Youssef Azzaoui

Project Analysis Report

CUDA Parallel Computing for American Call Options Pricing

I. Problem Analysis

An American call option can be characterized by:

- Its underlying asset $S_t$which value fluctuate with time t.
- The maturity date T.
- The strike K of the option.

The holder of the call option can exercise it whenever he wants between the maturity dates T and T_0 (set to zero). He will then receive the payout $\varphi(S_t) = \max(S_t - K, 0)$

The main assumption is that we can model the underlying asset price by using this Stochastic Differential Equation (SDE):

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

Where μ Is the standard deviation, σ the volatility and W the Brownian motion?

Moreover, under Itō's interpretation, the SDE has a closed form solution, named the Black-Schole formula:

$$S_t = S_0 \exp\left(\left(r - \frac{\sigma^2}{2}\right)t + \sigma W_t\right)$$

With the arbitrage-free market hypothesis it can be shown that the option priceP_t, called the premium, is the solution of the following stochastic optimization problem:

$$P_t = \sup_{t<\tau<T_0} E[\exp(-r(\tau - t)) * \varphi(S_t)]$$

The problem can now be formally described as finding the premium price P_0 with the following input data:

- The underlying asset initial price S_0
- The strike price K
- The maturity date T
- The annual interest rate r
- The volatility σ (which is the standard deviation of the underlying price)
- The step time size δt

II. Serial Algorithm Solution

There are many approach that were proposed by quantitative finance expert and a promising way to solve this problem is the Longstaff-Schwartz algorithm which could be simply described like this:

1. Generate M random trajectory for the underlying asset using the Black-Schole formula.
2. On each trajectory compute the optimal call option price.
3. By using a Monte-Carlo method compute the premium.

In the following, lets *nStep* be equal to $T/\delta t$, S_j^i the computed underlying asset price at step j for the trajectory i, P_j^i the computed premium at step j for the trajectory i, S_{t_j} and P_{t_j} two random variables.

1) Underlying asset price generation

The Black-Schole formula can be adapted to generate iteratively the underlying price for $i \in (1, M)$:

$$S_{t+\delta t}^i = S_t^i \exp\left(\left(r - \frac{\sigma^2}{2}\right)\delta t + \sigma(W_{t+\delta t} - W_t)\right)$$

and $(W_{t+\delta t} - W_t) \sim N(0, \delta t)$

Time Complexity: **O(nStep * M)**

2) Trajectory Call option pricing

a) General algorithm

The Long staff-Schwartz algorithm is the following in pseudo-code:

$\forall i \in (1, M) P(i, nStep) = \varphi(S_{nStep}^i)$
$for\ j = nStep\ to\ 1\ do$
  $for\ i = 1\ to\ M\ do$
    $P^*(i,j) = E\left(P_{t_j} \middle| S_{t_{j-1}} = S_{j-1}^i\right)$
    $if\ exp(-r * \delta t) P^*(i,j) < \varphi(S_{j-1}^i)\ then$
      $P(i, j-1) = \varphi(S_{j-1}^i)$
    $else$
      $P(i, j-1) = exp(-r * \delta t) P^*(i,j)$
  $end\ for$
$end\ for$

Here you can see that the algorithm assume that we are able to compute the expectation $E\left(P_{t_j}\middle|S_{t_{j-1}} = S_{j-1}^i\right)$ which is a non-trivial computation (Monte-Carlo in Monte-Carlo).

Let's assume that C_e is the time complexity to compute this expected value.
The overall algorithm complexity is equal to **O(nStep * M * C_e)**

b) Expected value computation

Tsilikis Van Roy has shown that $f: x \to E\left(P_{t_j}\middle|S_{t_{j-1}} = x\right)$ is in fact:

$$f = arg\min_{h\in L^2} E[(P_{t_j} - h(S_{t_{j-1}}))^2]$$

The most simple way to approach f is to project h in the polynomial space with the canonical basis: $(1, x, x^2, \ldots, x^n)$. Note that actually Laguerre Polynomial are being used in some implementation.
So formally, with $f^*(x) = \sum_{k=0}^{n} \alpha_k x^k$, one have:

$$\alpha^* = arg\min_{(\alpha_0,..,\alpha_n)\in R^{n+1}} E[(P_{t_j} - \sum_{k=0}^{n} \alpha_k (S_{t_{j-1}})^k)^2]$$

This is now a problem of linear regression that we can solve by many approach: SVD, QR or Cholesky algorithm. We will implement the former for its relative simplicity.

