Hanweck Associates, LLC

GPU-Accelerated Quant Finance: The Way Forward

NVIDIA GTC Express Webinar

February 29, 2012

Gerald A. Hanweck, Jr., PhD

CEO, Hanweck Associates, LLC

Hanweck Associates, LLC
30 Broad St., 42nd Floor
New York, NY 10004
www.hanweckassoc.com
Tel: +1 646-414-7274

Agenda

- Why GPUs in Quant Finance?
- Overview of GPU and CUDA Technology
- GPU Quant Finance Case Studies
  - 1: Pricing a basket barrier option
  - 2: Large-scale Monte Carlo value-at-risk (VaR)
  - 3: Random number generation
  - 4: Large-scale parametric VaR
  - 5: Stochastic volatility modeling
- Concluding Remarks

Q: What Is the Biggest Problem Facing the Capital Markets Today?

A: Intraday and Real-Time Risk Management

- Increasingly complex, global structured products
- Higher correlation risk and systemic risk
- Greater regulatory requirements
- Massive grid-computing infrastructure costs
- Exploding market-data message rates

Hanweck Associates: Overview

Recognized Leader in High-Performance Financial Risk-Management Systems

- Pioneered GPU-based computation in the financial industry.
- Extensive quant experience across equity, fixed-income, credit and FX asset classes, and their derivatives.

Hanweck Associates' Volera™ GPU-Accelerated Product Line

VOLERAFEED™

Real-time, low-latency datafeed of options implied volatilities and greeks covering global markets, powered by the Volera™ GPU-based engine.

VOLERARISK™

High-performance, real-time, large-portfolio risk and real-time pre/post-trade portfolio margining, powered by the Volera™ GPU-based engine.

Interactive Data

Options Volatility Service™

Historical, end-of-day options analytics database covering more than 6,000 U.S. companies over the past 12 years, distributed through Interactive Data and Thomson Reuters.

Premium Hosted Database™

Hosted historical and real-time "tick-level" database service of equity and options prices and analytics, with 300+ TB of data stored in an enterprise-scale cloud. "Data-and-Analytics-as-a-Service" paradigm.

ISE Implied Volatilities & Greeks Feed™

Real-time, low-latency datafeed of options implied volatilities and greeks covering global options markets, powered by the Volera™ GPU-based engine.

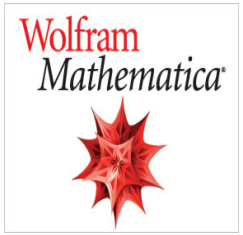
GPU Acceleration in Quant Finance

Random number generation	Trees and lattices	Equity derivatives
Path generation	Matrix algebra	Interest-rate derivatives
Payoff function acceleration	Numerical integration	Credit models
Statistical aggregation	Fourier transforms	Exotics and hybrids

Investment Banks	Asset Managers	Pension Plans
Hedge Funds	Insurance Companies	Mortgage Servicers

- ***10x faster*** "dollar-for-dollar" than conventional CPU computing.
- ***10x faster*** means:
  - overnight → over lunch
  - over lunch → get a cup of coffee
  - get a cup of coffee → don't blink!
- Better risk management.
- Reduce total cost of ownership and infrastructure cost explosion.

