GPU TECHNOLOGY CONFERENCE

PRICING AMERICAN OPTIONS WITH LEAST SQUARES MONTE CARLO ON GPUS

Massimiliano Fatica, NVIDIA Corporation

OUTLINE

- Overview
- Least Squares Monte Carlo
- GPU implementation
- Results
- Conclusions

OVERVIEW

- Valuation and optimal exercise of American-style options is a very important practical problem in option pricing

- Early exercise feature makes the problem challenging:
  - On expiration date, the optimal exercise strategy is to exercise if the option is in the money or let it expire otherwise
  - For all the other time steps, the optimal exercise strategy is to examine the asset price, compare the immediate exercise value of the option with the risk neutral expected value of holding the option and determine if immediate exercise is more valuable

OVERVIEW

- Algorithms for American-style options:
  - Grid based (finite difference, binomial/trinomial trees)
  - Monte Carlo
- GPUs are very attractive for High Performance Computing
  - Massive multithreaded chips
  - High memory bandwidth, high FLOPS count
  - Power efficient
  - Programming languages and tools
- This work will present an implementation of the Least Squares Monte Carlo method by Longstaff and Schwartz (2001) on GPUs

LEAST SQUARES MONTE CARLO

- If N is the number of paths and M is the number of time intervals:
  - Generate a matrix R(N,M) of normal random numbers
  - Compute the asset prices S(N,M+1)
  - Compute the cash flow at M+1 since the exercise policy is known
- For each time step, going backward in time:
  - Estimate the continuation value
  - Compare the value of immediate payoff with continuation value and decide if early exercise
- Discount the cash flow to present time and average over paths

LONGSTAFF - SCHWARTZ

- Estimation of the continuation value by least squares regression using a cross section of simulated data:
  - Continuation function is approximated as linear combination of basis functions

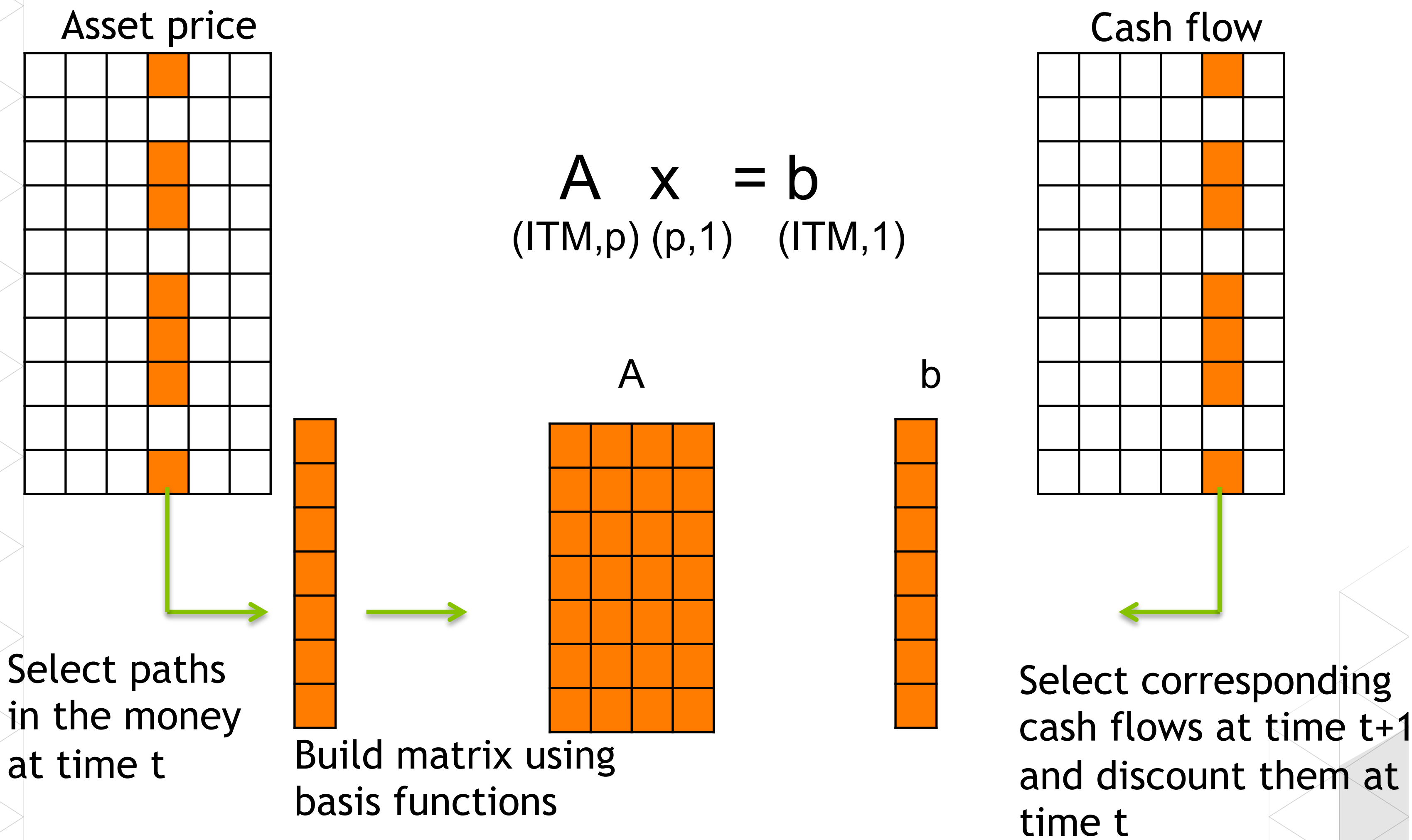
$$F(.,t_k) = \sum_{k=0}^{p} \alpha_k L_k(S(t_k))$$