By a using a simple Monte-Carlo approach to compute the expectation (no more conditional) and by using matrix notations, the problem becomes:

Let $j\epsilon(1, nStep)$,

Find $a^* = arg\min_{(\alpha_0,..,\alpha_n)\in R^{n+1}} \|Aa - B\|^2$
where $A_{i,k} = (S_{j-1}^i)^k$ *and* $B_i = P_j^i$ *with* $k \in (0, n)$ *and* $i \in (1, M)$

Hence, the derived normal equation is:

$$A^*A\, a^* = A^*B$$

And finally an estimate of the conditional expectation is given by $\sum_{k=0}^{n} \alpha_k (S_{j-1}^i)^k$

Thus, the process is:

1. Compute A , Transpose A : O((n+1) M)
2. Compute $R = A^*A$: O((n+1)^2 M)
3. Compute Y = A*B : O ((n+1)^2)
4. Compute the Cholesky L matrix of R=LDL* : O($(n+1)^3/3$)
5. Find z with Lz=Y by using the standard forward substitution : O((n+1)^2)
6. Find a^* with DL*a^* = z : O($(n+1)^3 + M(n+1)$)
7. Compute the conditional expectation : O(n+1)

Expected value computation complexity $C_e \approx O(n^3 + Mn^2)$

c) Monte-Carlo Premium computation

The premium is then simply computed: $P_0 = \frac{1}{M} \sum_{i=1}^{M} P(i, 0)$

$$\textbf{Total algorithm complexity} \approx \boldsymbol{O(nStep\ (Mn^3\ +\ M^2n^2))}$$

Further Optimizations ideas:

- Given the fact that many matrix are symmetric or triangular, we can optimize the matrix multiplication algorithm.
- Since for a fixed j a^* is the same for every trajectory i, we can compute it only one time during the second loop of the LS algorithm.

$$\textbf{Optimized algorithm complexity} \approx \boldsymbol{O(nStep\ (n^3\ +\ Mn^2))}$$

d) Notes about the serial algorithm

The first version of the serial algorithm that I developed used the simple Cholesky factorization algorithm (M = L*L, with L triangular) and I was also trying to compute directly the inverses of the triangular matrix. It was leading to incorrect results because of the rounding errors and the square root operator applied to very small numbers, and thus, giving completely false results.

With the current algorithm the output gives credible results and after numerous simulations it converges to a single value. But without a way to compare the results I can't guarantee that the serial algorithm that I developed is producing the rights values. As a side note the Black-Scholes model used here is of course far from being perfect, but since everyone on the market is using it, everyone agree on its derived results. Thus making floating point precision important in this case.

III) Selected Parallelization strategy

The conditional expectation computation part is the most time-consuming part of the algorithm and especially the part 1 and the part 2 , given the fact that the polynomial order *degree* is in fact quite low (Proven after time profiling on live simulation) . Luckily these parts are apt to be "simply" parallelized at first sight. So we focused our work on parallelizing the computation of A and the matrix multiplication A*A.

In order to do so, we made each thread responsible for a small number of random trajectories. And after research, we discovered that a library named cuRAND can be used to generate random variables in bulk on the GPU so we also generated on the GPU the underlying price matrix but we won't study this part because the underlying generation can be neglected in front of the rest.

The selected parallelization strategy is the following:

1. Make each thread compute a fraction of the numbers of random trajectories, store data in the GPU global memory. Send it to CPU memory.

2. $for\ j\ =\ nStep\ to\ 1\ do$
   2.1 : Compute A on the GPU. Send the result to the CPU.
   2.2 : Perform A*A on the GPU. Send the result to the CPU.
   2.3 : Perform the rests of the operations on the CPU.