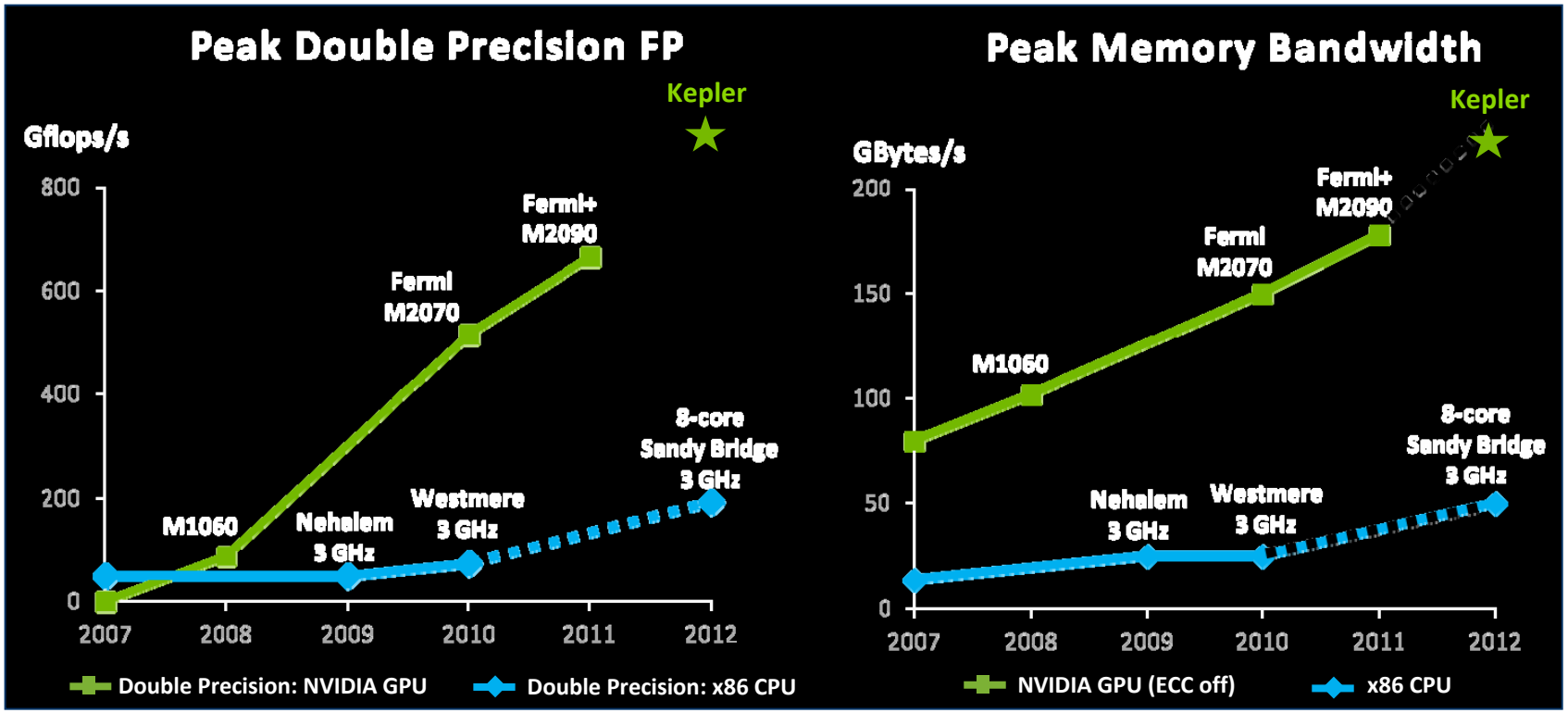
GPUs in Quant Finance



Source: NVIDIA

NVIDIA GPU Performance

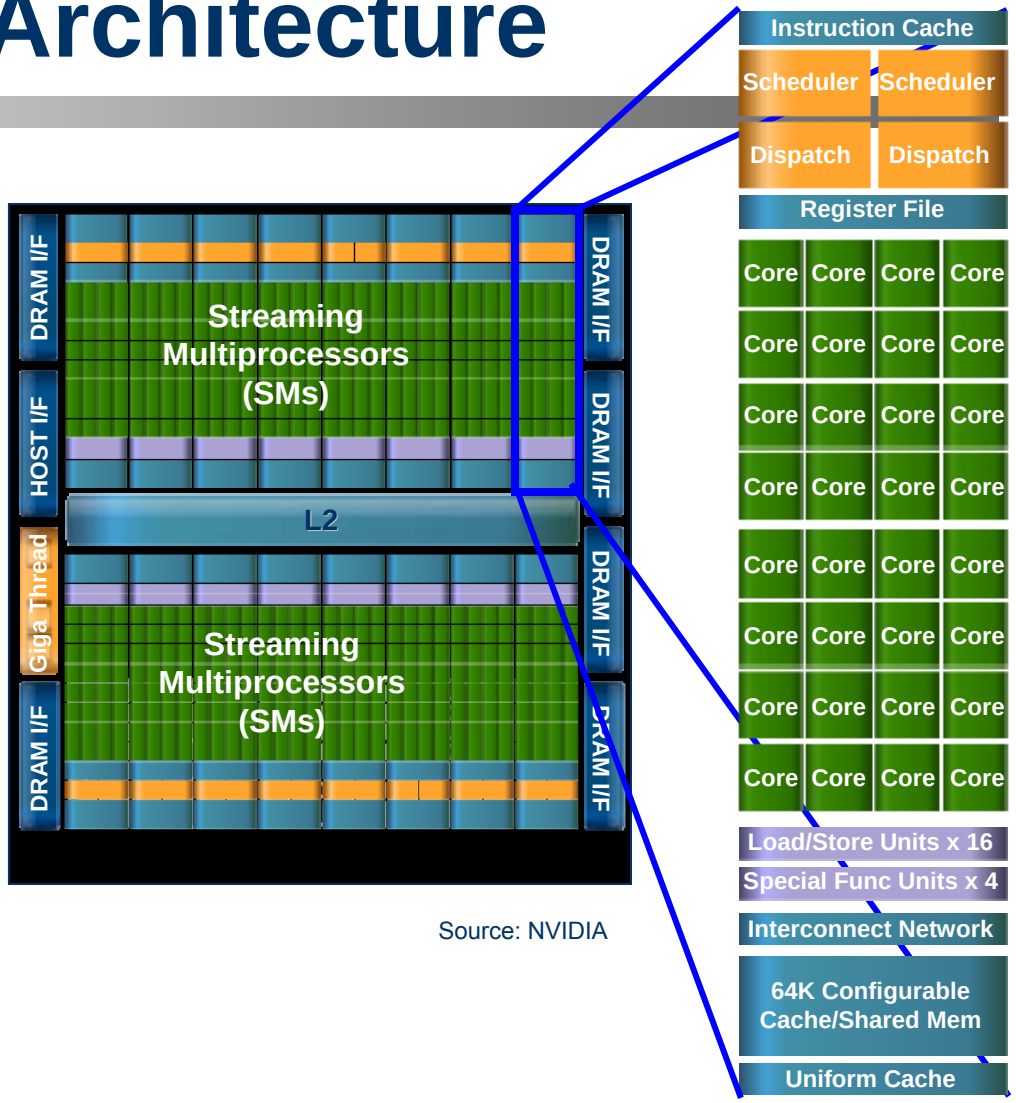
Increasing Double-Precision Performance and Memory Bandwidth



