Implementation of Mont Carlo algorithm related to finance on GPU

Alireza Keshavarzian, Amirahmad Ziaepour, S.Ahmad Motamedi
Dept. of Electrical eng
Amirkabir University of Technology, Tehran
Alireza.Keshavrzian@aut.ac.ir

***Abstract*— Mont Carlo algorithm has verity of application especially in finance which could predict the value of stocks. This algorithm due to its huge amount of computation, demands on devices could compute a large amount of operation simultaneously to meet the time and energy-efficiency constraints. In this report we assert a GPU-based implementation of Mont Carlo algorithm related to financial issues. GPUs could pave the way for us to leverage most sources of Parallelism in Mont Carlo.**

We examine two approaches— European method regardless to the steps among the paths and another method is applied for estimating the prices in each steps of path. For former method we achieve relatively 10x and for the latter we get 9.2x speedup over its C counterpart.

***Keywords*—Parallel, GPU, CUDA, Mont Carlo, finance, Black Scholes**

I. INTRODUCTION

FINANCIAL issues play a critical role in humans lives in such a way that most of companies invest a great portion of their money to predict stocks. These companies have applied some methods to make their estimation more accurate and reliable. Be that as it may, this method mostly relies on the repetitive and random numbers. Hence, computer with the ability the ability of computing large data or parallel instructions pave the way for this procedure.

Price of stocks and merchandises relies on some conditions consisting of time and risk which is indicated in this report as *T* and *R* respectively. It is obvious that the price of merchandises changes with the passage of time but the quality of this change is still a moot point. If we assume that the price of our stock at the beginning of time is P dollars, its value change by $P \times e^{-RT}$ formula. In other words, if our price is P after T years it will become$P \times e^{-RT}$.

If we indicate our investment in specific time with S_t, after a bit contention, we perceive that the next steps are following the Brownian motion. To delineate, if we indicate movement and velocity of this motion with μ and v respectively, the following formula originated:

$$dS_t = \mu S_t dT + v S_t dW_t$$

In this formula W_t is Wiener process which is often called Standard Brownian process. Delving further to the issue reveals that the Wiener process follow the $X = W_t - W_0 \sim N(0,T)$ which $N(\mu, \sigma^2)$ is mean distribution.

After solving this problem we reach the below formula:

$$dS_t = \mu S_t dt + v S_t dW_t \leftrightarrow \frac{dS_t}{S_t} = \mu dt + v dW_t \leftrightarrow S_t = S_0 e^{uT + v(W_T - W_0)}$$

By putting in use of Wiener procedure we reach:

$$S_t = S_0 e^{\mu T + v N(0,T)} = S_0 e^{\mu T + v\sqrt{T} N(0,1)}$$

The expected value according to the above equation will be:

$$E(S_T) = S_0 e^{\mu T} \times E\left(e^{N\left(0, v^2 T\right)}\right) = S_0 e^{\mu T} \times e^{0.5 v^2 T} = S_0 e^{(\mu + 0.5 v^2) T}$$

In this experiment, firs off, we produce sufficient random numbers which is declared in specific range to be applied in our forecasting formula. A simple example can be given to shed light on about what was elaborated above is that we generate one million random numbers to give hand to each paths in our prediction and then calculate the average of these numbers. Another solution is estimate the price of our stocks in each step in such a way that a great amount of memory is allocated for this process. In the following, we will aptly elaborate these methods.

II. FIRST METHOD

In first method, at first, we must generate our normal numbers to use them in Black Scholes algorithm. To elucidate, each path in Mont Carlo use one random number with mean of 0 and σ^2 as standard deviation. Due to this fact we generate 2048 random number in global memory of Graphic card. In order to achieve this goal, we invoke a function in device which produce random numbers. Significant point in this function is that each path or in another word each thread uses specific and unique seed to produce a random number. To deal with this aim, we adopt the clock of the system and the index of thread. It is obvious that combination of system clock and thread index makes a unique number for seed. Adopting this method assure us that random numbers produced from different seeds. Functions used in random numbers generators function invoked from its CUDA library called cuRand.

On GPU: Modern GPUs are throughput-oriented many core processors that rely on large-scale multithreading to attain high computational throughput and hide memory access time. The latest generation of NVIDIA GPUs has up to 15 multiprocessors, each with hundreds of arithmetic logic units (ALUs). GPU programs are called kernels, which run a large number of threads in parallel. For program which its operations are independent with each other, using GPU is rational choice. GPU due to its innumerable core in comparison to host, can pave the way for our algorithm to be executed in higher speed.

Function 1: generating random numbers

```
__global__ void cudaRand (double *d_out)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    curandState state;
    curand_init((unsigned long long)clock() + i, 0, 0, &state);
    d_out[i] = curand_uniform_double(&state);
}
```

After generating random numbers we must apply these numbers in another function which estimate the final value. According to pure independency of paths, using GPU could obviously increase our speed and performance. In order to average over these predicted stocks, we apply a reduction method in such a way that we break our one dimensional vector to pieces and then add these pieces with each other. The predicted value of stocks obtained by:

$$S \times e^{\mu T + v\sqrt{T} \times Rand(i)}$$

Using *shared memory*, we are able to enhance our performance. Shared memory has higher speed than its counterparts. Hence, we allocate the results of our paths for shared memory and after the finishing of our process, we add all of the numbers with each other. Adding a bunch of numbers in a vector simultaneously cause some errors which constrain us to go further. In order to solve this problem we use reduction algorithm in which two numbers in vectors are added and put the resultant value in one of memories. This formative summation continues until all of the data within a vector end up.

