FOUNTAINHEAD

Monte-Carlo Using CUDA Thrust

~ *A Practitioner's Guide* ~

Andrew Sheppard

Baruch College MFE, "Big Data in Finance" Course
29th January 2013

FOUNTAINHEAD

Objectives

In this talk I will cover:

1. Elements of the Monte-Carlo method, a short review.
2. Monte-Carlo on GPUs.
3. Monte-Carlo using CUDA Thrust.
4. Simple benchmark numbers.

Fountainhead

~ 1. Elements of Monte-Carlo ~

FOUNTAINHEAD

Elements

Random-Number Generation → Data Set Generation → Function Evaluation → Statistical Aggregation

Typical Monte-Carlo simulation steps (simplified):

1. Generate random numbers.
2. Data set generation.
3. Function evaluation.
4. Aggregation.

Fountainhead

Random Number Generation (RNG)

Pre-generate or on-the-fly? Pros (✔) and cons (✖):

	Pre-generate	***On-the-fly***
Time	✔	✖ (✔ sometimes)
Storage	✖	✔
Backtest	✔	✖
Quality	✔	✖

FOUNTAINHEAD

Parallel RNG (PRNG)

In choosing a RNG there are the conflicting goals of speed and quality (randomness). Challenges and benefits:

- Challenge: Quality (avoiding artifacts and avoiding correlation or overlap across nodes and devices).
- Challenge: PRNG algorithms.
- Benefit: Parallel generation (speed).
- Benefit: Co-location of data with compute (by default).

FOUNTAINHEAD

RNG Algorithms

Many choices. What’s best? Depends …

- XORWOW PRNG.
- Sobol RNG.
- Niederreiter RNG.
- Mersenne Twister PRNG.
- Tausworth, Sobol and L’Ecuyer.
- Brownian bridge generation.

FOUNTAINHEAD

CUDA RNG

PRNG (must be parallel RNGs) on GPUs poses some challenges:

- Linear Congruential Generator, or LCG (poor statistics).
- Multiple Recursive Generator, or MRG (poor statistics).
- Lagged Fibonacci Generator, or LFG (poor statistics).
- Mersenne Twister (good statistics, but slow).
- Combined Tausworthe Generator (poor statistics).

CUDA RNG (cont.)

- Hybrid Generator for which defects of one RNG are compensated for by another RNG - example, Tausworthe + LCG (see GPU GEMS 3).
- If pre-generation of random numbers is an option, take it as it will likely save a lot of time.
- CURAND and other RNG libs.

Data Set Generation

Important things to bear in mind:

- Device storage space (unless generated on-the-fly).
- Data transfer to/from device and across cluster.
- Type of memory storage (global, constant, texture).
- Ease of traversal of the data set (data structures).
- Data management for back/regression testing.

Function Evaluation

Fast evaluation techniques:

- Precision (`float` is faster than `double`).
- Approximations and lookups.
- Branching in GPU kernels is costly to performance.
- Use GPU optimized libraries (CUBLAS, CURAND, …).
- Use GPU optimized data structures and algorithms (such as CUDA Thrust).

Aggregation

Need to statistically aggregate results to arrive at an answer:

- Use parallel sum-reduction techniques.
- Use parallel sort to compute quantiles and other results.

In the case of HPC+GPU and Cloud+GPU, need to aggregate at two levels: 1) GPU and 2) Cluster.

Fountainhead

~ 2. Monte-Carlo on GPUs ~

Guiding Principles for CUDA Monte-Carlo

General guiding principles:

- Understand the different types of GPU memory and use them well.
- Launch sufficient threads to fully utilize GPU cores and hide latency.
- Branching has a big performance impact; modify code or restructure problem to avoid branching.

Guiding Principles for CUDA Monte-Carlo (cont.)

- Find out where computation time is spent and focus on performance gains accordingly; from experience, oftentimes execution time is evenly split across the first three stages (before aggregation).
- Speed up function evaluation by being pragmatic about precision, using approximations and lookup tables, and by using GPU-optimized libraries.

Guiding Principles for CUDA Monte-Carlo (cont.)

- Statistical aggregation should use parallel constructs (e.g., parallel sum-reduction, parallel sorts).
- Use GPU-efficient code: GPU Gems 3, Ch. 39; CUDA SDK reduction; MonteCarloCURAND; CUDA SDK radixSort.
- And, as always, parallelize pragmatically and wisely!

Fountainhead

~ 3. Monte-Carlo Using Thrust ~

FOUNTAINHEAD

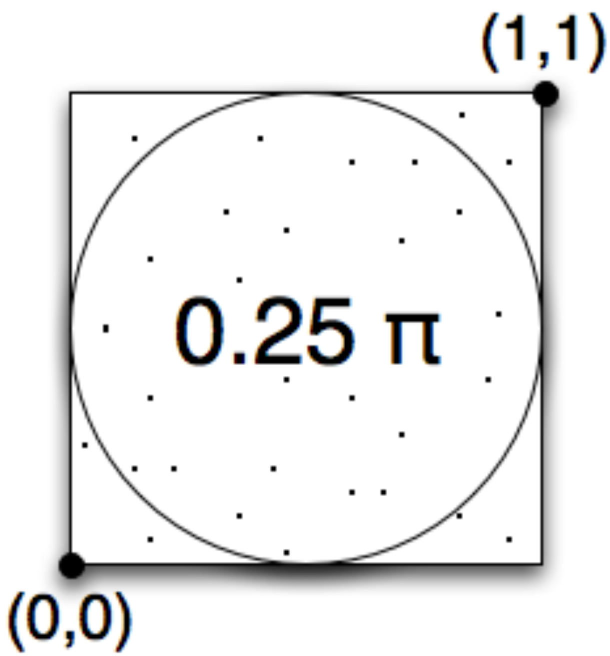
Example: Monte-Carlo using CUDA Thrust

Let's consider a simple example of how Monte-Carlo can be mapped onto GPUs using CUDA Thrust.

