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](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](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 $S$
  \[
  \text{payoff} = \max(K - S[i], 0)
  \]
LONGSTAFF-SCHWARTZ ALGORITHM

- Generate random prices for the stock
  - Split the time to expiry $T$ into $N$ time steps: $t_0, t_1, t_2, ...$
  - 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

  \[
  \text{for}( \text{int } ti = T-1; \text{ ti } > 0; \text{ --ti })
  \]

- 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 (\textit{in the money})
  - For each path
    - Predict the payoff for the path
    - Decide whether to exercise or continue on that path
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

\[
\begin{bmatrix}
1 & S[0] & S[0]^2 \\
\vdots & \vdots & \vdots \\
1 & S[N] & S[N]^2 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\max(K-S[0], 0) \\
\max(K-S[1], 0) \\
\max(K-S[2], 0) \\
\vdots \\
\max(K-S[N], 0) \\
\end{bmatrix}
\]
LINEAR REGRESSION

- We use the Singular Value Decomposition (SVD)
  \[ A = U \times \text{Sigma} \times V^T \]
- To build the (Moore-Penrose) pseudoinverse
  \[ V \times \text{Sigma}^{-1} \times U^T \]
- And compute
  \[ \beta = V \times \text{Sigma}^{-1} \times U^T \times \text{P} \]

- 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 $A$

  $R = UR \ast \Sigma R \ast VR^T \Rightarrow A = Q \ast UR \ast \Sigma R \ast VR^T$

- 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):

  \[ s_0, s_1, s_2, \text{Sum } s_i^0, \text{Sum } s_i^1, \text{Sum } s_i^2, \text{Sum } s_i^3, \text{Sum } s_i^4 \]

- Where \( s_i \) is the stock price on the \text{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
  \[ W = VR * \Sigma R^{-1} * UR^T \]

- Each CUDA block computes a different $W$

- At each iteration of the main loop, we compute $\beta$ as
  \[ \beta = W * (R^{-1})^T * A^T * P \]
Before the main loop, we build \( W \) (one block per time step)

\[
\begin{align*}
\text{int } m &= 0; \quad \text{double4 } \text{sums} = \{0.0\}; \\
\text{// Iterate over the paths. Each thread computes its own partial sums.} \\
\text{for( int } \text{path} = \text{threadIdx.x} ; \text{path } &< \text{num_paths} ; \text{path } += \text{THREADS_PER_BLOCK} ) \\
\{ \\
\text{// Load the asset price.} \\
\text{double } S &= \text{paths[offset + path]}; \\
\text{// Update the sums if the path pays off.} \\
\text{if( payoff.is_in_the_money(S) ) } \{ \\
\text{++m; } \\
\text{double } S2 &= S*S; \\
\text{sums.x } &= S; \text{ sums.y } &= S2; \text{ sums.z } &= S2*S; \text{ sums.w } &= S2*S2; \\
\} \\
\} \\
\text{m } &= \text{cub::BlockReduce<...>>(...).Sum(m); sums } = \text{cub::BlockReduce<...>>(...).Sum(sums);} \\
\text{// Build and store W. See the code.}
\end{align*}
\]
MAIN LOOP

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

```c
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 Graph]

- compute_final_sum_kernel
- compute_partial_sums_kernel
- update_cashflow_kernel
- compute_partial_beta_kernel
- prepare_svd_kernel
- generate_paths_kernel
- gen_sequenced
- generate_seed_pseudo_mrg
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:

- The companion code:

- Our STAC-A2 results: