Hanweck Associates, LLC

Monte Carlo Methods in CUDA

Gerald A. Hanweck, Jr., PhD
CEO, Hanweck Associates, LLC

The Thalesians Quantitative Finance Seminar

The Playwright Tavern, New York City
February 23, 2011

Hanweck Associates, LLC
61 Broadway, Suite 1608
New York, NY 10006
www.hanweckassoc.com
Tel: 1-646-414-7274

Agenda

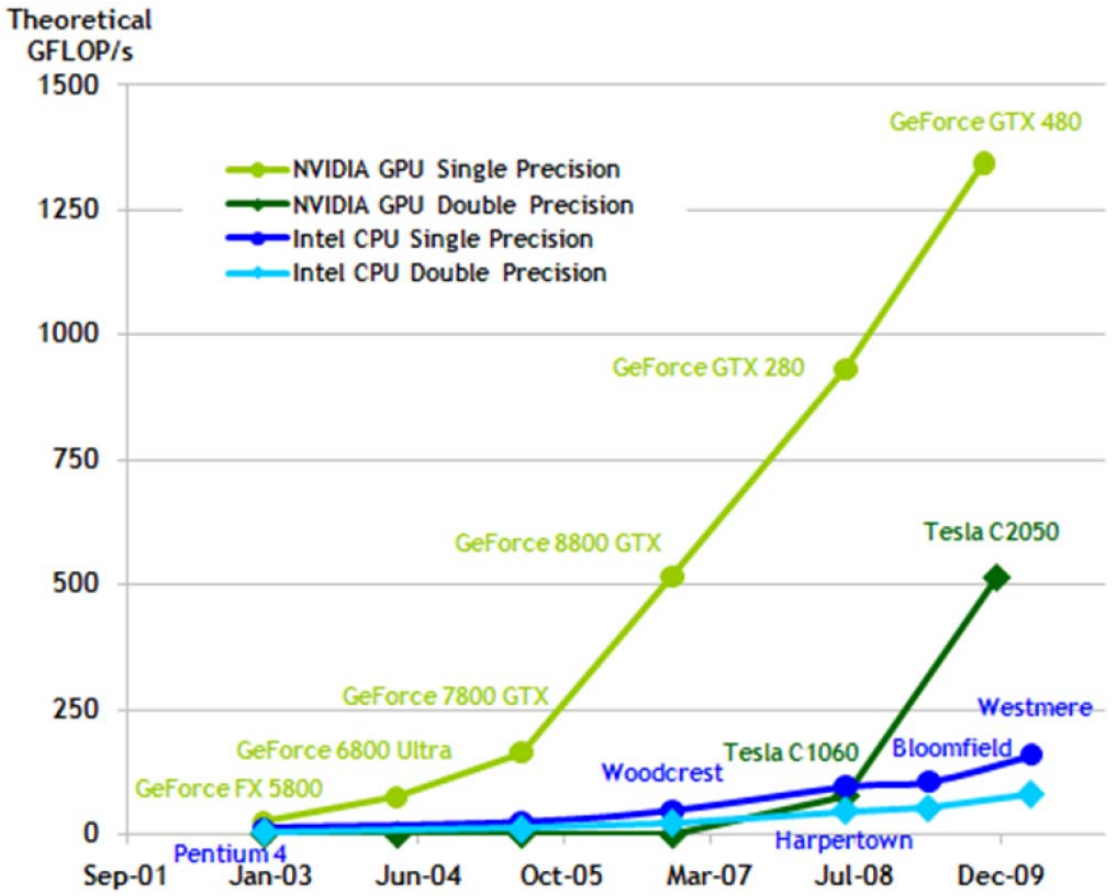
GPU Architecture and CUDA Overview

Monte Carlo Methods in CUDA: Some Guiding Principles

