The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

Introduction to GPU programming

Practice 10. Monte Carlo integration and option pricing

Nizhni Novgorod

2014

Author: S.I. Bastrakov

OBJECTIVES

The objective of this practice is to apply CURAND for Monte Carlo method applied to two problems: integration and option pricing.

ABSTRACT

This practice considers two important applications of Monte Carlo method: integration and option pricing. We demonstrate how to use optimized CURAND library via both Host and Device API to solve these problems.

BRIEF OVERVIEW

We consider pricing of Call European options. A price follows a stochastic model. Monte Carlo simulation is widely used for such simulations. It is performed as follows. Generate a large number of pseudorandom numbers. Computation for each specific number is called Monte Carlo path. For each path we compute price. Result is average of prices of all paths.

Let r be pseudorandom number of standard normal distribution. Price of an option is computed as:

$$Price(r) = \max\{s * e^{\mu+v*r} - x,\ 0\}$$

If computation is performed with numbers $r_1, r_2, \dots, r_n$, the result is:

$$Price = \frac{1}{n}\sum_{i=1}^{n} Price(r_i) = \frac{1}{n}\sum_{i=1}^{n} \max\{s * e^{\mu+v*r_i} - x,\ 0\}$$

To perform such simulation we need pseudorandom numbers of standard normal distribution. Those can be obtained from numbers uniformly distributed on [0, 1] using the second Box-Muller transform. Let u_1, u_2 be numbers on $[0, 1]$ and $u_1 \neq 0$. Then the following numbers are normally distributed with mean 0 and variance 1:

$$p_1 = \sqrt{-2\ln u_1}\cos(2\pi u_2)$$

$$p_2 = \sqrt{-2\ln u_1}\sin(2\pi u_2)$$

In the case we consider the fair price can be analytically computed by Black-Scholes formula. We compare result of Monte-Carlo simulation with the analytical result. It must converge with order 1/2, error of Monte Carlo simulation is proportional to $1/\sqrt{n}$.

Implementation on CPU is as follows:

```
double endCallValue(double s, double x, double r, double mu, double w)
{
```

```
    double callValue = s * exp(mu + w * r) - x;
    return (callValue > 0) ? callValue : 0;
}
double MonteCarloCPU(int n, double s, double x, double t, double r, double v)
{
    const double mu = (r - 0.5 * v * v) * t;
    const double w = v * sqrt(t);
    double * rnd = new double[n + 1];
    srand(12345);
    for (int i = 0; i < n; i += 2) {
        double u1;
        do {
            u1 = rand() / (double)RAND_MAX;
        } while (u1 == 0);
        double u2 = rand() / (double)RAND_MAX;
        rnd[i] = sqrt(-2.0 * log(u1)) * cos(2.0 * M_PI * u2);
        rnd[i + 1] = sqrt(-2.0 * log(u1)) * sin(2.0 * M_PI * u2);
    }
   double sum = 0;
    for (int i = 0; i < n; i++)
    {
        double callValue = endCallValue(s, x, rnd[i], mu, w);
        sum += callValue;
    }
    delete [] rnd;
    return exp(-r * t) * sum / (double)n;
}
```

Implementation on GPU is left for individual work. We consider 3 versions: naive, using CURAND external, using CURAND internal.

GPU implementation of Monte-Carlo integral using external CURAND:

```
__global__ void kernel_curand_external(int n, float a, float b, float c,
float d,
    float * x01, float * y01, char * count)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n)
        return;
    if (y01[i] * (d - c) + c  <= func_device(x01[i] * (b - a) + a))
        count[i] = 1;
    else
        count[i] = 0;
}
float integrateMonteCarlo gpu curand external(int n, float a, float b, float
c,
    float d, float * x01_device, float * y01_device,
    char * count_host, char * count_device)
{
    curandGenerator_t gen;
    curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_XORWOW);
    curandGenerateUniform(gen, x01_device, n);
    curandGenerateUniform(gen, y01_device, n);
    curandDestroyGenerator(gen);
const int num_threads_per_block = 256;
    int num_blocks = (n + num_threads_per_block - 1) / num_threads_per_block;
    kernel_curand_external<<<num_blocks, num_threads_per_block>>>(n, a, b, c,
d,
        x01_device, y01_device, count_device);
    cudaThreadSynchronize();
    cudaMemcpy(count_host, count_device, n * sizeof(char),
        cudaMemcpyDeviceToHost);
```

```
    int count = 0;
    for (int i = 0; i < n; ++i)
        if (count_host[i])
            ++count;
    return (float)count / (float)n;
}
```

GPU implementation of Monte-Carlo integral using internal CURAND:

```
__global__ void kernel_curand_internal(int n, float a, float b, float c,
float d,  char * count) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n)  return;
    curandStateXORWOW_t state;
    curand_init(2 * i, 0, 0, &state);
    float x = curand_uniform(&state) * (b - a) + a;
    float y = curand_uniform(&state) * (d - c) + c;
    if (y <= func_device(x))   count[i] = 1;
    else  count[i] = 0;
}
float integrateMonteCarlo_gpu_curand_internal(int n, float a, float b, float
c, float d, char * count_host, char * count_device)
{
    const int num_threads_per_block = 256;
    int num_blocks = (n + num_threads_per_block - 1) / num_threads_per_block;
    kernel_curand_internal<<<num_blocks, num_threads_per_block>>>(n, a, b, c,
d,
        count_device);
    cudaThreadSynchronize();
    cudaMemcpy(count_host, count_device, n * sizeof(char),
        cudaMemcpyDeviceToHost);
     int count = 0;
    for (int i = 0; i < n; ++i)
        if (count_host[i])
            ++count;
    return (float)count / (float)n;
}
```

FOR STUDENTS

Detailed information about CURAND is presented in [1].

REFERENCES

1. NVIDIA CURAND Documentation
   [http://docs.nvidia.com/curand/index.html#axzz3JRcPurfI].

INDIVIDUAL WORK

1. Generalize our Monte Carlo integration for integrals of arbitrary dimension.
2. Implement three versions of Monte Carlo option pricing on GPU: naive, using CURAND external, using CURAND internal. Compare performance of these versions.
3. Modify implementation to compute prices of several options with different parameters.
4. Create multi-GPU implementation for multi-option case, each GPU handles one or several options.