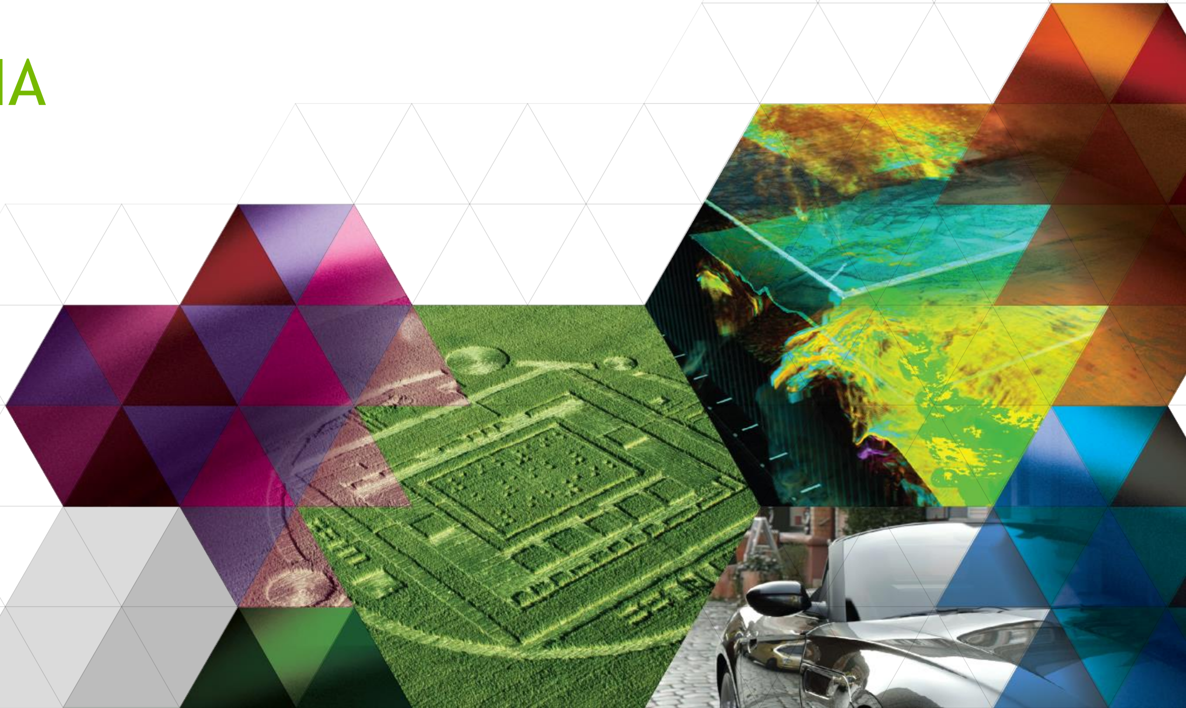
GPU TECHNOLOGY CONFERENCE

MONTE-CARLO SIMULATION OF AMERICAN OPTIONS WITH GPUS

Julien Demouth, NVIDIA



STAC-A2™ BENCHMARK

- STAC-A2™ Benchmark
  - Developed by banks
  - Macro and micro, performance ***and*** accuracy
  - Pricing and Greeks for American exercise basket option, correlated Heston dynamics, Longstaff Schwartz Monte Carlo
- Independently audited results
- GPU Solution
  - *“Over 9x the average speed of a system with the same class of CPUs but no GPUs”*
  - *“The first system to handle the baseline problem size in ‘real time’ (less than a second)”*

Please see **http://www.stacresearch.com/a2** for more details of the STAC-A2 Benchmark

Also see **http://devblogs.nvidia.com/parallelforall/american-option-pricing-monte-carlo-simulation** for more details on Longstaff-Schwartz Monte Carlo on GPUs

AMERICAN OPTIONS

- American put option on a stock
  - Alice buys a put option on a stock from Bob
  - Strike price **K**
  - Time to expiry **T**
  - Between now and time **T**, Alice *can* sell the stock to Bob at a price **K**

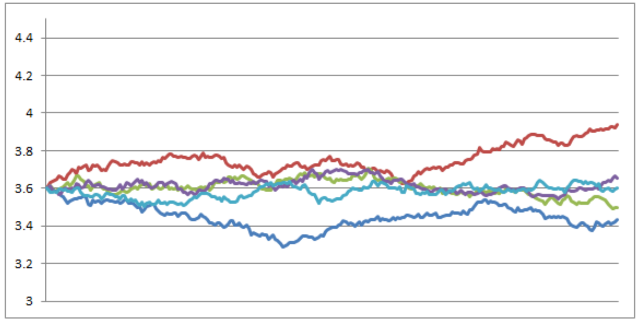
- Is today the right day to sell? How long should Alice wait?