- Random-Number Generation
- Path Generation
- Payoff Functions
- Aggregation

Q&A

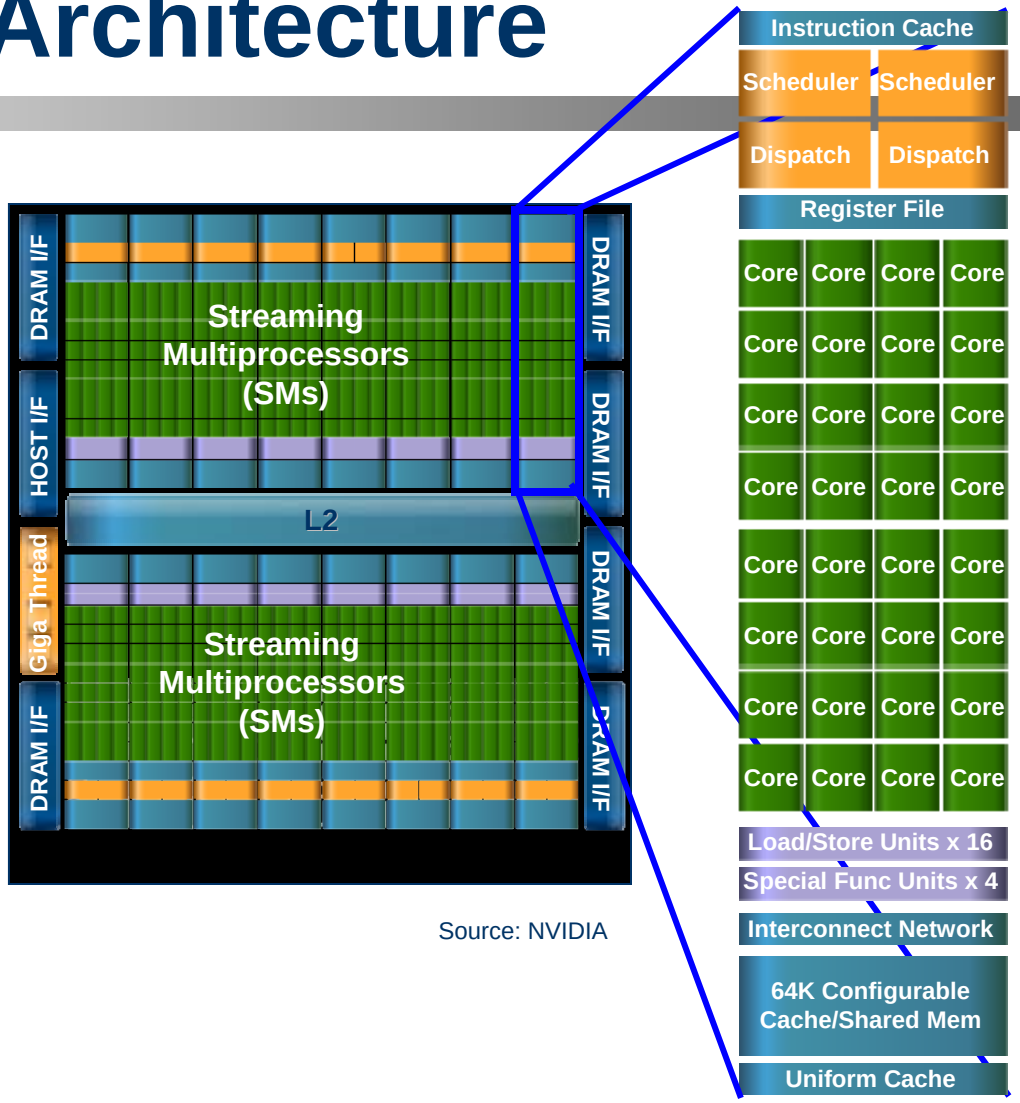
NVIDIA GPU Performance



Source: NVIDIA

NVIDIA GPU Architecture

- 32 CUDA cores per streaming multiprocessor (512 total)
- 8x peak double precision floating point performance (50% of peak single precision)
- Dual Thread Scheduler
- 64 KB of RAM for shared memory and L1 cache (configurable)