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

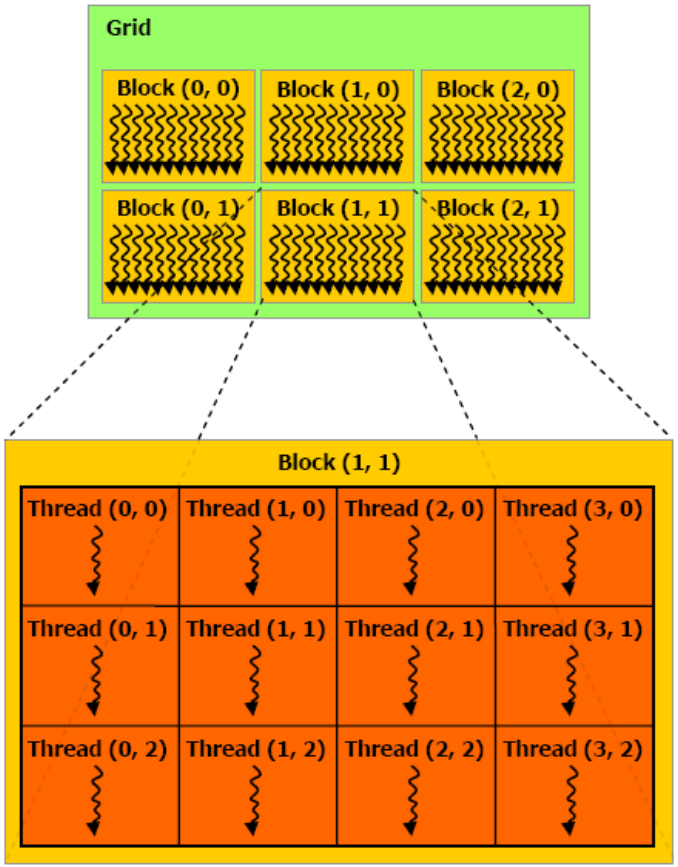
- CPU code passes control to GPU functions called *kernels*
- Data is transferred between CPU to GPU memory via DMA copies

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

Case Study #1: Basket Barrier-Option (Monte Carlo)

Valuing a basket barrier-option

Monte Carlo simulation of a multi-factor, local-volatility model for pricing lookback structures:

- 4 underlying assets
- 100,000 MC paths
- 750 steps per path

	CPU[1] Time (sec)	GPU[2] Time (sec)
Stage 1: RNG	20.2	0.139
Stage 2: Path Gen	23.5	0.181
Stage 3: Payoffs	8.5	0.223
Stage 4: Stats	9.3	0.061
Total	61.9	0.604
Performance Gain		**102x faster**

Realistic "dollar-for-dollar"[3] performance gain: **12x faster**

1. One core of Intel Xeon L5640 @ 2.26GHz
2. One NVIDIA Fermi C2070 GPU
3. Performance adjusted for: core/GPU density, amortized hardware costs, power/cooling costs, etc.

