Financial Risk Modeling on Low-power Accelerators: Experimental Performance Evaluation of TK1 with FPGA

Rajesh Bordawekar and Daniel Beece

IBM T. J. Watson Research Center

Outline

- Motivation
- Monte Carlo Option Pricing
  - Path Generation
  - Accumulator Forward Option
- Parallelization on TK1
- Experimental Evaluation
- Conclusions and Future Work

Motivation

- Monte Carlo simulation extensively used in financial modeling
  - Monte Carlo is a compute-bound problem
- FPGAs and GPUs are increasingly being used for accelerating financial kernels
  - Low power consumption of FPGA a key advantage over enterprise-class GPUs (e.g., a K40)
  - Lower price enables building price-competitive clusters
- Focus of this work:
  - Evaluate exploitation of TK1 for accelerating financial Monte Carlo (specifically pricing esoteric options)
  - Compare performance and power consumption

Pricing via Monte Carlo Simulation

- Used for pricing esoteric options
  - no analytic solution, typically 10% to 20% of pricing functions in a portfolio
- Low I/O- High Compute Workload: suitable for accelerators such as FPGA and GPUs
- Focus of this work: Accumulator Forward Options

Pricing Function: Accumulator Forward Option

- Option on a stock with defined “strike” and “barrier” prices
- At fixed intervals (e.g., each month)
  - seller is obliged to sell at the “strike” price
  - buyer is obliged to buy at the “strike” price
- No down side limit
  - buyer can loose a lot of money
- Limited up side
  - contract terminates if price exceeds the “barrier”
- Must use Monte Carlo approach for pricing
  - no analytic solution

Core Computation of the Accumulator Forward Options

- Stochastic paths (10^6) of stock prices for 365 days
  - Quasi-random number generation (Sobol)
  - Gaussian distribution (inverse normal)
  - Path generation (Black-Scholes)
- Compute cash flows (pricing function) for each path

Sobol Sequences

- Low-dispersion, quasi-random numbers
  - uniformly distributed on the interval (0, 1)
  - requires inverse-normal transformation
- Two parameters- number of samples and number of dimensions
  - 10^6 samples (paths) in 365 dimensions (days)
- Faster convergence compared to other techniques
- Excellent implementations available with very long periods
  - Joe & Kuo (Sequential), basis of CURAND Sobol QRNG
- Easy to generate
  - exploits bit-vector operations e.g., shift, xor, mask of constants.

Black-Scholes Stochastic Model

- The Black-Scholes model describes the evolution of stock's price through a stochastic differential equation (SDE) the expresses the percentage change as increments of a Brownian motion

$$\frac{dS(t)}{S(t)} = r \times dt + \sigma \times dW(t)$$

stock price at time "*t*"

drift (mean rate of return)

volatility of the price

Brownian Motion:
normally distributed random variable
(mean 0, variance "t")