  - Select the paths in the money
  - Select basis functions: monomial, orthogonal polynomials (weighted Laguerre,…)

$$L_k(S) = S^k$$

$$\begin{aligned} L_0(S) &= e^{-S/2} \\ L_1(S) &= e^{-S/2}(1-S) \\ L_2(S) &= e^{-S/2}(1-2S+S^2/2) \\ L_k(S) &= e^{-S/2}\frac{e^S}{k!}\frac{d^k}{dS^k}(S^k e^{-S}) \end{aligned}$$

LEAST SQUARES REGRESSION



LEAST SQUARES MONTE CARLO

RNG	Plenty of parallelism
Moment matching	Plenty of parallelism
Path generation	Plenty of parallelism if N is large
Regression	M dependent steps.
Average	Plenty of parallelism

RANDOM NUMBER GENERATION

- Random number generation is performed using the CURAND library:
  - Single and double precision
  - Normal, uniform, log-normal, Poisson distributions
  - 4 different generators:
    - XORWOW: xor-shift
    - MTGP32: Mersenne-Twister
    - MRG32K32A: Combined Multiple Recursive
    - PHILOX4-32: Counter-based

RANDOM NUMBER GENERATION

- Choice of:
  - Normal distribution
  - Uniform distribution plus Box-Muller:

$$n_0 = \sqrt{-2\log(u_1)}\sin(2\pi u_0)$$

$$n_1 = \sqrt{-2\log(u_1)}\cos(2\pi u_0)$$

- Optional moment matching of the data

$$n_i^* = \frac{(n_i - \mu)}{\sigma}$$

RNG GENERATION

```
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_PHILOX4_32_10);
curandSetPseudoRandomGeneratorSeed(gen,myseed);
if(bm==0) { /* Generate LDA*M double with normal distribution on device */
           curandGenerateNormalDouble(gen,devData, LDA*M,0.,1.); }
      else{
           /* Generate LDA*M doubles with uniform distribution on device and then apply Box-Muller transform */
             curandGenerateUniformDouble(gen,devData, LDA*M);
             box_muller<<<256,256>>>(devData,LDA*M);
           }
.....
curandDestroyGenerator(gen);

__global__ void box_muller(double *in,size_t N) {
    int tid = threadIdx.x;
    int totalThreads = gridDim.x * blockDim.x;
    int ctaStart = blockDim.x * blockIdx.x;
    double s,c;
    for (size_t i = ctaStart + tid ; i < N/2; i += totalThreads) {
       size_t ii=2*i;
       double x=-2*(log(in[ii]));
       double y=2*in[ii+1];
       sincospi(y,&s,&c);
       in[ii]  =sqrt(x)*s;
       in[ii+1]=sqrt(x)*c;
       }
}
```

