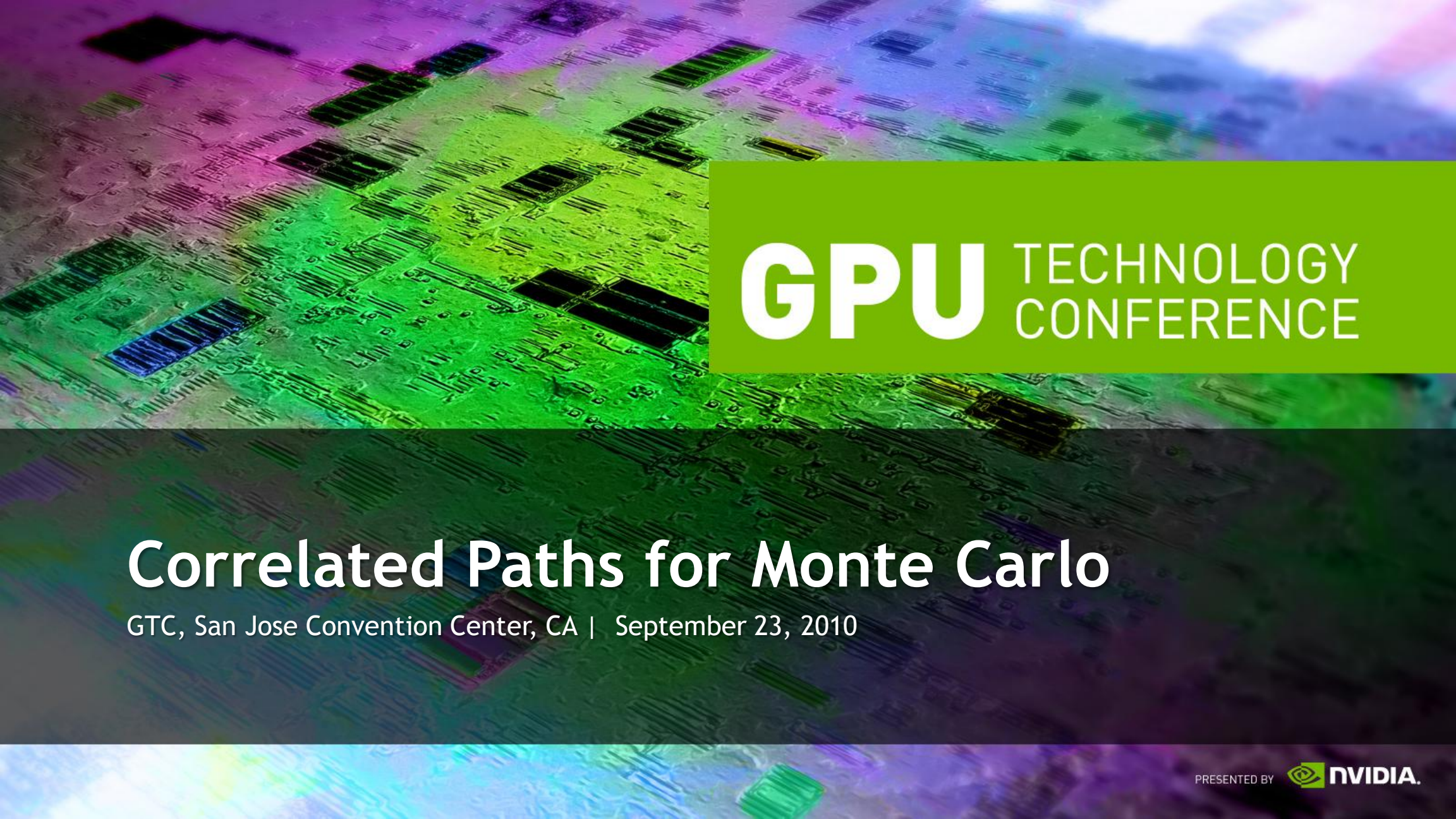


Correlated Paths for Monte Carlo

GTC, San Jose Convention Center, CA | September 23, 2010

PRESENTED BY NVIDIA.