Source: NVIDIA

CUDA Overview

Execution

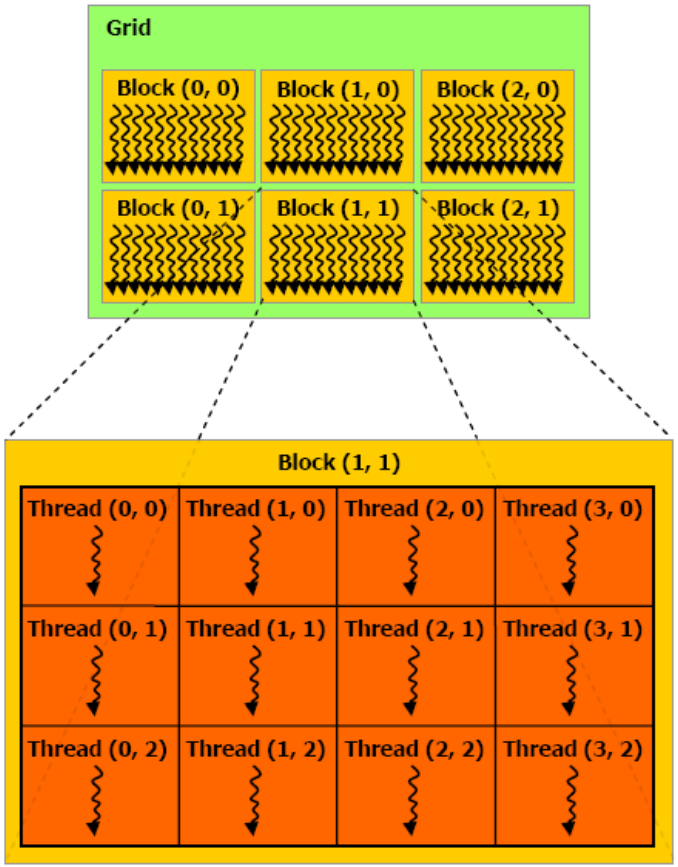
- CPU code passes control to GPU functions called *kernels*
- Data is transferred between CPU to GPU memory via DMA copies

Threading

- A kernel operates on a *grid* of *thread blocks* (up to 65,535 x 65,535 x 65,535)
- Each thread block runs multiple *threads* (up to 1,024 per thread block)
- Threads are grouped into SIMD-like *warps* (32 threads)

Memory

- GPU DRAM, or *global memory*, or *device memory* (multiple gigabytes) with *L2 cache*; generally slow (hundreds of clocks)
- Per SM *shared memory / L1 cache* (64KB) arranged in 32 *banks*; generally fast (2 clocks)
- *Registers*; very fast, but limited resource
- *Constant memory* (64KB shared by all SMs, read only by kernel code)
- *Texture memory* (spatially cached global memory with rudimentary interpolation, up to 65,536 x 65,535 elements, read-only by kernel code)



Source: NVIDIA

CUDA Overview
Example CUDA Code

```
//Preprocessed input option data
typedef struct{
    real S;
    real X;
    real MuByT;
    real VBySqrtT;
} __TOptionData;
static __device__ __constant__ __TOptionData d_OptionData[MAX_OPTIONS];

//GPU outputs before CPU postprocessing
typedef struct{
    real Expected;
    real Confidence;
} __TOptionValue;
static __device__ __TOptionValue d_CallValue[MAX_OPTIONS];

////////////////////////////////////////////////////////////////////////////////
// Overloaded shortcut payoff functions for different precision modes
////////////////////////////////////////////////////////////////////////////////
#ifndef DOUBLE_PRECISION
__device__ inline float endCallValue(float S, float X, float r, float MuByT, float VBySqrtT){
    float callValue = S * __expf(MuByT + VBySqrtT * r) - X;
    return (callValue > 0) ? callValue : 0;
}
#else
__device__ inline double endCallValue(double S, double X, double r, double MuByT, double VBySqrtT){
    double callValue = S * exp(MuByT + VBySqrtT * r) - X;
    return (callValue > 0) ? callValue : 0;
}
#endif
```

