
__global__ void MC_k(float dt, float sqrtdt, int steps, float T, int N,
    float* randomNormalArray, float* results){

    // Determining global thread ID and
    int idx = blockDim.x * blockIdx.x + threadIdx.x ;

    // Declaring offsets, local variables & initializing accumulators
    int offsetG1, offsetG2; float S, Y, price, G1, G2;
    float sumPayoffs = 0.0f, sumSquaredPayoffs = 0.0f;

    // Initializing local varying parameters
    float KR = Kd[idx / 100000];
    float Y0R = Y0d[(idx % 100000) / 10000];
    float mR = md[(idx % 10000) / 1000];
    float rhoR = rhod[(idx % 1000) / 100];
    float alphaR = alphad[(idx % 100) / 10];
    float nu2R = nu2d[idx % 10];

    // Precomputing constants
    float betaR = __fsqrt_rn(2.0f*alphaR*nu2R) * (1.0f - __expf(mR));
    float C1 = (1.0f - dt * alphaR);
    float C2 = dt * alphaR * mR;
    float C3 = betaR * sqrtdt * rhoR;
    float C4 = betaR * sqrtdt * __fsqrt_rn(1.0f - (rhoR * rhoR));

    // Computing for all Monte-Carlo trajectories (N)
    for (int i = 0; i < N; i++) {

        // Initial conditions
        S = 1.0f; Y = Y0R;
        // Updating offsets
        offsetG1 = 2000 * i ; offsetG2 = offsetG1 + 1999 ;

        // Computing for every time step (dt)
        for (int j = 0; j < steps; j++) {

            G1 = randomNormalArray[offsetG1 + j];
            G2 = randomNormalArray[offsetG2 - j];
            S *= (1.0f + __expf(Y) * sqrtdt * G1);
            Y = Y * C1 + C2 + C3 * G1 + C4 * G2;

        }

        // Avoiding extreme values of S distorting the upcoming NN
        if (S < 12.0f) {
            price = S - KR > 0.0f ? S - KR : 0.0f;
            sumPayoffs += price;
            sumSquaredPayoffs += price * price;
        }
    }

    results[2 * idx] = sumPayoffs / N;
    results[2 * idx + 1] = sumSquaredPayoffs / N;
}
```

Notice that the `expf()` (resp. `sqrtf()`) calls were replaced by `__expf()` (resp. `__fsqrt_rn()`) which feature equally-satisfying results and slightly better performance, especially when the `nvcc` compiler is called with the `-use_fast_math` argument.



Across all executions of the Monte-Carlo kernel, `v3-1` achieves a quasi-linear speedup of ~ 5.6 compared to `v3-0`. This is extremely significative ; the global execution time drops from nearly 7 hours to only 1 hour 15 minutes, which is reasonable for a powerful laptop and manageable for a one-time run.

However, beyond the scope of this paper, many more optimisations could be implemented to further reduce the execution time of this program. Here are some considerations about it :

- The current random array size is really convenient for index management, but it could be shrunk without significatively decreasing randomness quality. 50 000 random values would most likely be enough for all these simulations as long as the index/offset logic is adapted to follow a non-repetitive pattern, which is harsh to set up. It would free X`mb` of global `GPU` memory at runtime.
- Furthermore, some profiling reveals that memory access to the array from inside the Monte-Carlo kernel represents roughly 15% of its execution time. This means reducing the random array in size and chunk-loading it into fast `GPU` block-memory could lead to improvement.
- Profiling also reveals that `__expf()` - although performing better than `expf()` - is responsible for almost 60% of the MC kernel execution time. One could think about approximations or table-based approaches to reduce its computational cost.
- On a more general note, a threshold could be established to automatically terminate Monte Carlo simulations when their current trajectorie's incremental contribution to the global average estimate falls below a predefined level of significance.

Perspectives

HPC Optimisation

The work detailed in this document focuses on basic optimisations. Much more can be done, starting with a detailed profiling through adequate tools such as `Nsight Systems` and `Nsight Compute`.

Machine Learning

The `.csv` files generated by this code can be used as a learning set for a machine learning model. Some undisclosed ML tests showed easily-achievable and accurate predictions for these call-option prices.