Overview

- Background
- Application structure
- Correlated random number generation
- Putting it all together
- Summary

Objective and Assumptions

- Derivative with path-dependence, multiple assets
  - Complexity in time: How can we accelerate with the GPU?
  - Complexity in space: How should we structure on the GPU?
  - Correlation: How to implement on GPU?
  - Flexible: How to scale the application?
- Assumptions
  - Calibration is separate
  - Correlation is constant (e.g. derived from historical data)

Background

Option payoff

- Asian option*

$$payoff(t_N) = \max\left(\left(\frac{1}{N}\sum_{i=1}^{N} S(t_i)\right) - K, 0\right)$$

- Basket option+

$$payoff(t_N) = \max\left(\left(\sum_{j=1}^{M} w_j S_j(t_N)\right) - K, 0\right)$$

- Asian basket option

$$payoff(t_N) = \max\left(\left(\frac{1}{N}\sum_{i=1}^{N}\left(\sum_{j=1}^{M} w_j S_j(t_i)\right)\right) - K, 0\right)$$

* European-style call, fixed strike, discrete arithmetic average

\+ European-style call

Volatility

- Several ways to model volatility
  - Flat
  - Local volatility surface
  - Stochastic volatility

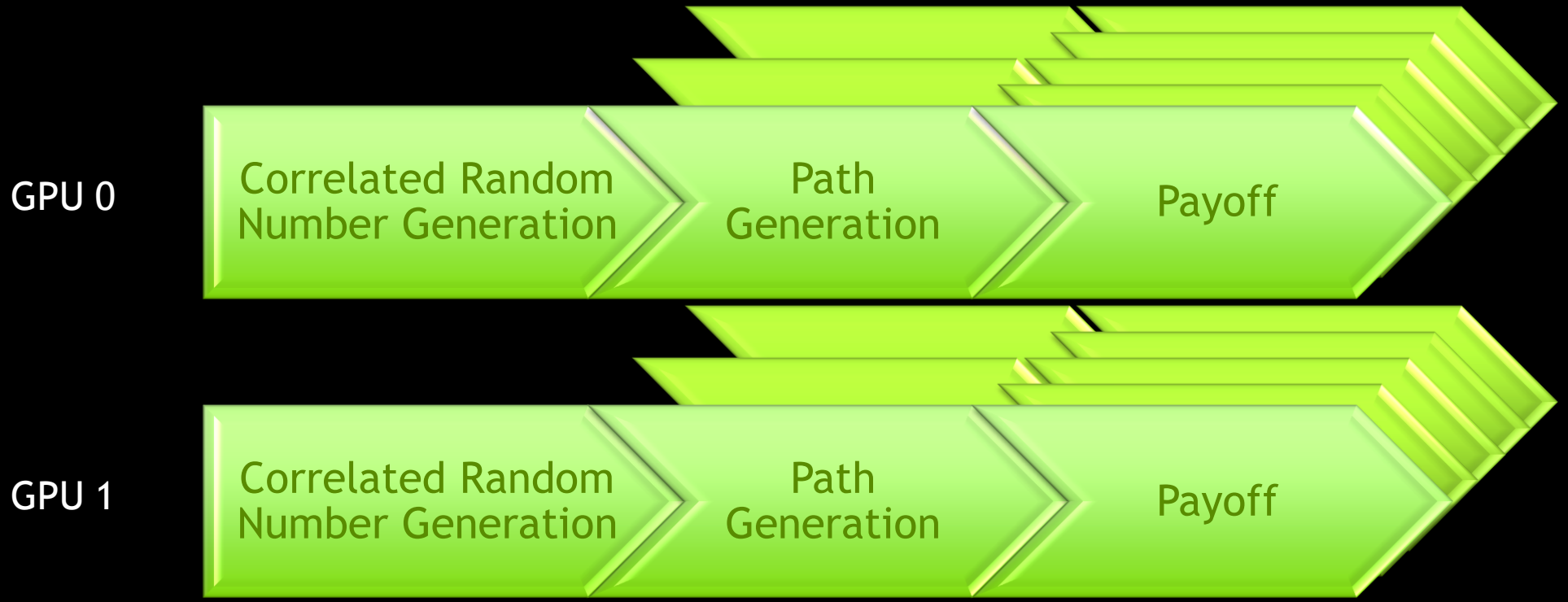
- For this exercise using Heston model for stochastic volatility

Application Structure

Preparation

- Batch work for efficiency
- Reuse paths
  - Related derivatives
- Reuse (correlated) random numbers
  - Derivatives based on same underlyings
  - Greeks