PATH GENERATION

- The stock price S(t) is assumed to follow a geometric Brownian motion

$$\begin{aligned} S_i(0) &= S0 \\ S_i(t+\Delta t) &= S_i(t) e^{(r-\frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t} Z_i} \end{aligned}$$

- Use of antithetic variables:

$$S_i(t+\Delta t) = S_i(t) e^{(r-\frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t} Z_i}$$

$$S_i^*(t+\Delta t) = S_i^*(t) e^{(r-\frac{\sigma^2}{2})\Delta t - \sigma\sqrt{\Delta t} Z_i}$$

  – reduce variance
  – reduce memory footprint

PATH GENERATION

```
__global__ void generatePath(double *S, double *CF, double *devData, double  S0, double K,
                             double R, double sigma, double dt, size_t N, int M, size_t LDA)
{
    int i,j;
    int totalThreads = gridDim.x * blockDim.x;
    int ctaStart = blockDim.x * blockIdx.x;

      for (i = ctaStart + threadIdx.x; i < N/2; i += totalThreads) {
       int ii=2*i;
       S[ii]=S0;
       S[ii+1]=S0;
       \\ Compute asset price at all time steps
       for (j=1;j<M+1;j++)
       {
        S[ii+  j*LDA]=S[ii  +(j-1)*LDA]*exp( (R-0.5*sigma*sigma)*dt + sigma*sqrt(dt)*devData[i+(j-1)*LDA] );
        S[ii+1+j*LDA]=S[ii+1+(j-1)*LDA]*exp( (R-0.5*sigma*sigma)*dt - sigma*sqrt(dt)*devData[i+(j-1)*LDA] );
       }
       \\ Compute cash flow at time T
       CF[ii  +M*LDA]=( K-S[ii  +M*LDA]) >0. ?  (K-S[ii+  M*LDA]): 0.;
       CF[ii+1+M*LDA]=( K-S[ii+1+M*LDA]) >0. ?  (K-S[ii+1+M*LDA]): 0.;
    }
}
```

Simple parallelization. Each thread computes multiple antithetic paths

LEAST SQUARES SOLVER

- System solved with normal equation approach

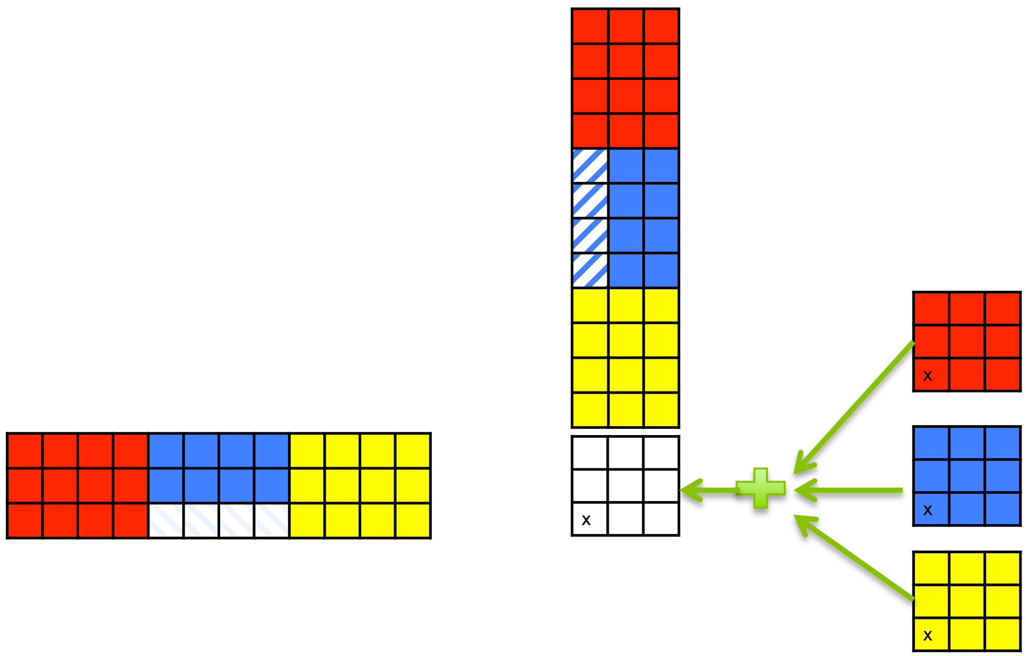
$$Ax = b \longrightarrow A^T A\, x = A^T b$$