The kernels for part 1 and part 2.1 are rather straightforward to write but the matrix multiplications A*A is quite “tricky”. Indeed the dimension n of the result matrix is very low compared to nRandomWalks, so a thread cannot be responsible alone for the computation of a cell. Our idea was to make each thread computing a small portion of the result of the multiplication for every cell:

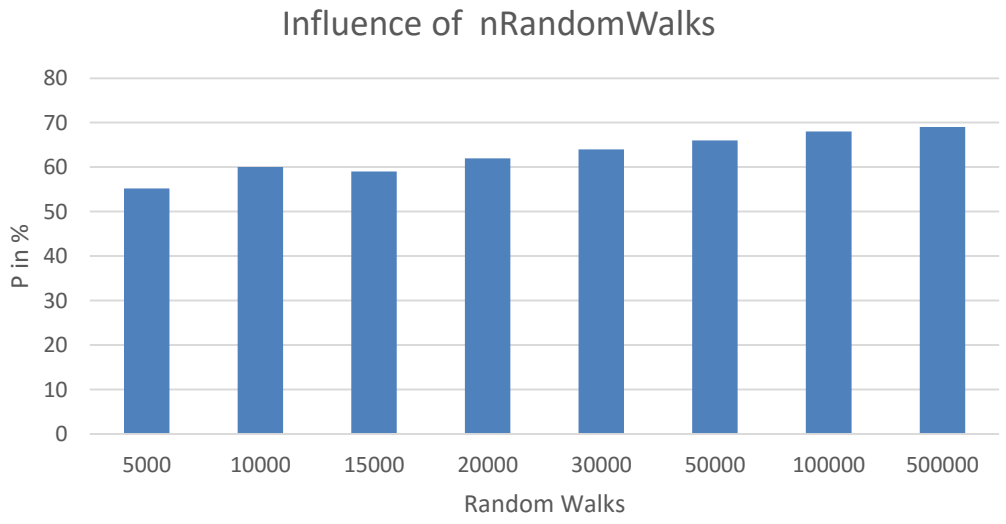
```
__global__ void computeAtAKernel
(double* A, double * Ata, int degree, int nRandomWalks, int simPerThread)
{

	int id = threadIdx.x;
	int firstRandomWalk = id * simPerThread;

		for (int i = 0; i< degree; i++)
		{
			for (int j = 0; j< degree; j++)
			{
				double sum = 0;
				int I = (i +id) % degree;
				int J = (j + id/degree) % degree;
				for (int k = firstRandomWalk; k < firstRandomWalk + simPerThread; k++)
				{
					sum += A[k*degree + I] * A[k *degree + J];
				}
				atomicAdd2(&Ata[I*degree + J], sum); //avoid races conditions
			}

	}
}
```

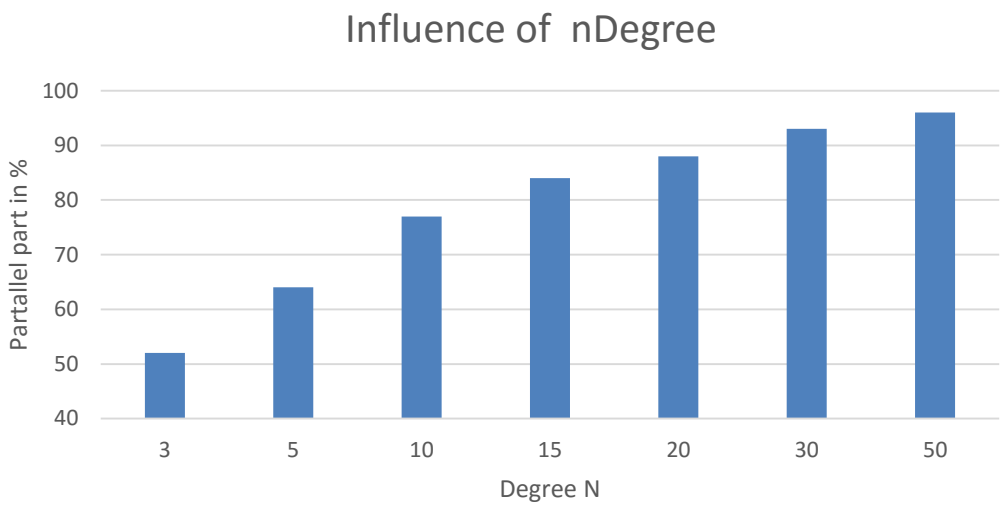
As you can see we are here using here a custom made atomicAdd function which allows to the threads to access concurrently to the same memory address of a Double value because the original one was limited to float and int.
I also introduced a small offset for each thread, in order to reduce the threads “congestion” by a factor of degree^2.