Decomposing the Application



Correlated Random Number Generation

Uncorrelated RNG - CURAND

- Marsaglia's XORWOW
  - xor-shift generator
  - Skip-ahead for non-overlapping streams
- Available in CUDA 3.2
  - PRNG (XORWOW) and QRNG (Sobol')
  - Uniform and Normal distributions
  - Batch and inline

Alternative RNGs

- NAG
  - PRNG: l'Ecuyer's MRG32k3a, MT19937
  - QRNG: Sobol'
  - www.nag.com/numeric/gpus
- Other
  - Mersenne Twister for Graphics Processors (MTGP)
  - www.math.sci.hiroshima-u.ac.jp/~m-mat

Underlying Asset Volatility

- Each asset is modeled by a traditional Heston model

$$\begin{pmatrix} dS_i(t) \\ dv_i(t) \end{pmatrix} = \begin{pmatrix} S_i(t)\left(r(t) - q_i(t)\right) \\ \kappa_i\left(\bar{v}_i - v_i(t)\right) \end{pmatrix} dt + \begin{pmatrix} S_i(t)\sqrt{v_i(t)} & 0 \\ 0 & \eta_i\sqrt{v_i(t)} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \rho_i & \sqrt{1-\rho_i^2} \end{pmatrix} \begin{pmatrix} dV_i(t) \\ d\tilde{V}_i(t) \end{pmatrix}$$

- Wiener processes $V_i(t)$ and $\tilde{V}_i(t)$ are independent, asset-volatility correlation modeled as ρ_i
- Input 1: single-asset Heston parameters for each UA

Cross-asset Correlation

- Correlation across assets is modeled in $V_i(t)$

$$\begin{pmatrix} dS_i(t) \\ dv_i(t) \end{pmatrix} = \begin{pmatrix} S_i(t)\big(r(t) - q_i(t)\big) \\ \kappa_i\big(\bar{v}_i - v_i(t)\big) \end{pmatrix} dt + \begin{pmatrix} S_i(t)\sqrt{v_i(t)} & 0 \\ 0 & \eta_i\sqrt{v_i(t)} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \rho_i & \sqrt{1-\rho_i^2} \end{pmatrix} \begin{pmatrix} dV_i(t) \\ d\tilde{V}_i(t) \end{pmatrix}$$

- $V_i(t)$ has correlation matrix $\Sigma = \rho_{i,j}$
  - Asset-volatility correlation between assets *i* and *j* is transferred from asset-asset ($\rho_{i,j}$) to asset-volatility by ρ_i and ρ_j
- Input 2: asset-asset correlations $\rho_{i,j}$

Correlating the Vectors

- Initialization:
  - Setup seeds
  - Decompose correlation matrix (Cholesky)
- For each time-step in each simulation:
  - Generate vector of i.i.d. random numbers, length N_{UA}
  - Apply asset-asset correlation ($\rho_{i,j}$) $\Rightarrow X(t)$
  - Generate vector of i.i.d. random numbers, length N_{UA}
  - Apply asset-volatility correlation for each UA (ρ_i) $\Rightarrow Y(t)$

Generating *X(t)* and *Y(t)* on the GPU

- Option A: Generate all UAs per thread
  i.e. *X(0..T)* and *Y(0..T)*
  - Apply all asset-asset and asset-vol correlations in same thread
  - $N_{threads} = N_{sims}$

- Option B: Generate one UA per thread
  i.e. $X_i(0..T)$ and $Y_i(0..T)$
  - Threads cooperate for asset-asset correlation
  - $N_{threads} = N_{sims} \times N_{UAs}$