CUDA Thrust is a C++ template library that is part of the CUDA toolkit and has containers, iterators and algorithms; and is particularly handy for doing Monte-Carlo on GPUs.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)



This is a very simple example that estimates the value of the constant `PI` while illustrating the key points when doing Monte-Carlo on GPUs.

(As an aside, it also demonstrates the power of CUDA Thrust.)

Example: Monte-Carlo using CUDA Thrust (cont.)

```
int main() {
    size_t N = 50000000; // Number of Monte-Carlo simulations.
    // DEVICE: Generate random points within a unit square.
    thrust::device_vector<float2> d_random(N);
    thrust::generate(d_random.begin(), d_random.end(), random_point());
    // DEVICE: Flags to mark points as lying inside or outside the circle.
    thrust::device_vector<unsigned int> d_inside(N);
    // DEVICE: Function evaluation. Mark points as inside or outside.
    thrust::transform(d_random.begin(), d_random.end(),
                      d_inside.begin(), inside_circle());
    // DEVICE: Aggregation.
    size_t total = thrust::count(d_inside.begin(), d_inside.end(), 1);
    // HOST: Print estimate of PI.
    std::cout << "PI: " << 4.0*(float)total/(float)N << std::endl;
    return 0;
}
```

Example: Monte-Carlo using CUDA Thrust (cont.)

```
struct random_point {
private:
    thrust::default_random_engine rng;
public:
    __device__ __host__
    float2 operator()(int index) {
        rng.discard(2*index);
        return make_float2(
                (float)rng() / thrust::default_random_engine::max,
                (float)rng() / thrust::default_random_engine::max);
    }
};
```

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
struct inside_circle {
private:
    __device__ __host__
    unsigned int inside(float2 p) const {
    return (((p.x-0.5)*(p.x-0.5)+(p.y-0.5)*(p.y-0.5))<0.25) ? 1 : 0;
    }
public:
    // Used for-on-the fly.
    __device__ __host__
    unsigned int operator()(int index) const {
        // Generate a random point.
        random_point point;
        return inside(point(index));
    }
};
```

Example: Monte-Carlo using CUDA Thrust (cont.)

Let's look at the code and how it relates to the steps (elements) of Monte-Carlo.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
// DEVICE: Generate random points within a unit square.
thrust::device_vector<float2> d_random(N);
thrust::generate(d_random.begin(), d_random.end(), random_point());
```

STEP 1: Random number generation. Key points:

- Random numbers are generated in parallel on the GPU.
- Data is stored on the GPU directly, so co-locating the data with the processing power in later steps.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

STEP 2: Generate simulation data. Key points:

- In this example, the random numbers are used directly and do not need to be transformed into something else.
- If higher level simulation data is needed, then the same principles apply: ideally, generate it on the GPU, store the data on the device, and operate on it in-situ.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
// DEVICE: Flags to mark points as lying inside or outside the circle.
thrust::device_vector<unsigned int> d_inside(N);
// DEVICE: Function evaluation. Mark points as inside or outside.
thrust::transform(d_random.begin(), d_random.end(),
                  d_inside.begin(), inside_circle());
```

STEP 3: Function evaluation. Key points:

- Function evaluation is done on the GPU in parallel.
- Work can be done on the simulation data in-situ because it was generated & stored on the GPU directly.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

```
// DEVICE: Aggregation.
size_t total = thrust::count(d_inside.begin(), d_inside.end(), 1);
// HOST: Print estimate of PI.
std::cout << "PI: " << 4.0*(float)total/(float)N << std::endl;
```

STEP 4: Aggregation. Key points:

- Aggregation is done on the GPU using parallel constructs and highly GPU-optimized algorithms (courtesy of Thrust).
- Data has been kept on the device throughout and only the final result is transferred back to the host.

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

Key takeaways from this example:

- Use the tools! CUDA Thrust is a very powerful abstraction tool for doing Monte-Carlo on GPUs.
- It's efficient too, as it generates GPU optimized code.
- Do as much work on the data as possible in-situ, and in parallel. Only bring back to the host the minimum you need to get an answer.

FOUNTAINHEAD

~ 4. Benchmark Numbers ~

FOUNTAINHEAD

Example: Monte-Carlo using CUDA Thrust (cont.)

Results for N = 50,000,000 data points (simulations)

	Pre-Compute Random Numbers	***On-the-Fly Random Numbers***
Intel Core2 Quad Core (4 cores) [1]	4.4 seconds	4.0 seconds
Nvidia GTX 560 Ti (384 cores) [2]	0.4 seconds	0.25 seconds

[1] C++ serial code. [2] C++ CUDA Thrust parallel code