


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”

Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Introduction to GPU programming

10 Practice
Monte Carlo integration and option pricing

Bastrakov S.I.
Software department

Contents

- Monte Carlo integration
- Monte Carlo option pricing

Monte Carlo integration

Implementation on CPU

```
float func(float x) {
    return 1.0f / (1.0f + x * x);
}
float integrateMonteCarlo(int n, float a, float b, float c, float d) {
    int count = 0;
    for (int i = 0; i < n; i++)     {
        float x = (float)rand() / (float)RAND_MAX * (b - a) + a;
        float y = (float)rand() / (float)RAND_MAX * (d - c) + c;
        if (y <= func(x))   ++count;
    }
    return (float)count / (float)n;
}
```

Naive implementation on GPU…

```
__device__ float func_device(float x) {
    return 1.0f / (1.0f + x * x);
}
__global__ void kernel_naive(int n, float * x, float * y, char *
count) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n)
        return;
    if (y[i] <= func_device(x[i]))   count[i] = 1;
    else  count[i] = 0;
}
```

Naive implementation on GPU…

```
float integrateMonteCarlo_gpu_naive(int n, float a, float b, float
c, float d, float * x_host, float * x_device, float * y_host, float *
y_device, char * count_host, char * count_device)
{
    for (int i = 0; i < n; i++) {
        x_host[i] = (float)rand() / (float)RAND_MAX * (b - a) + a;
        y_host[i] = (float)rand() / (float)RAND_MAX * (d - c) + c;
    }
    cudaMemcpy(x_device, x_host, n * sizeof(float),
cudaMemcpyHostToDevice);
    cudaMemcpy(y_device, y_host, n * sizeof(float),
cudaMemcpyHostToDevice);
```

Naive implementation on GPU…

```
    const int num_threads_per_block = 256;
    int num_blocks = (n + num_threads_per_block - 1) /
num_threads_per_block;
    kernel_naive<<<num_blocks, num_threads_per_block>>>(n,
x_device, y_device,
        count_device);
    cudaThreadSynchronize();
   cudaMemcpy(count_host, count_device, n * sizeof(char),
        cudaMemcpyDeviceToHost);
```

Naive implementation on GPU

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

Implementation 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;
}
```

Implementation using external CURAND…

```
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);
```

Implementation using external CURAND…

```
    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);
```

Implementation using external CURAND

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

Implementation 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;
}
```

Implementation using internal CURAND…

```
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);
```

Implementation using internal CURAND…

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

Function main…

```
int main()
{
    const int n = 100000;
    float a = 0.0f, b = 1.0f;
    float c = 0.0f, d = 1.0f;
    float exactResult = atan(b) - atan(a);
    const int seed = 0;
    const int repeats = 100;
    float result, result_gpu_naive, result_gpu_curand_external,
        result_gpu_curand_internal;
```

Function main…

```
LARGE_INTEGER before, after, freq;
QueryPerformanceFrequency(&freq);
std::cout << std::fixed << std::setprecision(3);
QueryPerformanceCounter(&before);
srand(seed);
for (int i = 0; i < repeats; ++i)
    result = integrateMonteCarlo(n, a, b, c, d);
QueryPerformanceCounter(&after);
double cpu_time = (after.QuadPart - before.QuadPart) /
    double(freq.QuadPart);
std::cout << "The CPU version took " << cpu_time << "
seconds.\n";
```

Function main…

```
QueryPerformanceCounter(&before);
float * x_host = new float[n], * y_host = new float[n];
char * count_host = new char[n];
float * x_device, * y_device;
char * count_device;
cudaMalloc((void**)&x_device, n * sizeof(float));
cudaMalloc((void**)&y_device, n * sizeof(float));
cudaMalloc((void**)&count_device, n * sizeof(char));
srand(seed);
for (int i = 0; i < repeats; ++i)
    result_gpu_naive = integrateMonteCarlo_gpu_naive(n, a, b,c,
d, x_host, x_device, y_host, y_device, count_host,  count_device);
```

Function main…

```
    delete [] x_host;
    delete [] y_host;
    delete [] count_host;
    cudaFree(x_device);
    cudaFree(y_device);
    cudaFree(count_device);
    QueryPerformanceCounter(&after);
    double gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The GPU naive version took " << gpu_time << "
seconds, ";
    std::cout << "speedup: " << cpu_time / gpu_time << "
times.\n";
```

Function main…

```
QueryPerformanceCounter(&before);
count_host = new char[n];
float * x01_device, * y01_device;
cudaMalloc((void**)&x01_device, n * sizeof(float));
cudaMalloc((void**)&y01_device, n * sizeof(float));
cudaMalloc((void**)&count_device, n * sizeof(char));
for (int i = 0; i < repeats; ++i)
    result_gpu_curand_external =
        integrateMonteCarlo_gpu_curand_external(n, a, b, c, d,
            x01_device, y01_device, count_host, count_device);
```

Function main…

```
    delete [] count_host;
    cudaFree(x01_device);
    cudaFree(y01_device);
    cudaFree(count_device);
    QueryPerformanceCounter(&after);
    gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The GPU CURAND external version took " <<
gpu_time << " seconds, ";
    std::cout << "speedup: " << cpu_time / gpu_time << "
times.\n";
```

Function main…

```
QueryPerformanceCounter(&before);
count_host = new char[n];
cudaMalloc((void**)&count_device, n * sizeof(char));
for (int i = 0; i < repeats; ++i)
    result_gpu_curand_internal =
        integrateMonteCarlo_gpu_curand_internal(n, a, b, c, d,
            count_host, count_device);
delete [] count_host;
```