Option A: All UAs per thread

Launch N_{sims} threads…

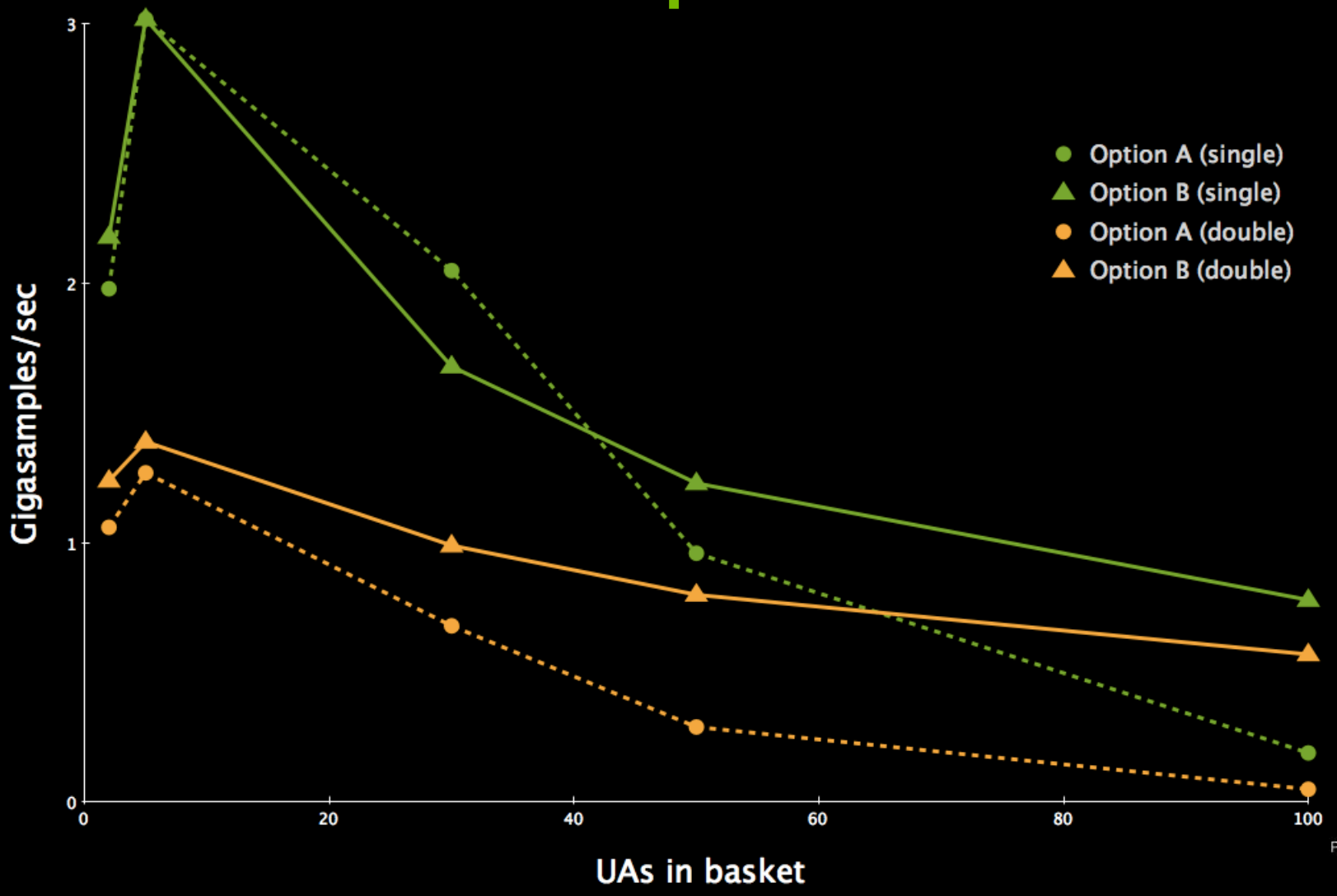
```
for t in 0..N_fixings
  for i in 0..N_UAs
      generate Normally distributed draw z_1
      for k in 0..i
          apply asset-asset correlation-factor ⇒ v_1
      store v_1 to shared memory
      generate Normally distributed draw z_2
      apply asset-vol correlation ⇒ v_2
      store v_1 and v_2 to global memory (i.e. X_i(t) and Y_i(t))
```


