F ountainhead

CUDA-Accelerated Monte-Carlo for HPC

~ *A Practitioner's Guide* ~

Andrew Sheppard

SC11, Seattle, WA

12-18 November 2011

F ountainhead

Objectives

In this tutorial I will cover:

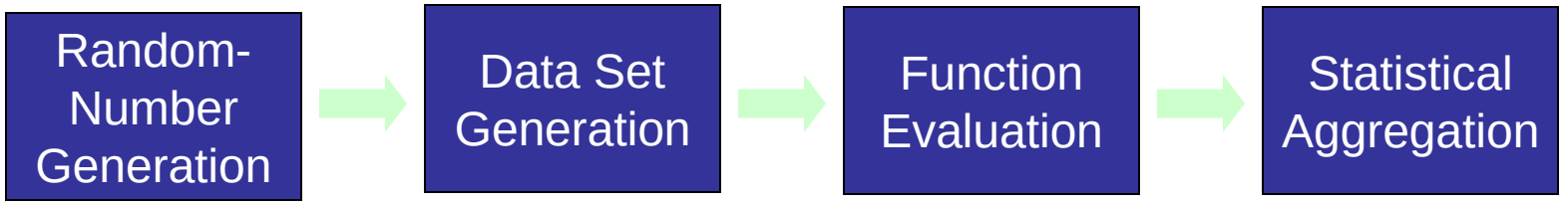
1. Elements of the Monte-Carlo method, a short review.
2. Monte-Carlo on GPUs.
3. Guidance on moving Monte-Carlo to HPC+GPU and Cloud+GPU.
4. Demo of Monte-Carlo on Cloud+GPU.

F ountainhead

~ 1. Elements of Monte-Carlo ~

F ountainhead

Elements



Typical Monte-Carlo simulation steps (simplified):

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

F ountainhead

Random Number Generation (RNG)

Pre-generate or on-the-fly? Pros (📄) and cons (⏳):

	Pre-generate	***On-the-fly***
Time	📄	⏳ (📄 sometimes)
Storage	⏳	📄
Backtest	📄	⏳
Quality	📄	⏳

F ountainhead

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).

F ountainhead

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.

F ountainhead

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).

F ountainhead

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.

F ountainhead

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.

F ountainhead

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).

F ountainhead

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.

F ountainhead

~ 2. Monte-Carlo on GPUs ~

F ountainhead

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.

F ountainhead

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.

F ountainhead

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!

F ountainhead

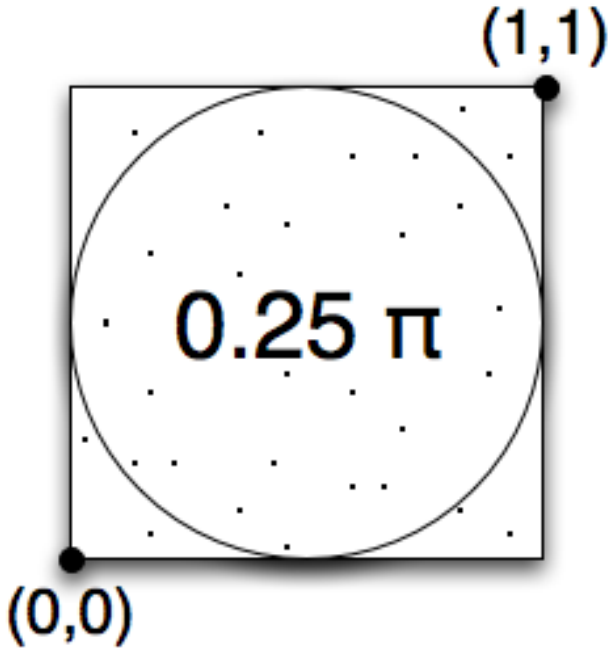
Example: Monte-Carlo using CUDA Thrust

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.

F ountainhead

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.)

F ountainhead

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;
}
```

F ountainhead

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

```
struct random_point {
    __device__
    float2 operator()() {
        thrust::default_random_engine rng;
        return make_float2(
                (float)rng() / thrust::default_random_engine::max,
                (float)rng() / thrust::default_random_engine::max);
    }
};
```

F ountainhead

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

```
struct inside_circle {
    __device__
    unsigned int operator()(float2 p) const {
    return (((p.x-0.5)*(p.x-0.5)+(p.y-0.5)*(p.y-0.5))<0.25) ? 1 : 0;
    }
};
```

F ountainhead

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

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

F ountainhead

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.

F ountainhead

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.

F ountainhead

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.

F ountainhead

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.

F ountainhead

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

F ountainhead

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.

F ountainhead

Example: Texture Memory

- To speed up Monte-Carlo on GPUs use the memory model efficiently, use hardware defined operations whenever possible, pre-compute when you can, and be willing to trade-off accuracy for speed.
- The following example illustrates all these approaches in speeding up the function evaluation part (STEP 3) of Monte-Carlo.

F ountainhead

Example: Texture Memory (cont.)

- In cases where complex simulation data (generated from RNG data) is used as input to function evaluation and is expensive to compute, consider pre-computing a lookup table and using interpolation.
- Consider, for example, a surface function that is computationally expensive to generate. Putting it in texture memory can speed things up.

F ountainhead

Example: Texture Memory (cont.)