Case Study #2: Large-Scale Monte Carlo VaR

Prototype developed for a European derivatives exchange to demonstrate feasibility of real-time risk-based portfolio margining on large portfolios:

- 10,000 Monte Carlo paths generated from factor shocks (2,500 factors) on 3,500 underlying stocks and indices.
- 350,000 distinct options representing the listed universe.
- Hundreds of large portfolios.
- Full binomial-tree revaluation of each option in each path.
- Calculation of VaR and expected shortfall under correlation scenarios (historical, perfect and no correlation).
- Hardware: 24 NVIDIA C2050 GPUs w/ 8-core Xeon host servers.

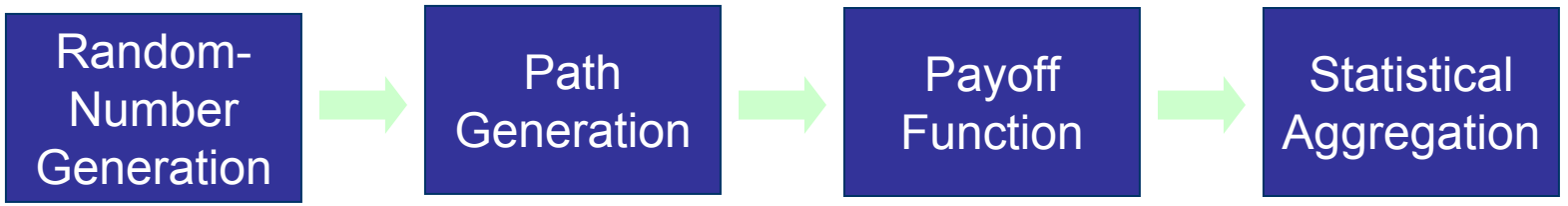
Large-Scale, Full-Revaluation Monte Carlo VaR:

< 1 minute

(hundreds of times faster than a single CPU core)

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!!!

Case Study #3: Random Number Generation

Implementation of a GPU-parallel Monte Carlo simulation and random-number generator for a major investment bank:

- Monte Carlo simulation of a multi-factor, local-volatility model for pricing lookback structures
- Implementation of an efficient GPU-parallel random-number generator*
- Hardware: 1 NVIDIA C2070 GPU w/ 8-core Xeon host server

GPU-Parallel Monte Carlo:
2.5 billion normal random numbers per second
(200x faster than a single CPU core)

* L'Ecuyer, Pierre; Richard Simard; E. Jack Chen and W. David Kelton, "An Object-Oriented Random-Number Package with Many Long Streams and Substreams," *Working Paper, December 2000*.

Random-Number Generation

Large base of existing GPU code and resources:

NVIDIA's cuRAND RNG library

http://developer.nvidia.com/curand

- L'Ecuyer (MRG32k3a), MTGP Mersenne Twister, XORWOW PRNG and Sobol QRNG

NVIDIA's CUDA SDK sample code:

- Niederreiter, Sobol QRNGs, Mersenne Twister
- 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*)