IV) Theoretical Analysis

a) Parameters influence analysis

In order to compute effectively the parallelizable part “p factor” we have to measure the parameters influence on it by measuring the time of the parallelizable part over the global time.

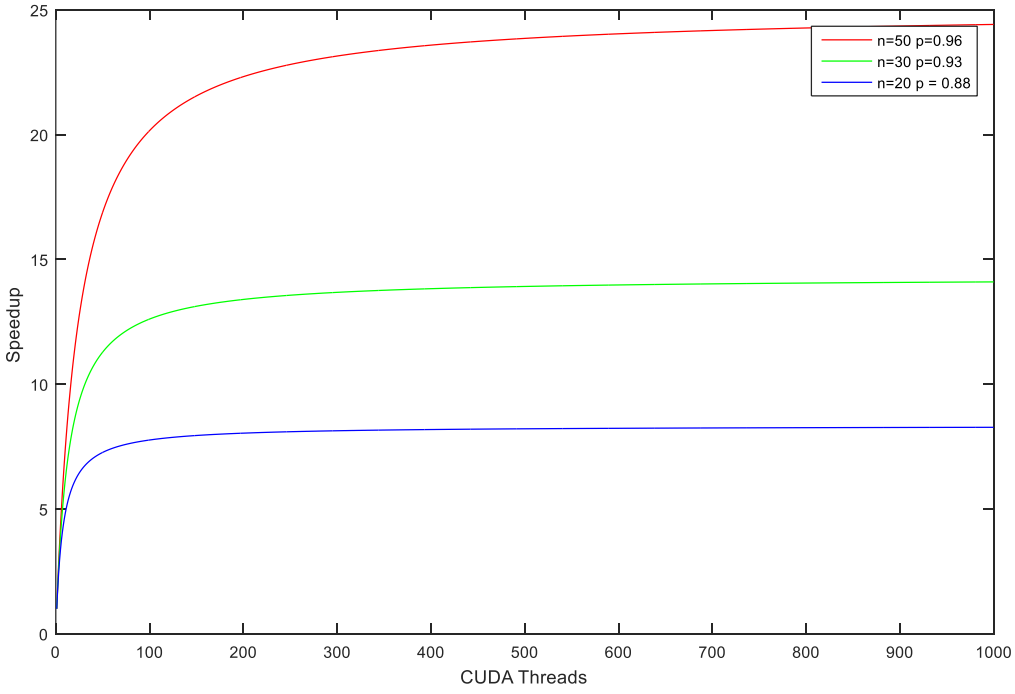




The results of those measurement tells that the more we are requesting precision, the more the parallelizable part becomes important. For a high number of simulations and a high number of degree we can almost hit 98% which is excellent and justify our approach. However it is impossible to handle such values for two raisons: the first one is the memory limitation on the GPU and the second one is that the double precision is not sufficient to compute which such scaling effect on the numbers of the matrix A.