```
// 2D texture memory lookup table.
texture<float, 2, cudaReadModeElementType> tex;

// Function evaluation kernel.
__global__ void kernel(float *g_results, int width, int height,
                       float param)
{
    // Normalized texture coordinates.
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
    float u = x / (float)width;
    float v = y / (float)height;
    // Lookup value.
    float lookup = tex2D(tex, u, v);
    // Do some additional calculations using lookup value.
    result = ... ;
    // Store result.
    g_results[y*width + x] = result;
}
```

F ountainhead

Example: Texture Memory (cont.)

```
int main(int argc, char **argv)
{
    // Obtain width, height and param from command line.
    ...
    // Allocate host memory for lookup table.
    int sizeLookup = width * height * sizeof(float);
    float *h_lookup = (float*)malloc(sizeLookup);
    // Build lookup table. (Alternatively, read from disk.)
    for (int i=0; i<width; i++) {
        for (int j=0; j<height; j++) {
                h_lookup[width*j+i] = expensive_compute(i,j);
        }
    }
    // Allocate device memory for results.
    float *g_results; cudaMalloc((void**)&g_results, sizeLookup);
...
```

F ountainhead

Example: Texture Memory (cont.)

```
...
    // Allocate CUDA array.
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(32, 0, 0, 0,
    cudaChannelFormatKindFloat);
    cudaArray* cu_array;
    cudaMallocArray(&cu_array, &channelDesc, width, height);
    cudaMemcpyToArray(cu_array, 0, 0, h_lookup, sizeLookup,
                      cudaMemcpyHostToDevice);
    // Texture parameters.
    tex.addressMode[0] = tex.addressMode[1] =
         cudaAddressModeClamp;
    tex.filterMode = cudaFilterModeLinear;
    tex.normalized = true;
    // Bind CUDA array to the texture.
    cudaBindTextureToArray(tex, cu_array, channelDesc);
...
```

F ountainhead

Example: Texture Memory (cont.)

```
...
    // Launch kernel.
    kernel<<< dimGrid, dimBlock, 0 >>>(g_results, width, height,
                                       param);

    cudaThreadSynchronize();

    // Copy results to host and print them out. Tidy up by
    // freeing memory. Exit program.
    ...
    return 0;
}
```

F ountainhead

Example: Texture Memory (cont.)

Key takeaways from this example:

- Simulation data that is computationally expensive to generate should be pre-computed.
- Be flexible and always consider the trade-off between accuracy and speed.
- Use the CUDA memory model to advantage, and hardware features, such as interpolation.

F ountainhead

~ 3. HPC+GPU & Cloud+GPU Monte-Carlo ~

F ountainhead

Guiding Principles for HPC/Cloud+GPU Monte-Carlo

General guiding principles:

- One of the main challenges of moving to HPC+GPU is that you are moving from a homogeneous environment (CUDA) to a heterogeneous environment (MPI+CUDA).
- Not too difficult even now (see example), and things will improve as MPI+GPU tools improve.
- Aggregation takes place across GPUs & cluster nodes.

F ountainhead

HPC+GPU Monte-Carlo Example

```
#include <mpi.h>

void run_monte_carlo_gpu_kernel();

int main(int argc, char *argv[]) /* "Hello World" example for HPC+GPU */
{
    int rank, size;

    generate_random_numbers(); // Pre-generate random data set using CPU.
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    run_monte_carlo_gpu_kernel(); // Evaluate funcs in parallel on GPUs.
    MPI_Finalize();
    aggregate_and_print_results(); // Aggregate results using CPU.
    return 0;
}
```

F ountainhead

HPC+GPU Monte-Carlo Example

```
__global__ void kernel(float *data, float *results, int N) // Runs on device.
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    ...
}

extern "C"
void run_monte_carlo_gpu_kernel() // Runs on cluster node.
{
        // Get chunk of simulation data.
        ...
        // Launch GPU kernel. Evaluate function values.
        ...
        // Save results
        ...
}
```

F ountainhead

Guiding Principles for Cloud+GPU Monte-Carlo

General guiding principles:

- Cloud is moving away from being a destination for data to being the source (origin) of data.
- Co-location of data with compute is a powerful model; make use of it.
- You must write code in a more flexible way to cope with a more variable cluster configuration (Cloud resources come and go and are reconfigured often).

F ountainhead

~ 4. Cloud+GPU Monte-Carlo Demo ~

F ountainhead

Cloud+GPU Demo

Let’s see how some of the guiding principles and practice of CUDA-accelerated Monte-Carlo for HPC can be applied:

- Demo of “Value at Risk” (VaR) on Cloud+GPU.

F ountainhead

About Fountainhead

Andrew Sheppard

❑ ***Fountainhead*** is a consulting, training and recruiting company that specializes in HPC & GPU for financial services. The principals at ***Fountainhead*** are Andrew Sheppard (speaker for this talk) and Didac Artes.

andrew.sheppard
@fountainhead.es

didac.artes
@fountainhead.es

Didac Artes