Random-Number Generation Some Guiding Principles

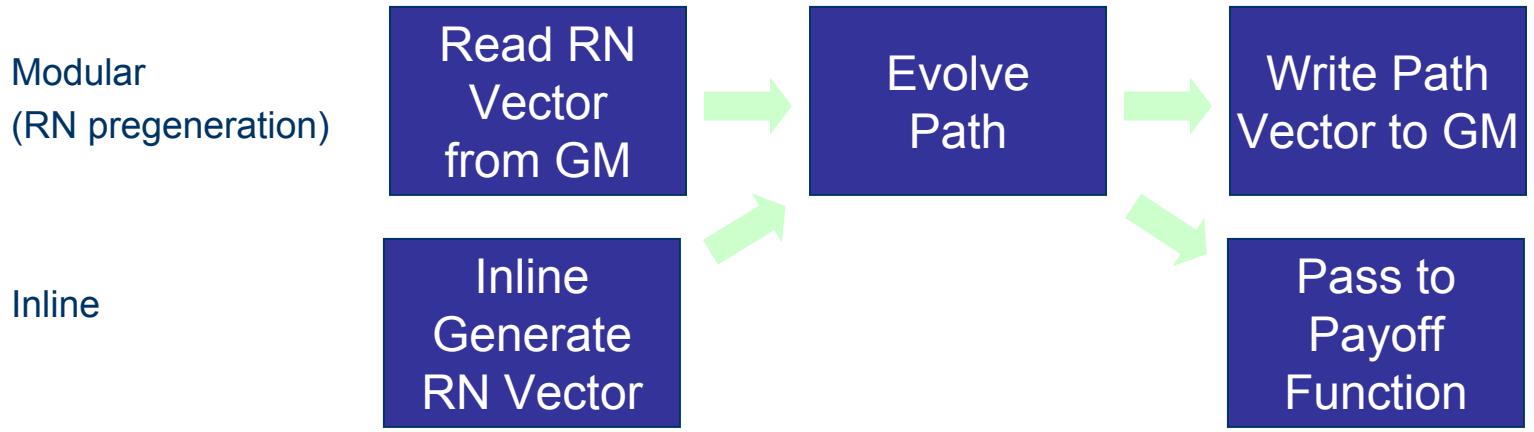
Pregeneration

- 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.