Source: NVIDIA SDK *MonteCarlo/MonteCarlo_kernel.cuh*

CUDA Overview
Example CUDA Code (cont'd)

```
////////////////////////////////////////////////////////////////////////////////
// This kernel computes partial integrals over all paths using a multiple thread
// blocks per option. It is used when a single thread block per option would not
// be enough to keep the GPU busy. Execution of this kernel is followed by
// MonteCarloReduce() to get the complete integral for each option.
////////////////////////////////////////////////////////////////////////////////
#define THREAD_N 256
static __global__ void MonteCarloKernel(
    __TOptionValue *d_Buffer,
    float *d_Samples,
    int pathN
){
    const int optionIndex = blockIdx.y;

    const real        S = d_OptionData[optionIndex].S;
    const real        X = d_OptionData[optionIndex].X;
    const real    MuByT = d_OptionData[optionIndex].MuByT;
    const real VBySqrtT = d_OptionData[optionIndex].VBySqrtT;

    //One thread per partial integral
    const int   iSum = blockIdx.x * blockDim.x + threadIdx.x;
    const int accumN = blockDim.x * gridDim.x;

    //Cycle through the entire samples array: derive end stock price for each path
    //accumulate into intermediate global memory array
    __TOptionValue sumCall = {0, 0};
    for(int i = iSum; i < pathN; i += accumN){
        real              r = d_Samples[i];
        real      callValue = endCallValue(S, X, r, MuByT, VBySqrtT);
        sumCall.Expected   += callValue;
        sumCall.Confidence += callValue * callValue;
    }
    d_Buffer[optionIndex * accumN + iSum] = sumCall;
}
```

Source: NVIDIA SDK *MonteCarlo/MonteCarlo_kernel.cuh*

CUDA Overview
Example CUDA Code (cont'd)

```
////////////////////////////////////////////////////////////////////////////////
// Host-side interface to GPU Monte Carlo
////////////////////////////////////////////////////////////////////////////////

...

const int blocksPerOption = (plan->optionCount < 16) ? 64 : 16;
const int          accumN = THREAD_N * blocksPerOption;
const dim3 gridMain(blocksPerOption, plan->optionCount);

MonteCarloKernel<<<gridMain, THREAD_N>>>(
    (__TOptionValue *)plan->d_Buffer,
    plan->d_Samples,
    plan->pathN
);
cutilCheckMsg("MonteCarloKernel() execution failed\n");

MonteCarloReduce<<<plan->optionCount, THREAD_N>>>(
    (__TOptionValue *)plan->d_Buffer,
    accumN
);
cutilCheckMsg("MonteCarloReduce() execution failed\n");

...

cutilSafeCall( cudaMemcpyFromSymbol(
    h_CallValue,
    d_CallValue,
    plan->optionCount * sizeof(__TOptionValue)
) );
```

Source: NVIDIA SDK *MonteCarlo/MonteCarlo_kernel.cuh*

CUDA Overview Performance Tips

Coalesce Global Memory Accesses

- Global memory is read/written as 32-, 64- or 128-byte transactions (naturally aligned)
- A warp can access a naturally aligned contiguous group of 32, 64 or 128 bytes in a single *coalesced* memory transaction
- Uncoalesced memory accesses require multiple transactions, reducing performance
- While L1 and L2 caches can mitigate some of the performance hit, try to coalesce memory accesses as much as possible (e.g., organize data contiguously, use padding)