Option B: One UA per thread

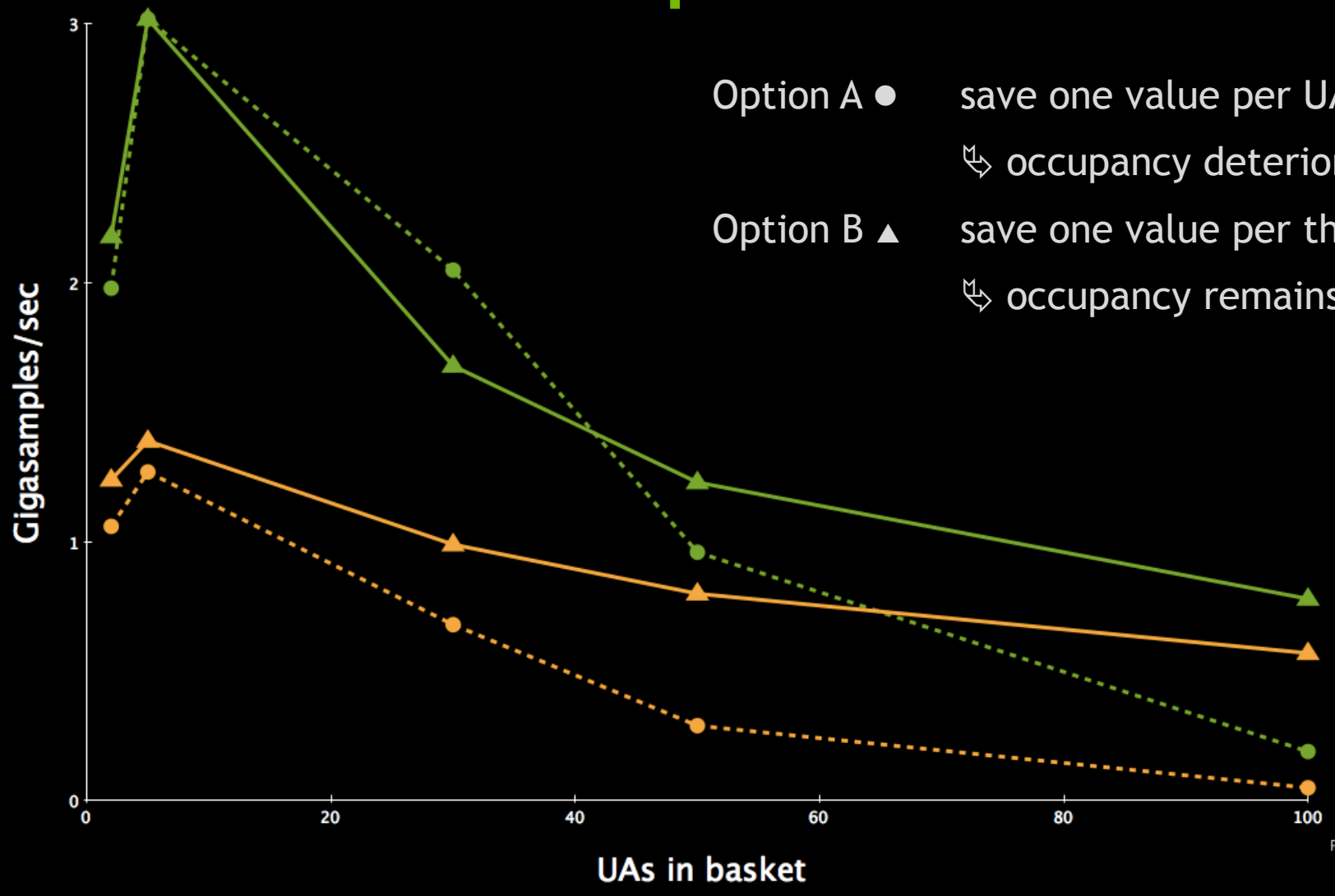
Launch $N_{sims} \times N_{UAs}$ threads…

```
for t in 0..N_fixings
  generate Normally distributed draw z_1
  store z_1 to shared memory, synchronize
  for k in 0..rank
      apply asset-asset correlation-factor ⇒ v_1
  generate Normally distributed draw z_2
  apply asset-vol correlation ⇒ v_2
  store v_1 and v_2 to global memory (i.e. X_rank(t) and Y_rank(t))
```