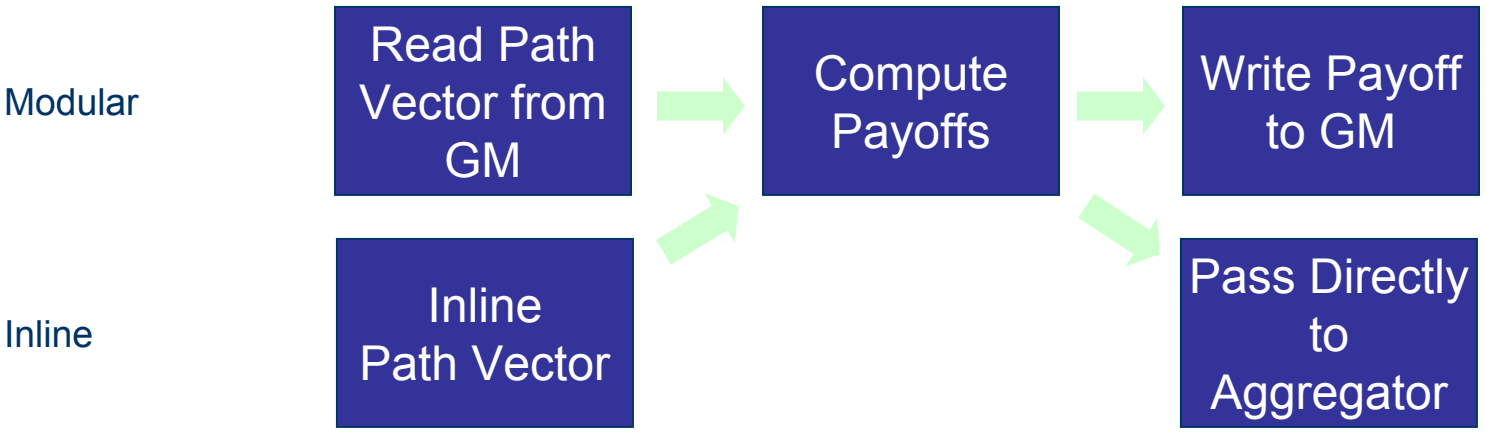
Path Generation
Some Guiding Principles



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

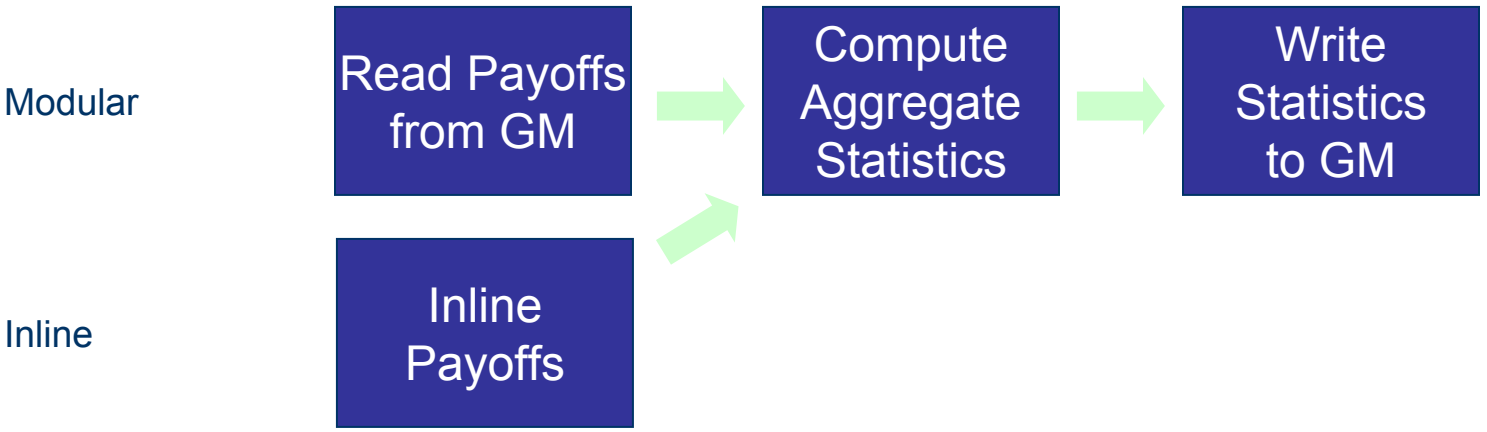
Payoff Functions
Some Guiding Principles



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

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*)

Case Study #4: Large-Scale Parametric VaR

System developed for a large investment bank to evaluate parametric factor VaR on millions of private client portfolios, with aggregation across accounts, advisors, offices and regions:

- 1.25 million portfolios
- 2,000 factors covering 400,000 global assets
- Hardware: 12 NVIDIA C2050 GPUs w/ 8-core Xeon host server

Large-Scale Parametric Factor VaR:
2 minutes
(hundreds of times faster than a single CPU core)

Case Study #5: Stochastic Volatility Modeling

Price options under a stochastic volatility model (Heston, 1993):

- European-style call and put options
- Solution involves numerical integration of complex-valued integrands
- Simpson's rule with dynamic integration ranges
- Hardware: NVIDIA C2070 GPU vs. Intel Xeon E5640 (2.67 GHz)

Option Pricing under Stochastic Volatility:

2,000 option pricings per second

(70x faster than a single CPU core)

The Heston Model

Let *S* represent the underlying asset price, and *v* represent its variance. Heston* represents their evolution through time as:

$$dS(t) = \mu S\, dt + \sqrt{v(t)}\, S\, dz_1(t)$$

$$dv(t) = \kappa[\theta - v(t)]dt + \sigma\sqrt{v(t)}\, dz_2(t)$$

where z_1 and z_2 are Weiner processes correlated as ρ.

Convenient properties of the Heston model:

- stochastic volatility $\Rightarrow$ volatility smiles
- correlation between underlying and volatility $\Rightarrow$ volatility skews
- mean-reverting volatility
- extends Black-Scholes
- extensible to stochastic jumps (e.g., Bates, 1996)
- **quasi closed-form solution for European call and put options!**

* Heston, Steven L., "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options," *The Review of Financial Studies,* 6(2), 1993, pp. 327-343.

The Heston Model: Quasi Closed-Form Solution

Heston formula for pricing a European call option on *S* with strike *K*, time to expiration *T*, constant interest rate *r*, and market price of volatility risk λ:

$$C(S, v, t) = SP_1 - KP(t, T)P_2$$

$$P_j(x, v, T; \ln[K]) = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}\left[\frac{e^{-i\phi \ln[K]} f_j(x, v, T; \phi)}{i\phi}\right] d\phi$$

$$f_j(x, v, t; \phi) = e^{C(T-t;\phi) + D(T-t;\phi)v + i\phi x}$$

$$C(\tau; \phi) = r\phi i\tau + \frac{a}{\sigma^2}\left\{(b_j - \rho\sigma\phi i + d)\tau - 2\ln\left[\frac{1 - ge^{d\tau}}{1 - g}\right]\right\}$$

$$D(\tau; \phi) = \frac{b_j - \rho\sigma\phi i + d}{\sigma^2}\left[\frac{1 - e^{d\tau}}{1 - ge^{d\tau}}\right]$$

$$g = \frac{b_j - \rho\sigma\phi i + d}{b_j - \rho\sigma\phi i - d} \qquad d = \sqrt{(\rho\sigma\phi i - b_j)^2 - \sigma^2(2u_j\phi i - \phi^2)}$$

$$u_1 = \tfrac{1}{2} \quad u_2 = -\tfrac{1}{2} \quad a = \kappa\theta \quad b_1 = \kappa + \lambda - \rho\sigma \quad b_2 = \kappa + \lambda \quad P(t, t + \tau) = e^{-r\tau}$$

Quasi-Closed-Form Solution (cont'd)

$$P_j(x,\ v,\ T;\ \ln[K]) = \frac{1}{2} + \frac{1}{\pi}\int_0^{\infty} \mathrm{Re}\left[\frac{e^{-i\phi\ \ln[K]} f_j(x,\ v,\ T;\ \phi)}{i\phi}\right] d\phi$$

Using Heston to price a European call or put option involves numerically integrating two complicated complex-valued functions.