Avoid Bank Conflicts

- Shared memory is organized into 32 banks
- A *bank conflict* occurs if two or more threads in a warp access different 32-bit words in the same bank
- A warp can access all 32 banks in one transaction as long as there are no bank conflicts
- A bank conflict requires the shared-memory access to be broken up into multiple transactions, reducing performance (sometimes severely)
- Try to avoid bank conflicts if at all possible by organizing data in shared memory appropriately

CUDA Overview
Performance Tips (cont'd)

Avoid Divergent Branches

- All threads in a warp run in SIMD fashion
- If threads take *divergent branches* (e.g., in an if-else clause or a loop), the different paths are serialized, which can severely reduce performance
- Try to avoid divergent branches if possible (e.g., use arithmetic operations, group similar threads together)

Use Constant Memory

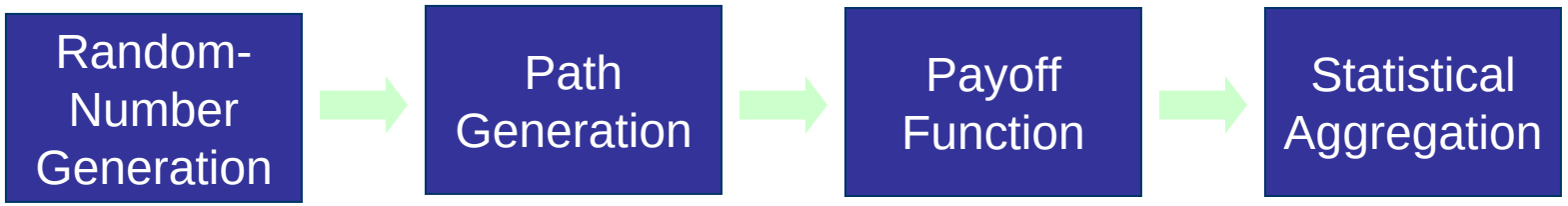
- Useful for static model parameters, covariance matrices, cashflow schedules, call schedules, etc.
- Limited to 64KB, read-only by the kernel
- Fast, particularly when all threads in a warp should read the same location

Use Texture Memory

- Useful for static model parameters, covariance matrices, cashflow schedules, call schedules, etc.
- Can be very large, read-only by the kernel
- 2-D arrays in texture memory benefit from *spatial caching*
- Best performance is achieved when reads are clustered together in the 2-D array

Monte Carlo Methods in CUDA: Some Guiding Principles

A typical MC system consists of four stages:



When designing MC methods in CUDA, consider:

- Typically, CPU computation time is about evenly split between the first three stages; so focus on performance gains across all three
- RNG (including transforms) should be "GPU-efficient" to exploit the hybrid SIMD/MIMD parallelism of the GPU while maintaining statistical quality
- Should RNs be pregenerated or generated "inline"?
- Path generation and payoff functions should use global memory efficiently and avoid divergent branching if possible
- Statistical aggregation should use parallel constructs (e.g., parallel sum-reduction, parallel sorts)
- PARALLELIZE WISELY!!!

Random-Number Generation

Large base of existing GPU code and resources:

- NVIDIA's CURAND RNG library
  http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CURAND_Library.pdf
  - XORWOW PRNG and Sobol QRNG (32-bit)
- NVIDIA's CUDA SDK sample code:
  - Niederreiter and Sobol QRNGs
  - Mersenne Twister PRNG
  - Monte Carlo examples
- *GPU Gems 3* and *GPU Computing Gems* (Emerald Edition)
  GPU Gems 3 is available online: http://developer.nvidia.com/object/gpu-gems-3.html
  - Tausworth, Sobol and L'Ecuyer (MRG32k3a)
  - Monte Carlo examples (*GPU Gems 3*)
- NAG Numerical Routines for GPUs (beta)
  http://www.nag.com/numeric/GPUs/doc.asp
  - Sobol and L'Ecuyer (MRG32k3a)
  - Brownian bridge generation

Random-Number Generation Some Guiding Principles

Pregeneration

- Use GPU kernel(s) (or even CPU code) to generate a bulk set of RNs.
- Store to GPU global memory for use by subsequent path-generation and payoff-function kernels.