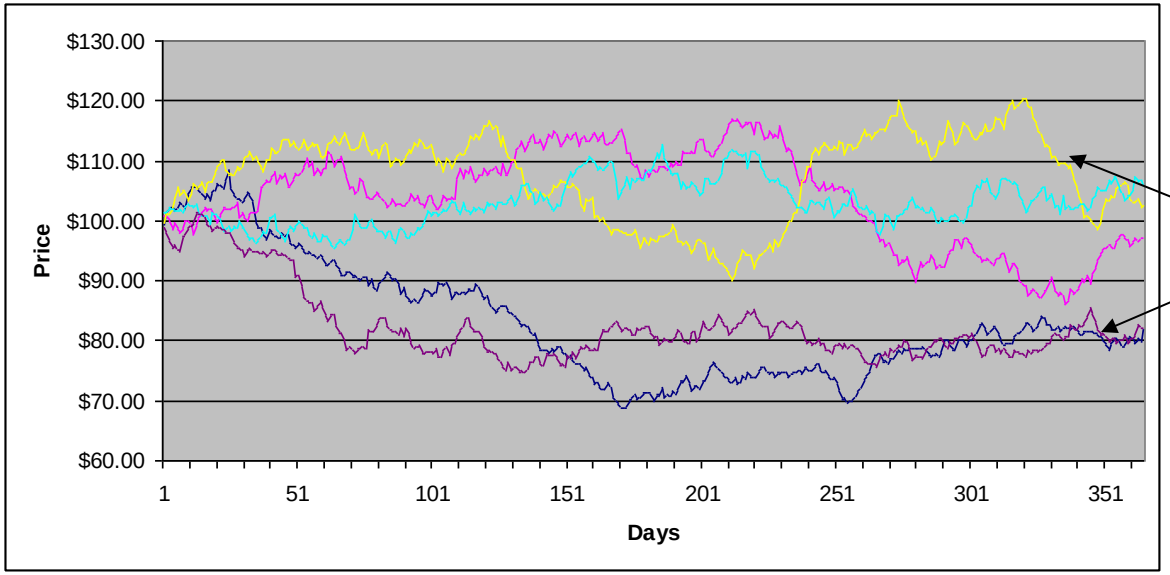
SDE Solution

$$S(t) = S(0) \times e^{\left(\left[r - \frac{1}{2}\sigma^2\right] \times t + \sigma \times \sqrt{t} \times Z\right)}$$

stock price at time "*t*"