Function 2: estimate the price of stocks

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
__shared__ double cache[threadsPerBlock];
double temp = 0;
int cacheIndex = threadIdx.x;
while (tid < N)
{
   temp += s*exp(mu*T + V*sqrt(T)*randnum[tid]);
   tid += blockDim.x * gridDim.x;

}
cache[cacheIndex] = temp;
__syncthreads();
int i = blockDim.x / 2;
while (i != 0)
{
   if (cacheIndex < i)
      cache[cacheIndex] += cache[cacheIndex + i];
   __syncthreads();
   i /= 2;
}
if (cacheIndex == 0)
   output[blockIdx.x] = cache[0];
```

III. SECOND METHOD

Although the performance of our results is the culmination of sundry factors each one of which plays a critical role. The decision about whether GPU could fortify our results or not is still a controversial issue. To assure about the legitimacy of this contention we program our second method which has large portion of dependency in GPU. The result of our program shows that even with this amount of sequential portion, GPUs take precedence over the host. The GPU which we used in our experiment was GeForce 650 GTX which based on Kepler architecture.

In this method we should predict the all steps of our paths to obtain precise result. According to expedition demonstrated below, we proceed step by step to achieve our desirable results.

$$S_{t+dt} = S_t \times e^{(r-0.5\sigma^2)dt} + \sigma\sqrt{dt}Z$$

"Z" In this formula indicate a random number.

The procedure of this method is far simpler than the former but the amount of memory and dependency of its process is more than the first one. We should generate a large numbers of random numbers to support each step in the paths. Black Scholes function in second method does not need any aggregations of vectors. Significant part of this method is steps produced from its last sequences. To calculate the value of each step, we should process its previous values. Consequently, dependency among steps is a vivid constraint for us. Hence, we allocate a specific thread for each path and for each step through paths, we apply the unique random number. Another way to make our program more independent is that by having the first and the last value of the path, we can evaluate an inaccurate the value of middle of the path. Adopting this method is similar to reduction which could boost the independency of our strategy. Because of huge amount of paths and limitation in threads, we must allocate multi-paths for each threads. In what follows, we demonstrate how this method works.

Function 3: estimate the price of stocks in second method

```
    while (tid < N)
    {
       for ( int i = 0 ; i < 200 ; i++)
       {
          temp = temp*exp((mu-0.5*V_pow)*T +
V*sqrt(T)*randnum[tid]);

          output[tid*200 + i] = temp;

       }
       tid += blockDim.x * gridDim.x;
       temp = s;
    }
```

At the end of this experiment we obtain a valuable result in such a way that the performance of our algorithm increased 7.3x to the MATLAB version of this code.

All of these processes, from generating the random numbers to calculating the price of stocks, executed in 156.3_{ms} in best case. It should be also pointed out that the amount of paths in this experience is 131,072 and 200 steps for each path which means that 26,214,400 amount of steps.

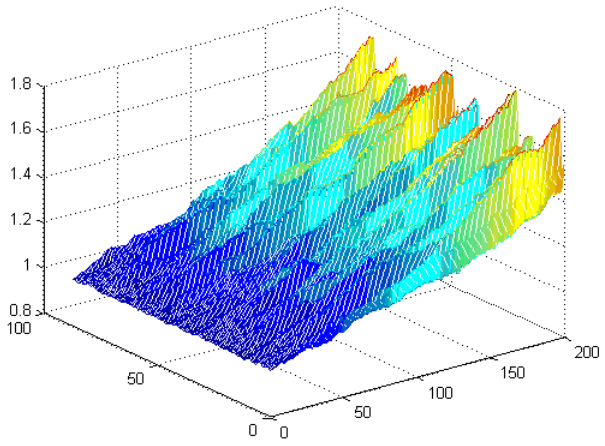

Figure 1: scheme of some of paths

According to Figure 1, this algorithm is ascending which means that the price of our stocks mostly increase with passage of time. At first, all of the values started from unit and end up around1.5. Another issue that should not go unnoticed is that this implementation and algorithm based on European method. One another strategy which is more precise in comparison to European model is American. In American method we should use some recursive and regressions which is another part of our project.

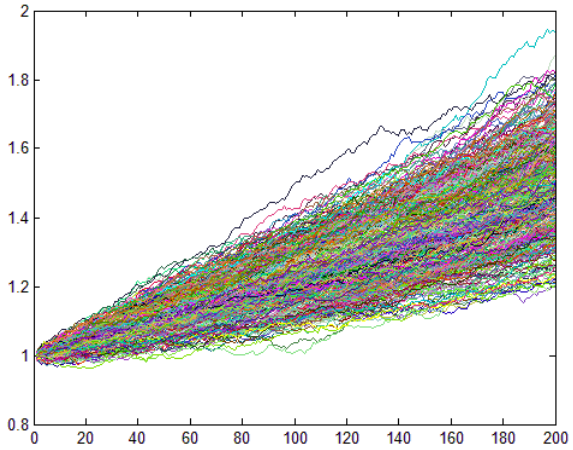

Figure 2: steps of 1024 paths