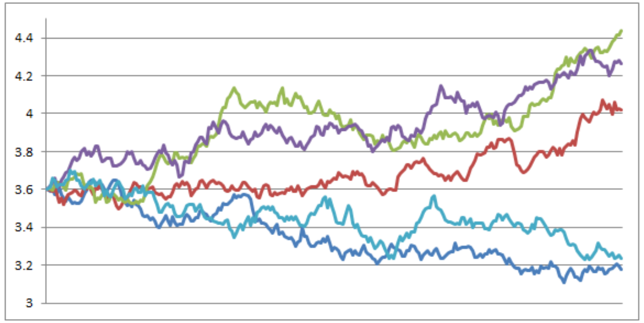
- The option pays off if K is higher than the stock price S

payoff = max(K - S[i], 0)

LONGSTAFF-SCHWARTZ ALGORITHM

- Generate random prices for the stock
  - Split the time to expiry **T** into **N** time steps: t0, t1, t2, …
  - Use **M** independent paths





- Different schemes to generate the stock prices:
  - Euler scheme
  - Andersen QE (used in STAC-A2)

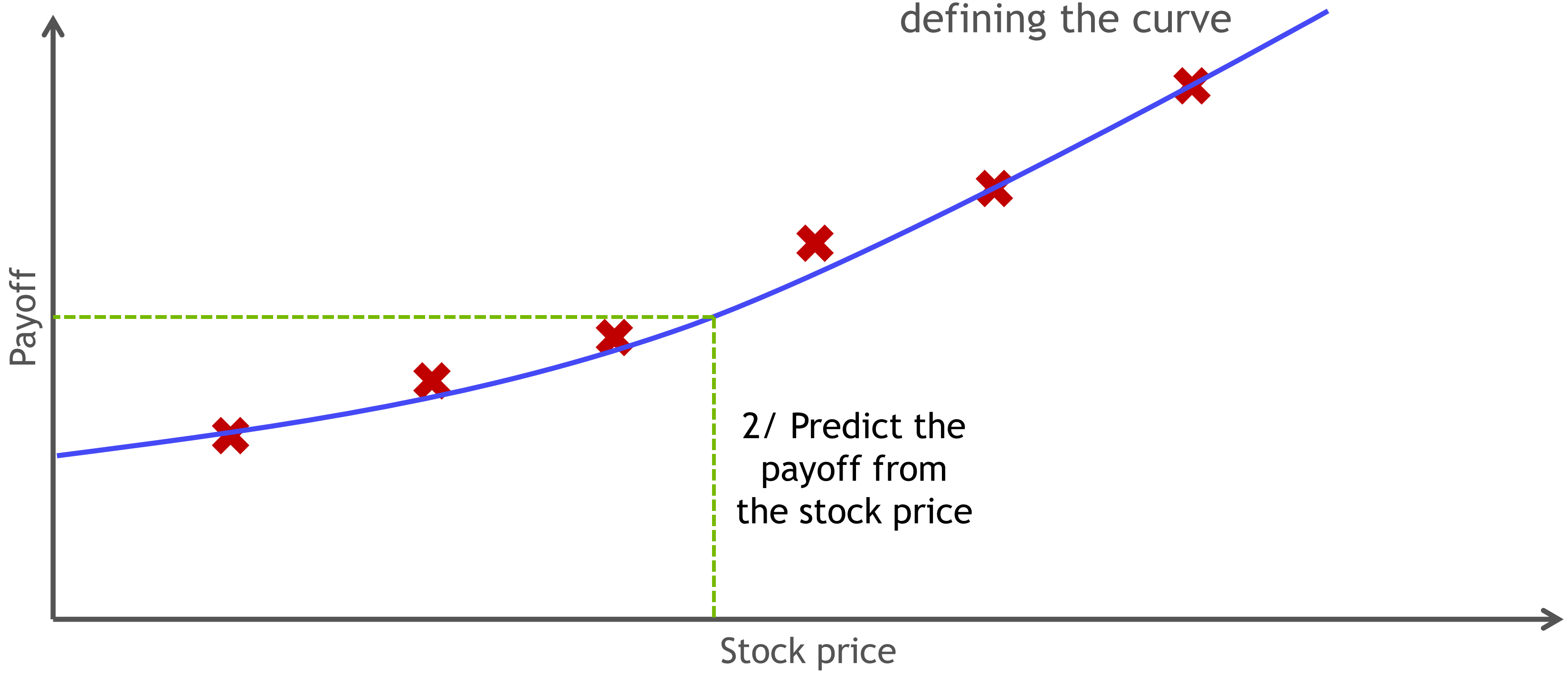
LONGSTAFF-SCHWARTZ ALGORITHM

- Compute the payoff at time T along each path
- Walk back in time

```
for( int ti = T-1 ; ti > 0 ; --ti )
```