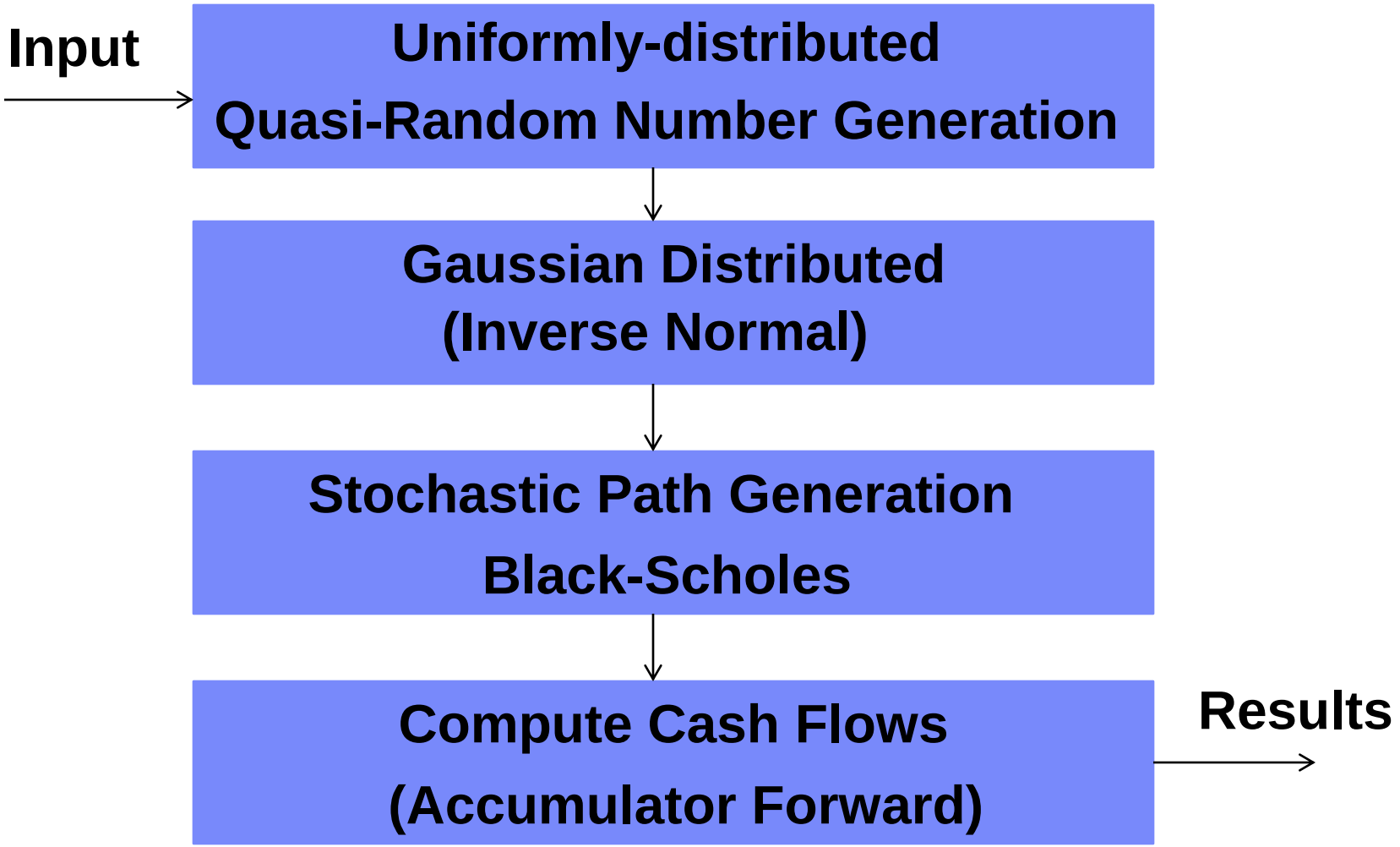
initial stock price

standard normal random variable (mean 0, variance 1)