Inline Generation

- Build a GPU-based RNG into the MC code, typically as a __device__ function.
- Path-generation and payoff-function kernels call the RNG as needed.

Random-Number Generation
Some Guiding Principles (cont'd)

Pregeneration vs. Inline Generation

	Pregeneration	Inline Generation
Pros	✓ Straightforward to implement ✓ Modular (easy to swap different RNGs, including 3rd-party and CPU libraries) ✓ Can optimize allocation of threads, thread blocks and shared memory to suit the RNG ✓ Allows use of complex RNGs and transforms ✓ Generate once, use many	✓ FAST! ✓ No (or little) global-memory use ✓ Flexible: generate as many or few RNs as needed
Cons	× Can be global-memory intensive (reads/writes and quantity) × Requires sufficient RNs to be generated up front	× Difficult to implement, particularly with complex RNGs × Cannot integrate 3rd-party or CPU RNGs × Can lead to heavy register swapping × Requires RNG threading and shared-memory use to be compatible with that of path generation and payoff functions × Leads to very involved code architecture

Random-Number Generation Transforms

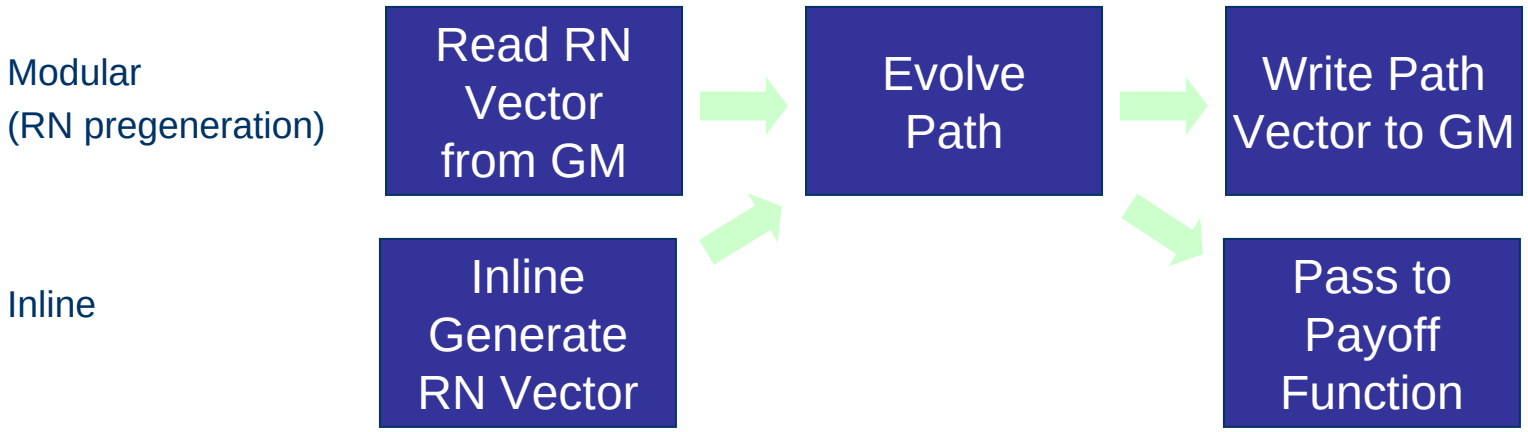
GPU-Efficient Uniform-to-Gaussian Transforms

- Avoid methods with divergent branching/looping (rules out e.g., sampling methods, ziggurat)
- GPUs have very efficient *sincos* function in hardware
- This makes Box-Muller a common choice for GPUs
- Inverse distribution is another common choice where inverse distribution can be computed efficiently
- *GPU Gems 3* describes Wallace Gaussian generator to directly generate Gaussian RNs on the GPU