Performance Comparison on C2050



Performance Comparison on C2050



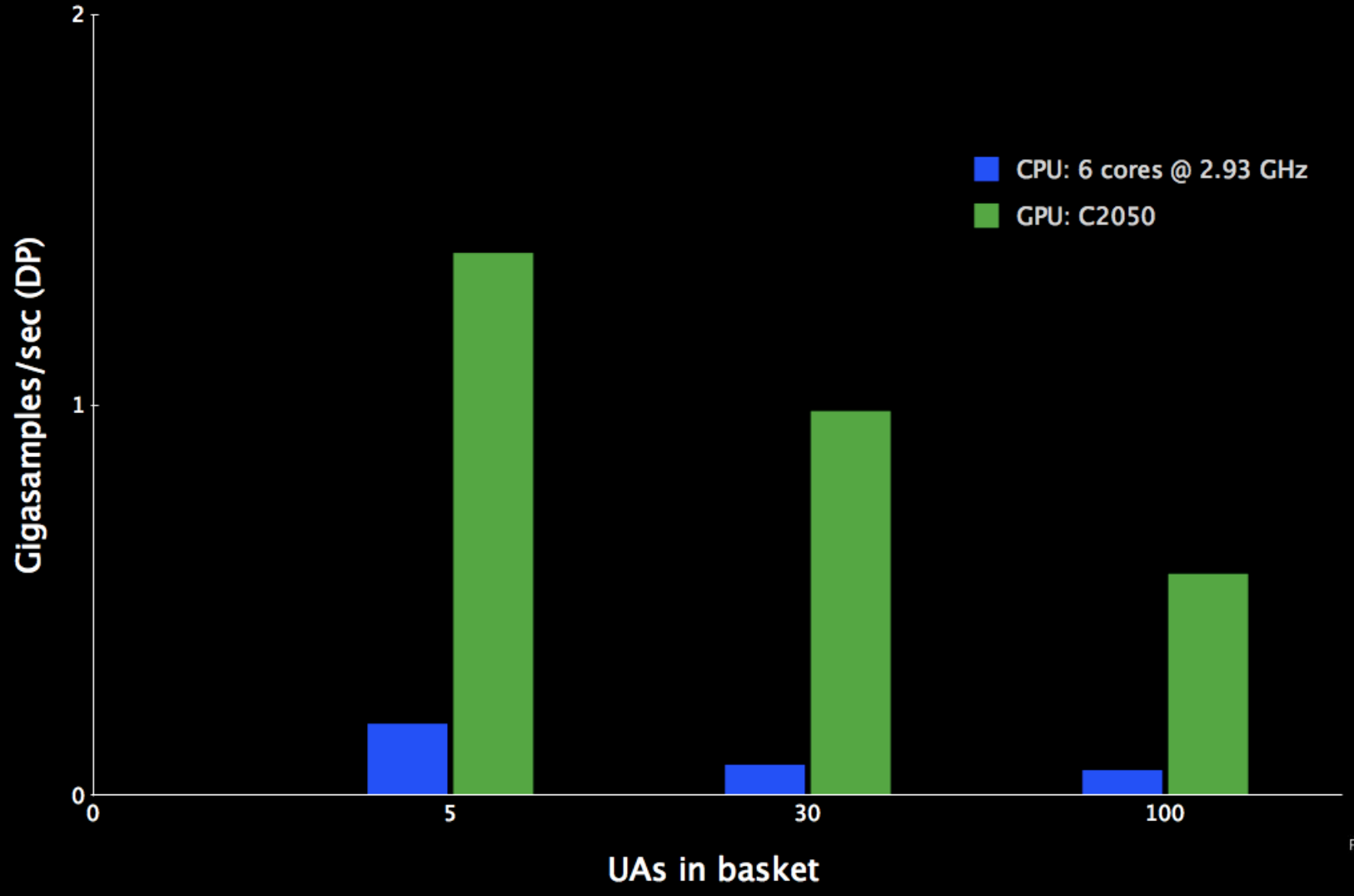
Option A ● save one value per UA per thread