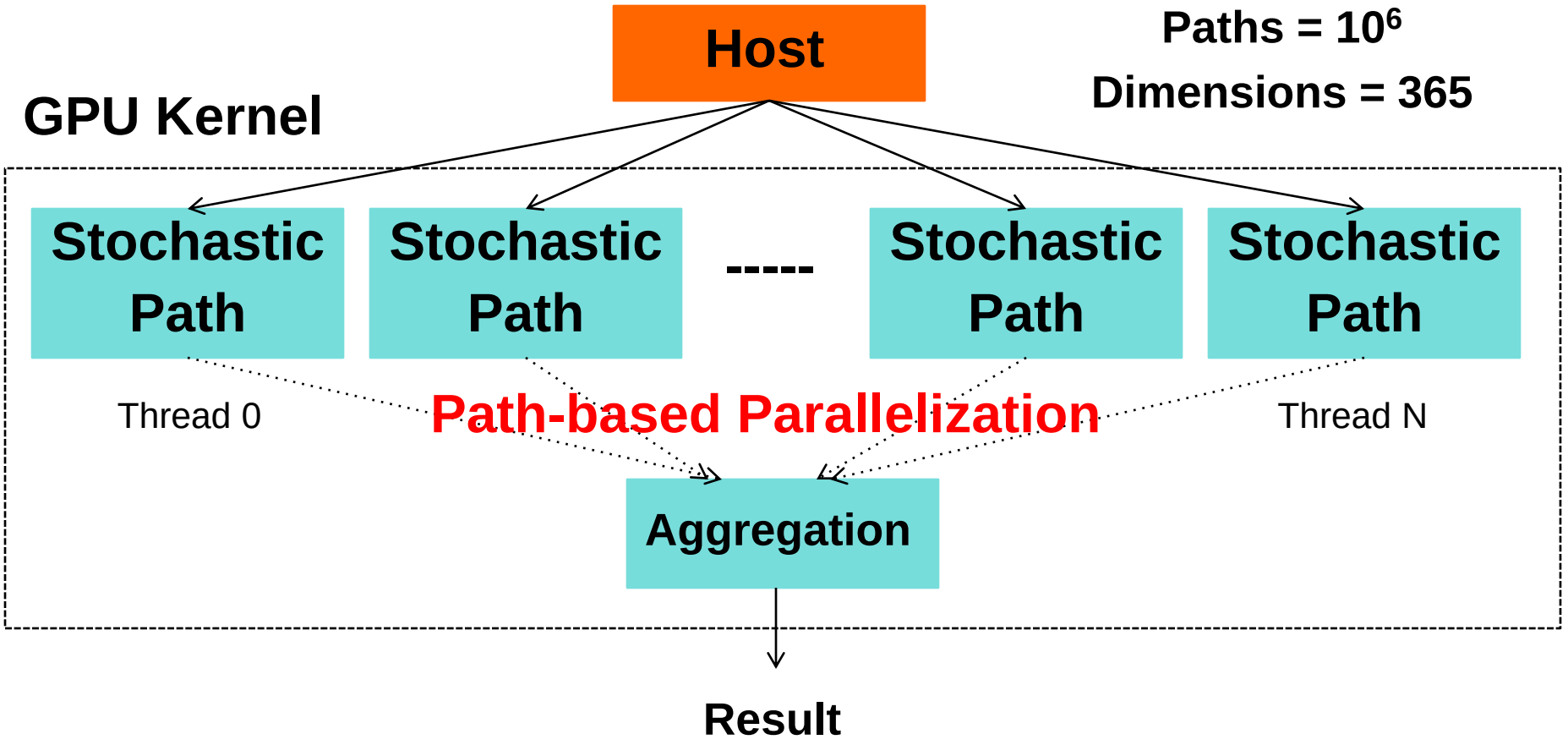
Paths

Execution Flow of the Monte-Carlo Computation



Parallelizing the Monte-Carlo Computation on GPU

Each thread executes one or more distinct paths.
Individual cash flows aggregated to compute final result



TK1 Implementation Details

- Issues impacting TK1 implementation
  - Weak ARM host: need to do everything on the TK1
  - TK1 has low memory bandwidth (peak 9 GB/s)
    - Minimize device memory accesses
  - TK1 has few physical cores: limit on the threadblock count
- Core computations on the TK1 **(Single-precision calculations)**
  - Sobol QRNG generation
    - Using CURAND Sobol generator versus native implementation
  - Inverse-normal calculations
  - Sum reduction to calculate final result
    - Uses warp functions to reduce usage of atomicAdd()

Implementation of Sobol Generator

- Sobol generators follow a simple recurrence
  - $x(n+1) = x(n) \oplus v(c)$ [Bratley and Fox, Algorithm 659]

    where $v(c)$ is called the direction number
  - x(n) computed using Gray code representation of "n"
    - Gray code(n) = ….. $g_3 g_2 g_1$.
    - Gray code(n) and Gray code (n+1) differ in one bit
    - x(n) = $g_1 v_1 \oplus g_2 v_2 \oplus$ ..
- For generating M samples in N directions, it requires N * 32 direction numbers (32 integers per dimension)
- Calculations across dimension completely independent
- Within a dimension, sample "i" can be calculated directly by solving the recurrence

Parallelizing Sobol Generator on GPU

- Sobol parallelization strategy depends on how the overall computation is parallelized
- Current strategy uses path-based parallelization
  - Each thread executes 365 iterations, each for a dimension
  - At every iteration “j” , thread “i” calculates a unique sample of index *map(i)* in dimension “j”
    - At every iteration “j” each thread operates on the 32 direction numbers for the direction “j”
    - Total data fetched from device memory = 32 * 365 * #thread-block
- Current CURAND interface can not support this execution pattern
  - Reading pre-computed 365×10^6 random numbers from TK1’s device memory extremely inefficient