- For each time step **ti**
  - Fit a model to predict the payoffs at **ti+1** from the stock prices at **ti**
    - Using the payoffs and stock prices from all the paths (*in the money*)
  - For each path
    - Predict the payoff for the path
    - Decide whether to exercise or continue on that path

LINEAR REGRESSION



1/ Find the coefficients (*beta*) defining the curve

2/ Predict the payoff from the stock price

LINEAR REGRESSION

- Linear model to fit

```
beta[0] + beta[1]*x + beta[2]*x^2
```

- We want to find **beta** which minimizes

```
|| A*beta – P ||^2
```

- **A** is the matrix of powers of stock prices, **P** vector of payoffs

```
    | 1  S[0]  S[0]^2 |                  | max(K-S[0], 0) |
A = | 1  S[1]  S[1]^2 |              P = | max(K-S[1], 0) |
    | 1  S[2]  S[2]^2 |                  | max(K-S[2], 0) |
    | …    …       …  |                  |       …        |
```

LINEAR REGRESSION

- We use the Singular Value Decomposition (SVD)

```
A = U * Sigma * V^T
```

- To build the (Moore-Penrose) pseudoinverse

```
V * Sigma^-1 * U^T
```

- And compute

```
beta = V * Sigma^-1 * U^T * P
```

- How can we efficiently build the pseudoinverse?

KEY DESIGN POINTS

- Expose as much parallelism as possible
  - Eliminate unneeded synchronization points
    - E.g. move computations outside of the main loop
  - Inside kernels, maximize the amount of independent work
    - E.g. threads do sequential work in parallel before a parallel reduction

- Reduce memory transfers to a minimum
  - Have coalesced memory accesses
    - E.g. map on thread per Monte Carlo path
  - Recompute rather than store intermediate results
    - E.g. do not store the square of S[i]

BUILD THE PSEUDOINVERSE

- Each **A** is a long-and-thin matrix with 32,000 rows x 3 columns
  - One matrix **A** per time step
  - It takes too much time and space to compute the SVD of **A** as-is

- A well-known approach: Build the QR decomposition of **A**

`A = QR`

- **R** is much smaller. Compute the SVD of **R** to build the SVD of **A**

`R = UR * SigmaR * VR^T  =>  A = Q * UR * SigmaR * VR^T`

- Since **R** is 3x3, we can compute its SVD on a multiprocessor

COMPUTE THE QR DECOMPOSITION

- Householder-based algorithm to build the QR decomposition
  - 3x dot products over ~32,000 elements
  - 3x 32,000x32,000 rank updates

- There are too many memory accesses!!!

- Our solution: **R** can be built using **8** scalars (see the code):

```
s0, s1, s2, Sum si^0, Sum si^1, Sum si^2, Sum si^3, Sum si^4
```

- Where **si** is the stock price on the **i-th** path which pays off

COMPUTE THE QR DECOMPOSITION

- During the main loop, **Q** can be built on-the-fly using **A** and **R**

$$Q = AR^{-1}$$

- In summary, we build all the **W** matrices before the main loop

$$W = VR * SigmaR^{-1} * UR^T$$

- Each CUDA block computes a different **W**

- At each iteration of the main loop, we compute **beta** as

$$\text{beta} = \overset{\text{Pseudoinverse}}{\boxed{W * \underbrace{\boxed{(R^{-1})^T * A^T}}_{Q^T}}} * P$$

BUILD THE PSEUDO-INVERSE

- Before the main loop, we build **W** (one block per time step)

```
int m = 0; double4 sums = {0.0};

// Iterate over the paths. Each thread computes its own partial sums.
for( int path = threadIdx.x ; path < num_paths ; path += THREADS_PER_BLOCK )
{
  // Load the asset price.
  double S = paths[offset + path];

  // Update the sums if the path pays off.
  if( payoff.is_in_the_money(S) ) {
    ++m;
    double S2 = S*S;
    sums.x += S; sums.y += S2; sums.z += S2*S; sums.w += S2*S2;
  }
}
m = cub::BlockReduce<...>(...).Sum(m); sums = cub::BlockReduce<...>(...).Sum(sums);

// Build and store W. See the code.
```