↳ occupancy deteriorates

Option B ▲ save one value per thread

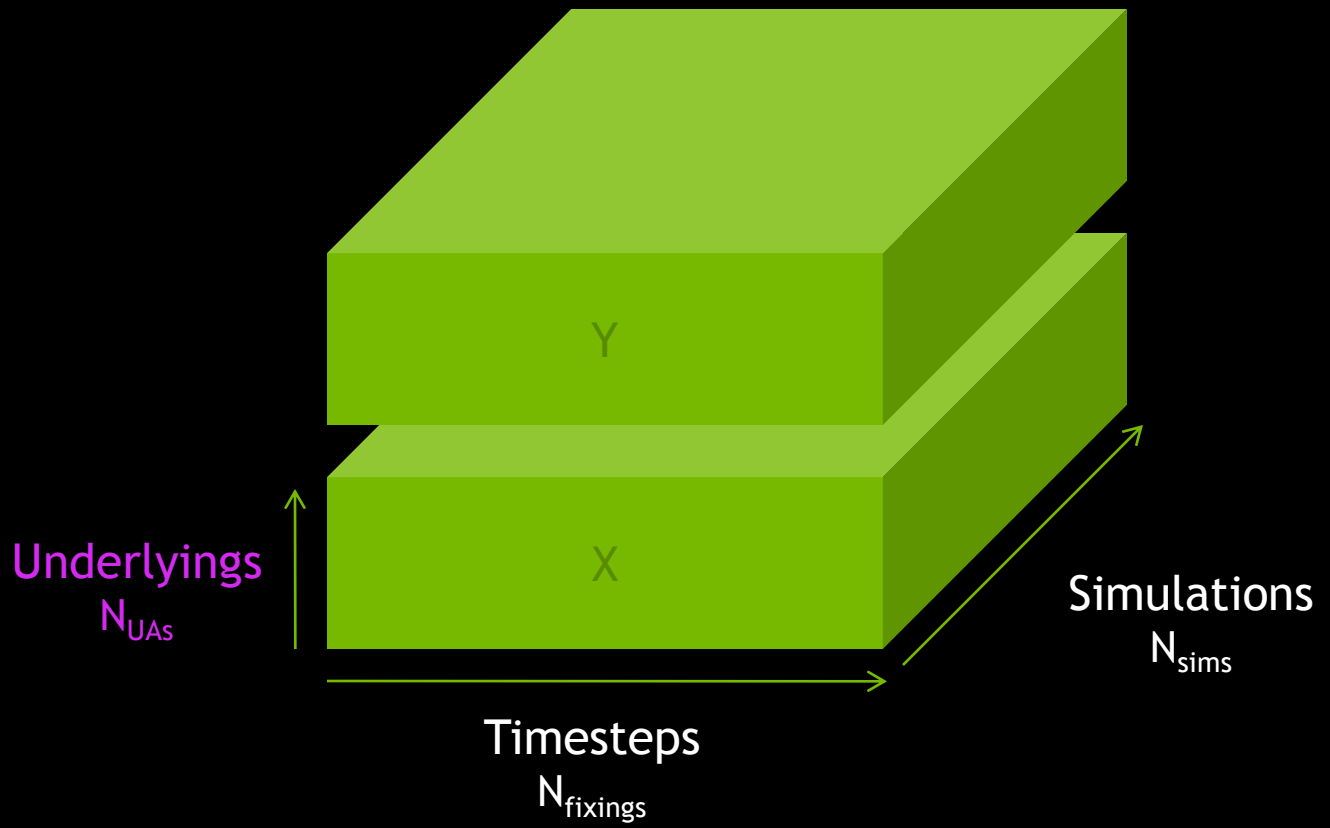
↳ occupancy remains good

Comparison with Host



PUTTING IT ALL TOGETHER

Correlated Random Numbers



UA index is fastest changing in memory

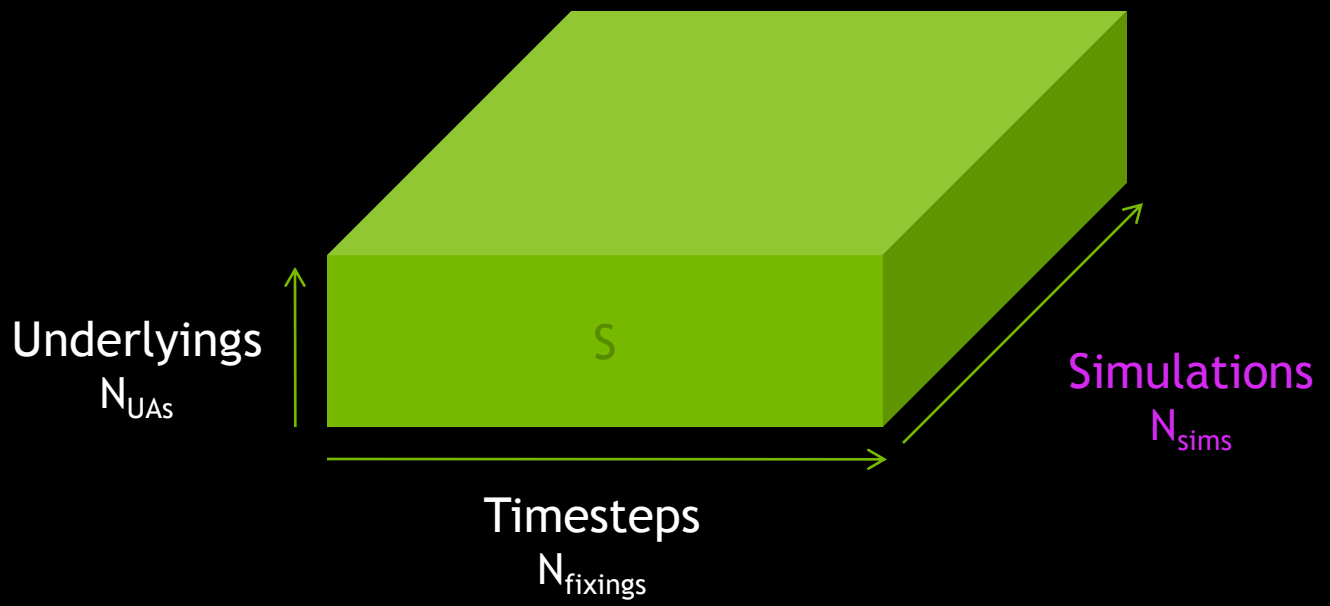
Generating the paths

- Each UA path is independent ⇒ $N_{sims} \times N_{UAs}$ paths

Launch $N_{sims} \times N_{UAs}$ threads…

```
for t in 0..N_fixings
   compute dS or d(log S)          Use X(0..T) from CRNG
   compute dv                      Use Y(0..T) from CRNG
   update and store S(t), update v
```

Output from Path Generation



Simulation index is fastest changing in memory

Computing the Expected Payoff

- Compute the payoff in each simulation, then reduce

Launch N_{sims} threads…

```
for t in 0..N_fixings
   for k in 0..N_UAs
      add w_k S_k(t) to basket sum
   add basket sum to path sum
compute mean
compute payoff
```

Flexible Payoffs with Thrust

- Thrust allows a more functional approach

```
thrust::transform(thrust::counting_iterator<unsigned int>(0),
                  thrust::counting_iterator<unsigned int>(0) + numSims,
                  payoffs.begin(),
                  compute_payoff(numFixings,
                                 numUAs,
                                 numSims,
                                 thrust::raw_pointer_cast(&paths[0]),
                                 params));

expayoff = thrust::reduce(payoffs.begin(), payoffs.end()) / numSims;
```

Flexible Payoff Functor

```
struct compute_payoff : public thrust::unary_function<unsigned int, Real>
{
    unsigned int numFixings, numUAs, numSims;
    Real strike, *paths;
    __host__ __device__ compute_payoff(…) : … {}
    __host__ __device__ Real operator()(unsigned int simIdx) {
            Real *base = paths + simIdx;