Per-thread execution of Sobol generator

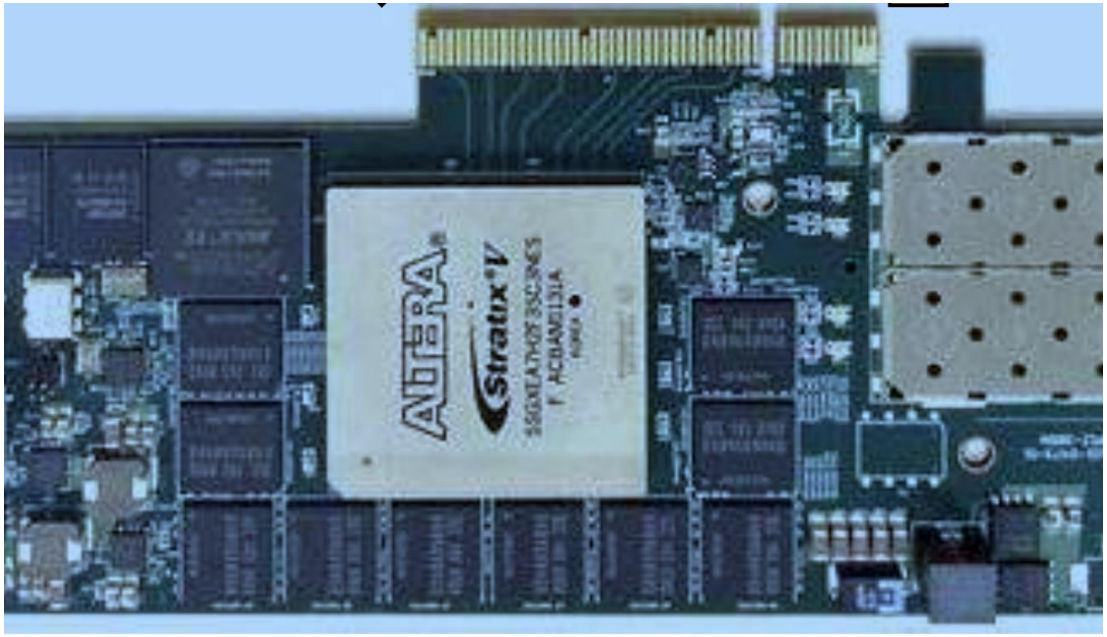
```
int stride= iterations; /* Stride = #Iterations */
int loops = __ffs(stride);
/* gid is between 0 and #iterations */
unsigned int gid = blockID* threads_per_block + iam;
unsigned int directions[32];
unsigned int X=0, mask=0;
/* Fetch direction vectors for dimension "j" (day " j") */
unsigned g = gid ^ (gid >> 1);
/* We want X ^= g_k * v[k], where g_k is one or zero. */
for (unsigned int k=0; k < loops -1 ; k++){
     mask = -(g & 1);
     X ^= mask & directions[k];
     g = g >> 1;
}
sobolSample_i_j = (float) X * k_2powneg32;  /* i == gid */
```

Modified version of code used in the Sobol QRNG Sample

Uses Joe and Kuo's (ACM TOMS 2003) dimension numbers

Experiment Evaluation: FPGA Setup

Altera Stratix V connected to Power 8 host



Implements a 1024-dimension Sobol Generator

Result aggregation computed on the Power 8 host

Experimental Results: 10^6 Paths and 365 Days

- **TK1: 12.28 sec @ 3 Watts** (ARM Host)
  - **0.013 sec for 1K Paths**
- **FPGA: 0.2 sec @ 9 Watts** (Aggregation done on the P8 host)
  - TK1 without aggregation takes **12.17 sec**
- Other architectures:
  - **K40: 0.053 sec @ 68 Watts** (Needs CPU host)
  - **x86 (IB): 1 sec, 20 threads**
- Cost Analysis
  - A TK1 board at least 50x cheaper than enterprise class multi-core CPU+accelerator system
  - GPU has smaller NRE ($) than FPGA

Experimental Results: TK1 Performance Issues

- Three expensive components
  - Sobol Calculations:
    - xor, bit shifts
    - Coalesced accesses to fetch 32 direction numbers
  - Inverse-normal and Path calculations
    - Exp, log, FMA operations
  - Result aggregation uses atomicAdd()
- Number of thread blocks can affect the performance
  - Using 1024 blocks of 128 threads each
- Overall GPU performance affected by Sobol, Inverse-normal, and Path Calculations
  - cost of accessing direction vectors insignificant

GPU versus FPGA

- FPGA was faster than TK1
  somewhat slower than K40
- FPGA consumes more power than TK1
  less than K40
- GPU programming easier than FPGA
  more flexible and less NRE compared to FPGA
- Same code runs on TK1 and K40

Conclusions and Future Work

- Implemented Monte-Carlo Pricing model for Accumulator Forward Options on the TK1
- TK1 performance affected by the computational functions (sobol, inverse-normal, pricing)
  - Need to investigate performance optimization opportunities
- Low power GPUs could be very competitive if run on enterprise class host