Fitting the model to market data can require hundreds of thousands of option pricing evaluations.

Real-time applications require fast calculations.

Unfortunately, numerical integration is computationally intensive... but it is also *embarrassingly parallel.*

Enter the GPU!

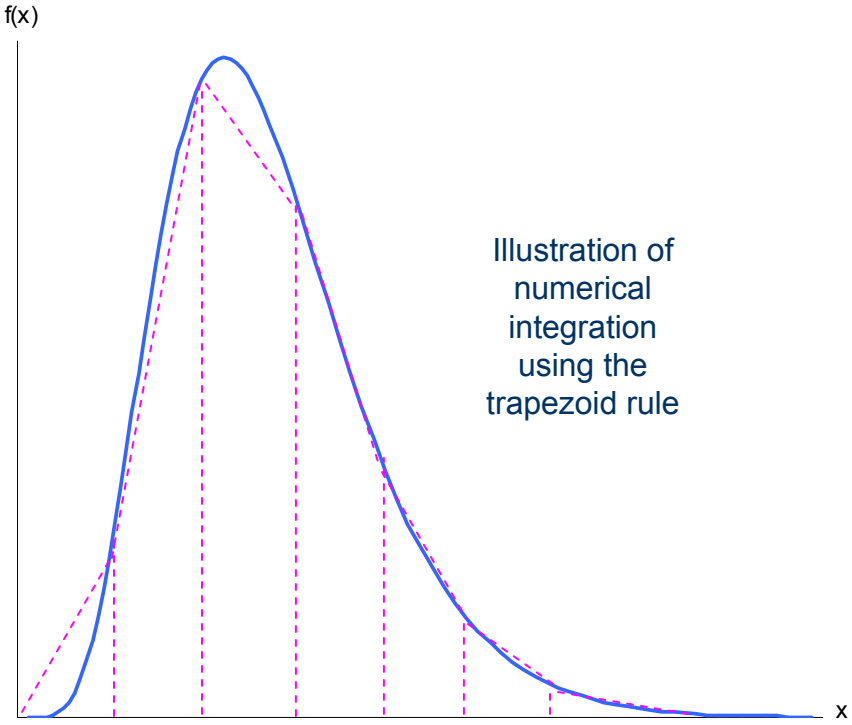
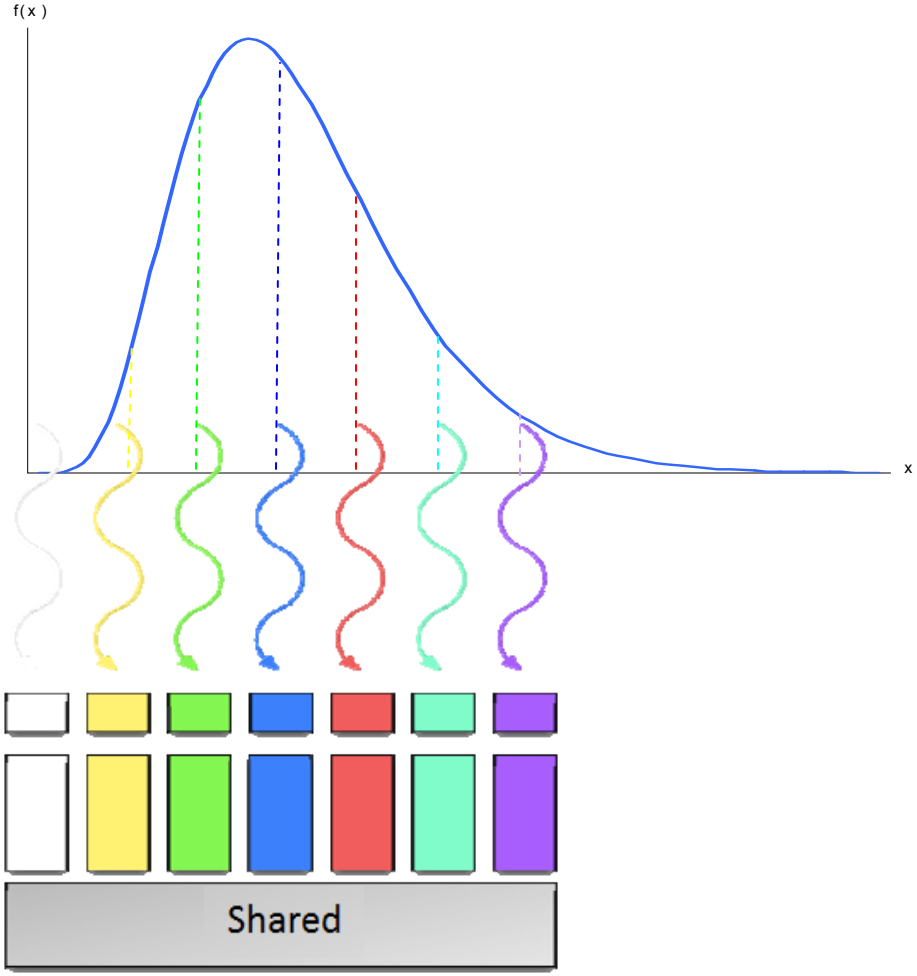