- The element (l,m) of $A^T A$ and the element l of $A^T b$:

$$\sum_{j \in ITM} L_l(j) L_m(j) \qquad \sum_{j \in ITM} L_l(j) b(j)$$

- The matrix A is never stored, each thread loads the asset price and cash flow for one path and computes the terms on-the fly, adding them to the sum if the path is in the money
- Two stages approach, possible use of compensated sum and extended precision

COMPUTATION OF A^TA



RESULTS

- CUDA 5.5
- Tesla K20X
  - 2688 cores
  - 732 MHz
  - 6 GB of memory
- Tesla K40
  - 2880 cores
  - Boost clock up to 875 MHz
  - 12 GB of memory

RNG PERFORMANCE

Generator	Distribution	Time (ms) $N=10^7$	Time (ms) $N=10^8$
XORWOW	Normal	12.99	34.03
XORWOW	**Uniform +Box Muller**	12.65	30.93
MTGP32	**Normal**	3.48	32.95
MTGP32	Uniform +Box Muller	3.92	37.53
MRG32K	Normal	4.46	26.44
MRG32K	**Uniform +Box Muller**	4.02	22.02
PHILOX	Normal	2.89	27.40
PHILOX	**Uniform +Box Muller**	2.53	24.12

COMPARISON WITH LONGSTAFF-SCHWARTZ

S	σ	T	Finite difference	Longstaff paper	GPU
36	.20	1	4.478	4.472	4.473
36	.20	2	4.840	4.821	4.854
36	.40	1	7.101	7.091	7.098
36	.40	2	8.508	8.488	8.501
38	.20	1	3.250	3.244	3.248
38	.40	2	3.745	3.735	3.746
38	.20	1	6.148	6.139	6.138
38	.40	2	7.670	7.669	7.663
44	.20	1	1.110	1.118	1.112
44	.40	2	1.690	1.675	1.684
44	.20	1	3.948	3.957	3.944
44	.40	2	5.647	5.622	5.627

Finite differences: implicit scheme with 40000 time steps per year, 1000 steps p
LSMC with 100000 path and 50 time steps. Philox generator for GPU results.

ACCURACY VS QR SOLVER

Put option with strike price=40, stock price=36, variability=.2, r=.06, T=2
Reference value is 4.840

Basis functions	Normal Equation (GPU)	QR (CPU)
2	4.740095193796793	4.740095193796793
3	4.815731393932048	4.815731393932048
4	4.833172186198728	4.833172186198728
5	4.833251309474664	4.833251309474664
6	4.835805059904685	4.836251721596485
7	4.837584550853037	4.837803730367345
8	4.838283073214879	4.839358646526560

Regression coefficients at the final step for 4 basis functions

Normal equation	155.156982160074	-397.156557357517	353.391768458585	-108.545827892269
QR	155.157227263422	-397.157372674635	353.392671772303	-108.546161230726