            Real sumBasketValue = static_cast<Real>(0);
            for (unsigned int t = 0 ; t < numFixings ; t++) {
                Real basketValue = static_cast<Real>(0);
                for (unsigned int ua = 0 ; ua < numUAs ; ua++) {
                    basketValue += base[ua * numSims];
                }
                sumBasketValue += basketValue / numUAs;
                base += numSims * numUAs;
            }
            return max((sumBasketValue / numFixings) - strike, 0);
        }
};
```

Flexible Payoff Functor

```
struct compute_payoff : public thrust::unary_function<unsigned int, Real>
{
    unsigned int numFixings, numUAs, numSims;
    Real strike, *paths;
    __host__ __device__ compute_payoff(…) : … {}
    __host__ __device__ Real operator()(unsigned int simIdx) {
        Real *base = paths + simIdx;
        Real sumBasketValue = static_cast<Real>(0);
        for (unsigned int t = 0 ; t < numFixings ; t++) {
            Real basketValue = static_cast<Real>(0);
            for (unsigned int ua = 0 ; ua < numUAs ; ua++) {
                basketValue += base[ua * numSims];
            }
            sumBasketValue += basketValue / numUAs;
            base += numSims * numUAs;
        }
        return max((sumBasketValue / numFixings) - strike, 0);
    }
};
```

Determine simulation index

Compute basket value at each fixing

Compute payoff

Summary

Maximizing Performance

- Monte Carlo phases are typically suited to GPU
  - Calibration, RNG, path generation, payoff, reduction
- Application design leads to biggest speedup
  - Create batches of similar work
  - Reuse intermediate data if appropriate

Maximizing Productivity

- Leverage libraries
  - RNGs CURAND, NAG, MTGP
  - Thrust code.google.com/p/thrust/
  - CUSP code.google.com/p/cusp-library/
  - CUBLAS
  - CULAtools/MAGMA
  - CUFFT
  - …