Correlated Gaussian RN vectors

- Cholesky decomposition of covariance matrix is common: y'=Cz
- Matrix multiplication: single- or multi-threaded?
- Consider texture or constant memory for C

Path Generation
Some Guiding Principles



General Architecture Decisions

- One thread per path, or one thread per factor, or in between?
- Pregenerated or inline-generated RNs?
- Modular or inline paths?
  - Modular: write path information to global memory for use by separate payoff-function kernel
  - Inline: pass path information directly (typically in shared or global memory); payoff function typically in same kernel

Path Generation
Some Guiding Principles (cont'd)

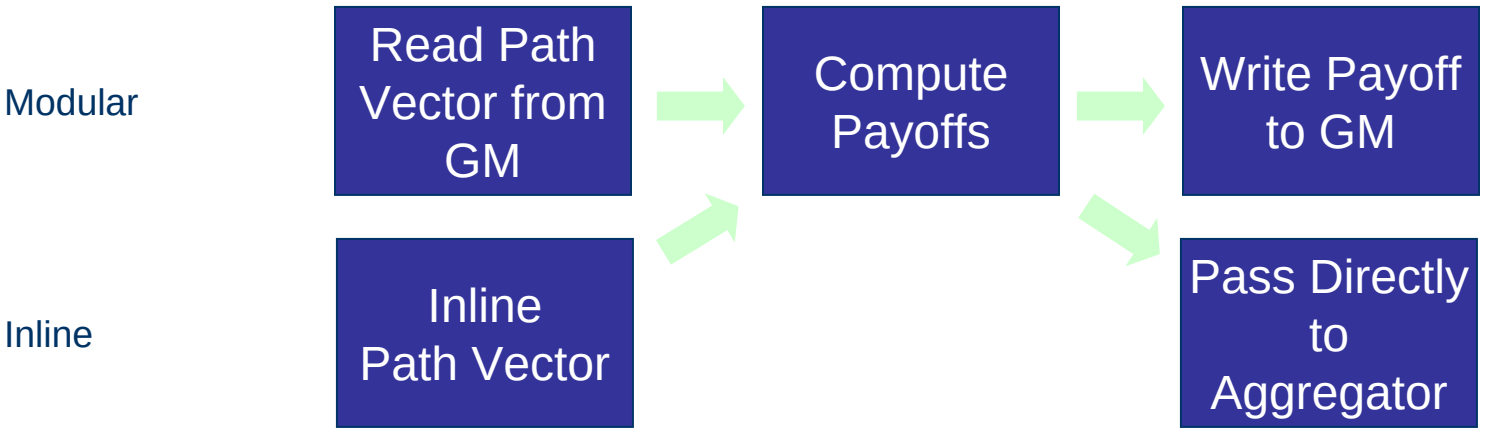
Threading

- More complex simulations (e.g., multiple factors) may benefit from groups of threads (even an entire thread block) per path
- Divergent branching hurts performance: it can be more efficient to pad groups of threads to a warp (and allow some threads to idle) to avoid divergent branching between paths

Memory

- Structure global-memory reads of RNs so they are coalesced:
  - pad global memory arrays to 64- or 128-byte boundaries
  - read groups of RNs into shared memory for later access
  - L1 cache can help, but better to be explicit
- Beware bank conflicts in shared-memory accesses
- Structure global-memory writes of path data so it is coalesced and can be efficiently read by the payoff functions
- Consider using textures or constant memory for model parameters

Payoff Functions
Some Guiding Principles



General Architecture

- Complex payoff functions can be difficult to optimize for the GPU due to lookbacks, lookforwards (e.g., Longstaff-Schwartz), cashflow schedules, waterfalls, call schedules, prepayment functions, etc.
- These often lead to:
  - divergent branching
  - uncoalesced global-memory reads
  - bank conflicts
- Optimization tradeoff is (as usual) performance vs. complexity