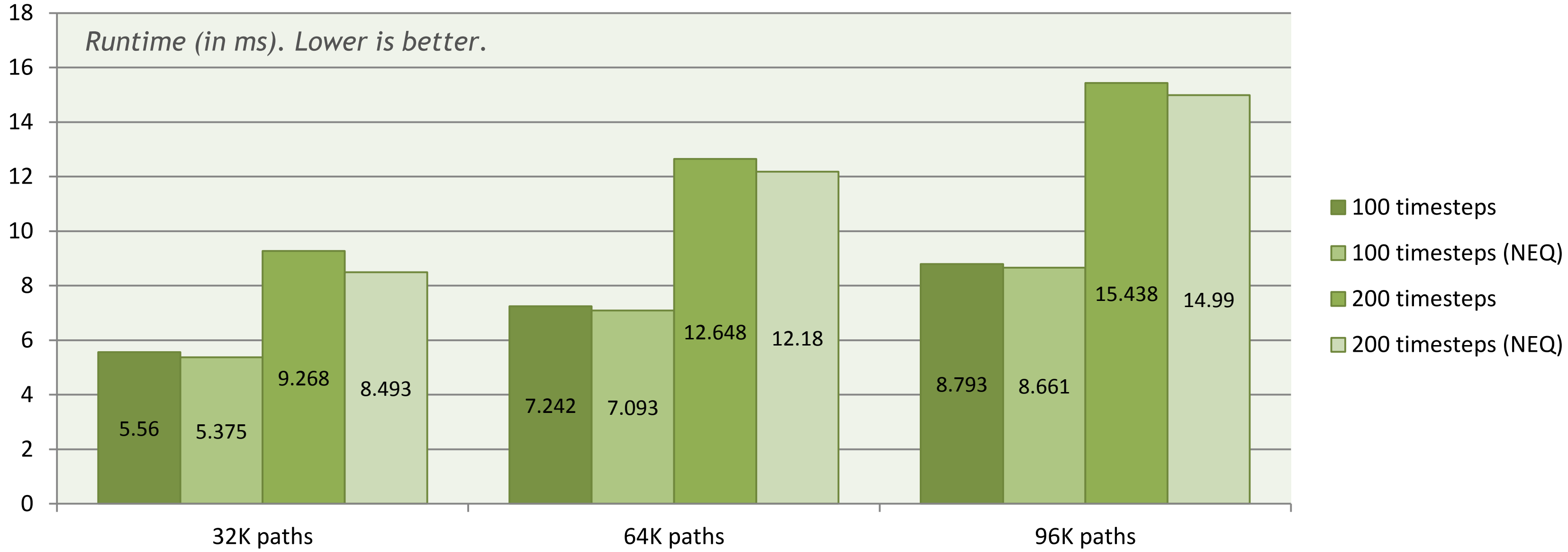
MAIN LOOP

- **W** is a 3x3 matrix and **R** has only 6 non-zero values
- We map one CUDA thread per path (or more)

```
if( threadIdx.x < 15 ) // Load W for the block.
  smem_W[threadIdx.x] = W[threadIdx.x];
__syncthreads();

double3 beta = {0.0}; // Each thread computes a partial sum of beta.

// Iterate over the paths.
for( int path = tidx ; path < num_paths ; path += blockDim.x*gridDim.x )
{
  double S = stock[path]; double S2 = S*S; // Rebuild A on the fly

  ... // Update beta. No global memory access!!!
}

beta = cub::BlockReduce<...>(...).Sum(beta); // Parallel reduction

... // Store beta
```