RESULTS DOUBLE PRECISION

```
nvprof ./american_dp -g3
American put option N=524288 (LDA=524288)  M=50 dt=0.020000
Strike price=40.000000 Stock price=36.000000 sigma=0.200000 r=0.060000 T=1.000000

Generator: MRG
BlackScholes put = 3.844
Normal distribution
RNG generation time = 8.763488 ms
Path generation time = 3.288832 ms
LS time = 7.512192 ms, perf = 136.792 GB/s
GPU Mean price  =4.476522e+00


Time      Calls    Avg       Min       Max      Name
6.4127ms     1   6.4127ms  6.4127ms  6.4127ms  gen_sequenced<curandStateMRG32k3a
3.4666ms    49   70.746us  69.632us  71.680us  second_kernel
3.2417ms     1   3.2417ms  3.2417ms  3.2417ms  generatePath
3.0826ms    49   62.909us  61.856us  63.840us  tall_gemm
1.9224ms     1   1.9224ms  1.9224ms  1.9224ms  generate_seed_pseudo_mrg
480.03us    49   9.7960us  9.5360us  10.208us  second_pass
126.24us     1   126.24us  126.24us  126.24us  redusum
12.384us     3   4.1280us  3.7760us  4.3200us  [CUDA memset]
11.936us     1   11.936us  11.936us  11.936us  BlackScholes
5.7920us     2   2.8960us  2.8480us  2.9440us  [CUDA memcpy DtoH]
```

RESULTS SINGLE PRECISION

```
nvprof ./american_sp -g3
American put option N=524288 (LDA=524288)  M=50 dt=0.020000
Strike price=40.000000 Stock price=36.000000 sigma=0.200000 r=0.060000 T=1.000000

Generator: MRG
BlackScholes put = 3.844
Normal distribution
RNG generation time = 5.920544 ms
Path generation time = 1.882912 ms
LS time = 6.319168 ms, perf = 162.617 GB/s
GPU Mean price  =4.475582e+00
```

Time	Calls	Avg	Min	Max	Name
3.5837ms	1	3.5837ms	3.5837ms	3.5837ms	gen_sequenced<curandStateMRG32k3a
3.0940ms	49	63.142us	61.984us	64.256us	tall_gemm
2.2367ms	49	45.646us	44.608us	46.688us	second_kernel
1.9065ms	1	1.9065ms	1.9065ms	1.9065ms	generate_seed_pseudo_mrg
1.8345ms	1	1.8345ms	1.8345ms	1.8345ms	generatePath
505.38us	49	10.313us	10.144us	10.656us	second_pass
127.71us	1	127.71us	127.71us	127.71us	redusum
12.544us	3	4.1810us	3.8080us	4.3840us	[CUDA memset]
7.8080us	1	7.8080us	7.8080us	7.8080us	BlackScholes
5.7280us	2	2.8640us	2.7840us	2.9440us	[CUDA memcpy DtoH]

PERFORMANCE COMPARISON WITH CPU

- 256 time steps, 3 regression coefficients
- CPU and GPU runs with double precision, MRGK32A RNG

Paths	Sequential*	Xeon E5-2670* (OpenMP, vect)	K20X	K40	K40 ECC off
128K	4234ms	89ms	26.5ms	22.9ms	21.2ms
256K	8473ms	171ms	43.9ms	38.0ms	35.1ms
512K	17192ms	339ms	78.8ms	67.7ms	63.2ms

For the GPU version going from 3 terms to 6 terms only increases the runtime to 66.4ms. The solve phase goes from 27.8ms to 30.8ms.

* Source Xcelerit blog

CONCLUSIONS

- Successfully implemented the Least Squares Monte Carlo method on GPU
- Correct and fast results
- Future work:
  - QR decomposition on GPU

Massimiliano Fatica and Everett Phillips (2013) “Pricing American options with least squares Monte Carlo on GPUs”. In *Proceedings of the 6th Workshop on High Performance Computational Finance* (WHPCF '13). ACM, New York, NY, USA,