Payoff Functions
Some Guiding Principles (cont'd)

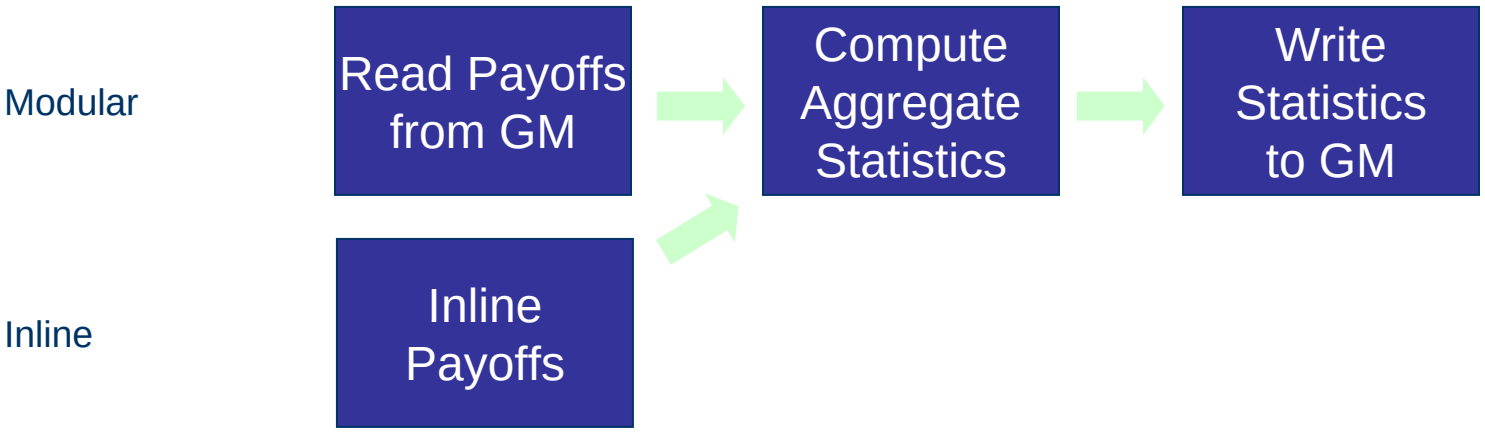
Threading

- More complex simulations (e.g., multiple factors) may benefit from groups of threads (even an entire thread block) per path
- Divergent branching hurts performance: it can be more efficient to pad groups of threads to a warp (and allow some threads to idle) to avoid divergent branching between paths

Memory

- Structure global-memory reads of path data so they are coalesced:
  - pad global memory arrays to 64- or 128-byte boundaries
  - read groups of paths into shared memory for later access
  - L1 cache can help, but better to be explicit
- Beware bank conflicts in shared-memory accesses
- Structure global-memory writes of payoff data so it is coalesced and can be efficiently read by the aggregators
- Consider using textures or constant memory for reference data

Aggregation



General Architecture

- For relatively large numbers of paths/payoffs, GPU can accelerate aggregation statistics
- Use parallel sum-reduction techniques to compute moments (e.g., *GPU Gems 3*, Ch. 39; CUDA SDK *reduction, MonteCarloCURAND*)
- Use parallel sort to compute quantiles and VaR (e.g., CUDA SDK *radixSort*)

This presentation has been prepared for the exclusive use of the direct recipient. No part of this presentation may be copied or redistributed without the express written consent of the author. Opinions and estimates constitute the author's judgment as of the date of this material and are subject to change without notice. Information has been obtained from sources believed to be reliable, but the author does not warrant its completeness or accuracy. Past performance is not indicative of future results. Securities, financial instruments or strategies mentioned herein may not be suitable for all investors. The recipient of this report must make its own independent decisions regarding any strategies, securities or financial instruments discussed. This material is not intended as an offer or solicitation for the purchase or sale of any financial instrument.

Additional information is available upon request.