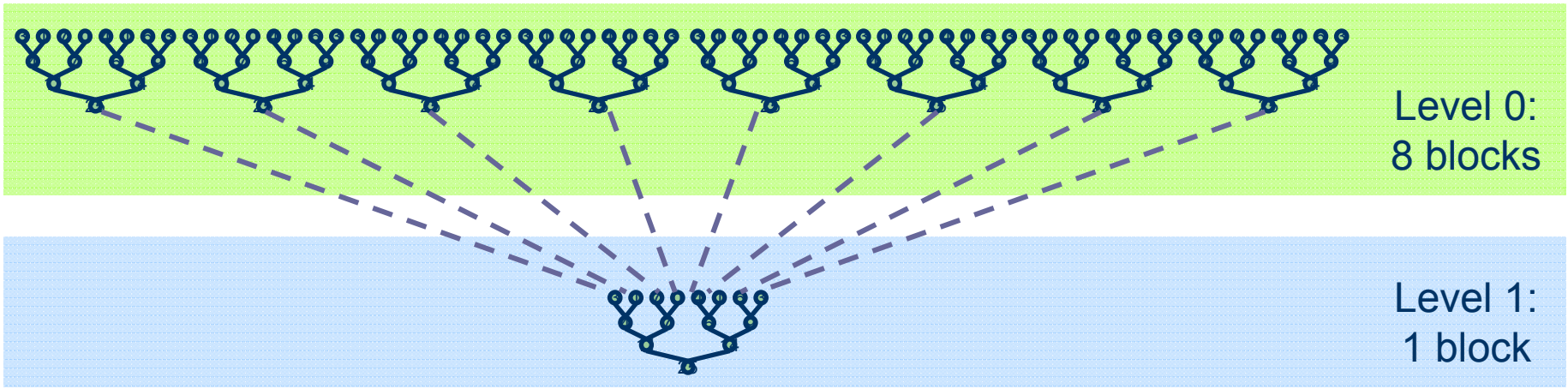
Illustration of numerical integration using the trapezoid rule

Numerical Integration

- Each thread evaluates one piece of the integral and saves the result to shared memory
- Each thread block then performs a sum reduction on all points evaluated within the thread block
- One value from each thread block is saved to global memory and copied back to host memory
- These results are then summed on the CPU to compute the value of the integral. If many thread blocks are needed in the first kernel call, then another sum reduction kernel can be executed to compute this sum



Sum Reduction



- A sum reduction technique is used to sum the value of the Heston integral
- As shown above, multiple kernel call can be used to perform the sum if many points need to be evaluated

Concluding Remarks

Real-time and intra-day risk management is a major problem facing the financial industry today... but it is pushing conventional computing to its limits.

GPUs *are* the way forward. Major financial institutions are using them for quant finance.

Performance gains of *more than 10x – dollar for dollar* – are achievable in practice in many common use cases, which is generally sufficient to offset the costs of new development.

GPU programming in general – and CUDA in particular – push developers to parallelize their code.

Parallelizing quant finance is critical if quant finance software is to take advantage of the advances in many-core hardware.