Function main…

```
    cudaFree(count_device);
    QueryPerformanceCounter(&after);
    gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "The GPU CURAND internal version took " <<
gpu_time << " seconds, ";
    std::cout << "speedup: " << cpu_time / gpu_time << "
times.\n";
```

Function main

```
    std::cout << "\n";
    std::cout << "Exact              result: " << exactResult << "\n";
    std::cout << "CPU                result: " << result << "\n";
    std::cout << "GPU naive          result: " << result_gpu_naive
<< "\n";
    std::cout << "GPU CURAND external result: " <<
result_gpu_curand_external << "\n";
    std::cout << "GPU CURAND internal result: " <<
result_gpu_curand_internal << "\n";
}
```

Monte Carlo option pricing

Problem statement

- Pricing of Call European options
- Stochastic model, use Monte Carlo simulation
- 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

Problem statement

- 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 $r_1, r_2, \ldots, r_n$, the result is:

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

Generating pseudorandom numbers of normal distribution

- Can be obtained from numbers uniformly distributed on [0, 1]
- Use 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:

$$p_1 = \sqrt{-2 \ln u_1} \cos(2\pi u_2)$$
$$p_2 = \sqrt{-2 \ln u_1} \sin(2\pi u_2)$$

Correctness analysis

- 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…

```
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);
```

Implementation on CPU…

```
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);
 }
```

Implementation on CPU

```
    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

- Implementation on GPU is left for individual work.
- Implement three versions following the pattern in Monte Carlo integration:
  - Naive
  - Using CURAND external
  - Using CURAND internal

Function main…

```
double CND(double d) {
    const double A1 = 0.31938153; const double A2 = -
0.356563782;  const double A3 = 1.781477937;  const double
A4 = -1.821255978;  const double A5 = 1.330274429;
    const double RSQRT2PI =
0.39894228040143267793994605993438;
    double K = 1.0 / (1.0 + 0.2316419 * fabs(d));
    double cnd = RSQRT2PI * exp(- 0.5 * d * d) *
          (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));
    if (d > 0)
        cnd = 1.0 - cnd;
    return cnd;
}
```

Function main…

```
double BlackScholes(double s, double x, double t, double r,
double v)
{
    double sqrtT = sqrt(t);
    double    d1 = (log(s / x) + (r + 0.5 * v * v) * t) / (v * sqrtT);
    double    d2 = d1 - v * sqrtT;
    double CNDD1 = CND(d1);
    double CNDD2 = CND(d2);
    double expRT = exp(- r * t);
    return s * CNDD1 - x * expRT * CNDD2;
}
```

Function main…

```
int main() {
    int n = 100000000;
    double s = 24;
    double x = 22.0;
    double t = 3.0;
    double r = 0.06;
    double v = 0.1;
    double exactPrice = BlackScholes(s, x, t, r, v);
    std::cout << "Exact price: " << exactPrice << "\n\n";
    LARGE_INTEGER before, after, freq;
    QueryPerformanceFrequency(&freq);
```

Function main…

```
QueryPerformanceCounter(&before);
    double priceCPU = MonteCarloCPU(n, s, x, t, r, v);
    QueryPerformanceCounter(&after);
    double cpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "CPU result: " << priceCPU << ", test " <<
        (fabs(priceCPU - exactPrice) < 0.01 * exactPrice ?
"PASSED\n" : "FAILED\n");
    std::cout << "CPU time: " << cpu_time << " seconds\n\n";}
```

Function main…

```
 QueryPerformanceCounter(&before);
    double priceGPU = MonteCarloGPU_naive(n, s, x, t, r, v);
    QueryPerformanceCounter(&after);
    double gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "GPU naive result: " << priceGPU << ", test " <<
        (fabs(priceGPU - exactPrice) < 0.01 * exactPrice ?
"PASSED\n" : "FAILED\n");
    std::cout << "GPU naive time: " << gpu_time << "
seconds\n\n";
```

Function main…

```
 QueryPerformanceCounter(&before);
    priceGPU = MonteCarloGPU_CURAND_external(n, s, x, t, r,
v);
    QueryPerformanceCounter(&after);
    gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "GPU external CURAND result: " << priceGPU
<< ", test " <<
        (fabs(priceGPU - exactPrice) < 0.01 * exactPrice ?
"PASSED\n" : "FAILED\n");
    std::cout << "GPU external CURAND time: " << gpu_time << "
seconds\n\n";
```

Function main

```
 QueryPerformanceCounter(&before);
    priceGPU = MonteCarloGPU_CURAND_internal(n, s, x, t, r,
v);
    QueryPerformanceCounter(&after);
    gpu_time = (after.QuadPart - before.QuadPart) /
double(freq.QuadPart);
    std::cout << "GPU internal result: " << priceGPU << ", test "
<<  (fabs(priceGPU - exactPrice) < 0.01 * exactPrice ?
"PASSED\n" : "FAILED\n");
    std::cout << "GPU internal time: " << gpu_time << "
seconds\n\n";
    return 0;
}
```

Individual work

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

References

- NVIDIA CURAND Documentation [http://docs.nvidia.com/curand/index.html#axzz3JRcPurfl]

Authors

- Bastrakov S.I.,
  Assistant of the Software department of CMC faculty.
  bastrakov@vmk.unn.ru