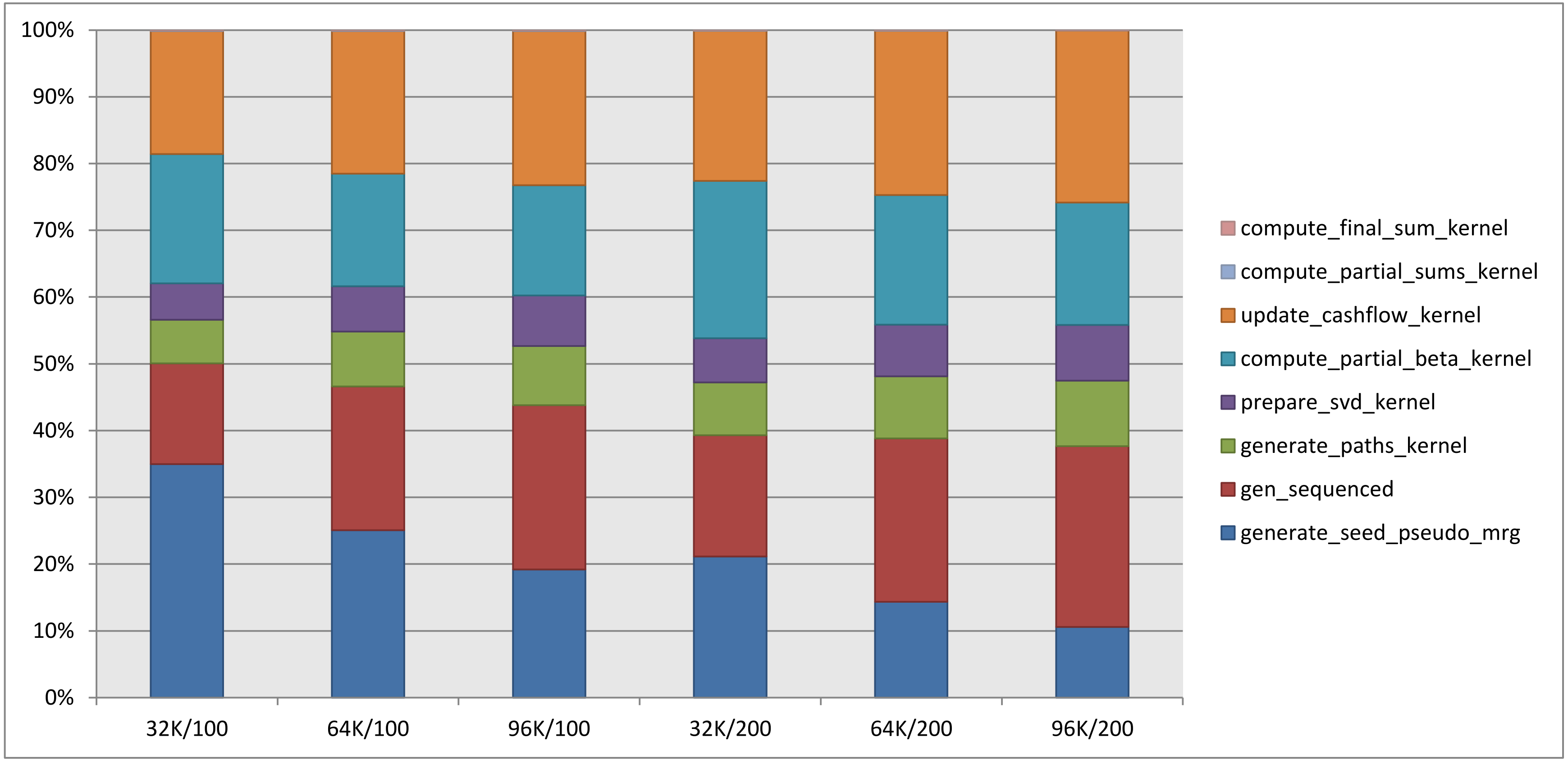
PERFORMANCE RESULTS

- Tesla K40 (875MHz, 3004MHz), runtime in milliseconds



- NEQ: Linear regression using the Normal Equation
- Timings include the generation of paths

PERFORMANCE RESULTS



PERFORMANCE RESULTS

- The importance of the main loop (update_cashflow/compute_partial_beta)
  - Increases with the number of time steps
  - At 32K/64K paths, the two kernels are limited by latency
    - High impact of instruction and constant cache misses
  - The loop is impacted by launch latency of kernels (for #paths <= 32K)
    - See how to reduce the impact in the companion code (#define WITH_FUSED_BETA)

- On 32K/64K paths, we have a limited number of CUDA blocks
  - Tail effects (load balancing is not optimal)
  - We need more paths or work on several problems in parallel
    - Idea: Use several CPU threads and CUDA streams
    - Keep it in mind when you design your infrastructure

CONCLUSION

- GPUs are good at American option pricing
  - See our STAC-A2 results (compared to high-end CPUs/GPU-like)
- Robust algorithms like the SVD can be implemented

- Our blog post:
- http://devblogs.nvidia.com/parallelforall/american-option-pricing-monte-carlo-simulation/
- The companion code:
- https://github.com/parallel-forall/code-samples/tree/master/posts/american-options
- Our STAC-A2 results:
- http://www.stacresearch.com/nvidia/4dec13