b) Speedup Analysis

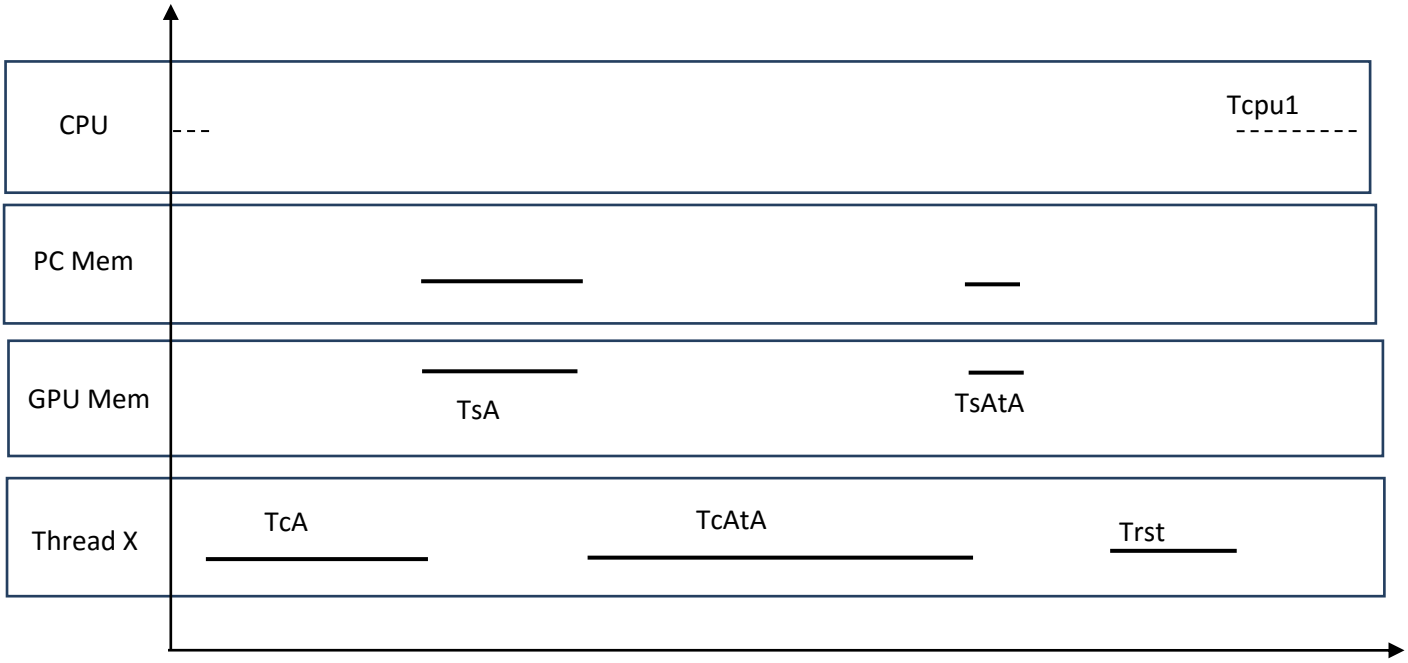
Without taking into account data transfers time and some GPU routines, here is a first version of the Amdahl Law with *nRandomwalk* (M) = 30000 .



Amdahl Law without taking data transfer time into consideration

c) Timing diagram

Since memory allocation during the initialization phase is executed only at the first iteration (360 in total) we don't take it into consideration. Here is the timing diagram of the two first step of the conditional expectation computation inside one iteration.



With:
- T_{cA} the time to compute the powers of the matrix A
- T_{sA} the time to send it back to the CPU memory
- T_{cAtA} the time to compute A*A and T_{sAtA} the time to send the result back to the CPU.
- T_{rst} the time to erase the GPU memory.
- T_{Cpu0} The time taken by the CPU outside the loop.
- T_{Cpu1} The time taken by the CPU inside an iteration

The parallel time is then:

$$T_{par} = T_{Cpu0} + nStep(\frac{T_{cA} + T_{cAtA}}{nThreads} + T_{sA} + T_{sAtA} + T_{rst} + T_{Cpu1})$$

At a first approximation, without taking into account the *AtomicAdd2* introduced delay we have:

$$T_{ser} = T_{Cpu0} + nStep(T_{cA} + T_{cAtA} + T_{Cpu1})$$

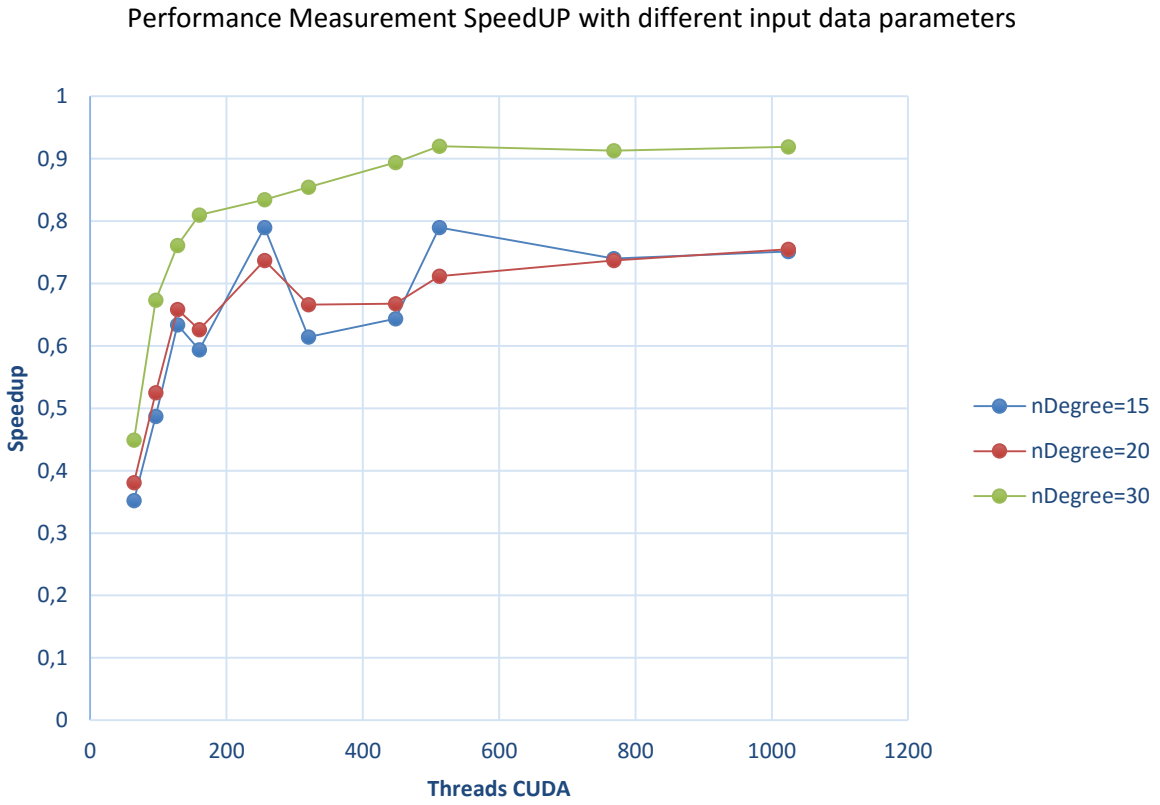
The theoretical speedup formula is then:

$$S = \frac{T_{Cpu0} + nStep(T_{cA} + T_{cAtA} + T_{Cpu1})}{T_{Cpu0} + nStep(\frac{T_{cA} + T_{cAtA}}{nThreads} + T_{sA} + T_{sAtA} + T_{rst} + T_{Cpu1})}$$

V) Performances evaluation

Due to a driver problem we were unable to start the simulations with 1 and 32 threads. Indeed a CUDA thread cannot be stuck in a kernel for a too long time.
The number of threads should be a multiple of a warp size 32 and must divide nRandomWalks (M).

For the implementation we will chose *nStep* = 365 and *nRandomWalks* = 32768.



The measured speedup, while following the trends expected, is absolutely not good. We can not even superpose the theoretical one on it.
In fact the communication time is too long in front of the computation time.
To verify it we launched the simulation with a large number of randomWalk and we obtained a speedup of 3 with 130000 simulation on 1024 CUDA cores. In fact the more the computation load is elevated the more the data transfers time can be neglected, but the simulations were too long to measure it quantitatively.
We unfortunately neglected this aspect during the design of our algorithm, thinking that the PCI express bus was enough fast to handle it.

V) Conclusion

The data communication time was the major issue of our strategy. One possible way to improve it would have been to send the vector B, perform the multiplication AB on the GPU then return the result: it would have reduced by a factor of *nDegree/2* the amount of exchanged data.
Yet for a very high amount of random walks generated, the speedup obtained justify the usage of CUDA.

To conclude, I found very interesting to work on this sort of real life applications of high performance computing. It helped me to improve my parallelization skill and to discover a little bit the mathematical finance area where I'm completely stranger.

For any comments/suggestions:
Youssef.Azzaoui@epfl.ch