GPU-based Quasi-Monte Carlo Algorithms for Matrix Computations

Aneta Karaivanova

(with E. Atanassov, S. Ivanovska, M. Durchova)

anet@parallel.bas.bg

Institute of Information and Communication Technologies

Bulgarian Academy of Sciences

Acad. G. Bonchev St., Bl.25A

1113 Sofia, Bulgaria

Outline of the Talk

- Formulation of the Problems
- MCMs and QMCMs for Matrix-Vector Products
- Scrambling: purpose and methods
- Owen scrambling on GPUs
- Some Numerical Results

Formulation of the Problems

Given $A \in R^{n \times n}$, $f \in R^n$ consider the problems:

- Find the scalar product of a given vector and the solution and of a system of linear algebraic equations:

$$x = Ax + f$$

- Find some of the elements of the inverse matrix $C = A^{-1}$
- Find (λ, x), $x \in R^n$, such that

$$Ax = \lambda x$$

Solving SLAE via Neumann Series

- Assume a SLAE in the form $x = Ax + f$, all the eigenvalues of A lie within the unit circle.
- Given x_0, consider the sequence:

$$x^{(k)} = Ax^{(k-1)} + f, \quad k = 1, 2, \ldots.$$

- The approximate solution is the truncated Neumann series

$$x^{(k)} = f + Af + A^2 f + \ldots + A^{(k-1)} f + A^k x^{(0)}, \quad k > 0$$

with a truncation error of $x^{(k)} - x = A^k (x^{(0)} - x)$.

- We can estimate (h, x), where h is a given vector as:

$$(h, x) \approx h^T f + h^T Af + h^T A^2 f + \ldots + h^T A^{(k-1)} f + h^T A^k x^{(0)}, \quad k > 0.$$

- We can use MCM and QMCM for computing matrix-vector products

Computing Extremal Eigenvalues: Power Method

- The eigenvalue problem:

$$A \in R^{n \times n}, \quad u \in R^n$$

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_{n-1}| > |\lambda_n|.$$

$$Au = \lambda u$$

- The power method:

$$x^m = Ax^{m-1}/||Ax^{m-1}||$$

$$\lambda^{(m)} = \frac{(h, A^m f)}{(h, A^{m-1} f)} \underset{m \to \infty}{\longrightarrow} \lambda_{max}$$

- We can apply MCM and QMCM for computing matrix-vector products

Computing Extremal Eigenvalues: The Resolvent Method

- The resolvent matrix: $R_q = [I - qA]^{-1} \in {}^{n\times n}$

$$[I - qA]^{-m} = \sum_{i=1}^{\infty} q^i C_{m+i-1}^i A^i, \quad |q\lambda| < 1$$

- Connection between eigenvalues: $\mu = \frac{1}{1-q\lambda}$

  $q > 0$, μ_{max} corresponds to λ_{max},

  $q < 0$, μ_{max} corresponds to λ_{min}

- Power method for the resolvent matrix:

$$\mu^{(m)} = \frac{(h, [I - qA]^{-m} f)}{(h, [I - qA]^{-(m-1)} f)} \underset{m\to\infty}{\longrightarrow} \mu = \frac{1}{1 - q\lambda}.$$

- The Resolvent estimation:

$$\lambda = \frac{1}{q}\left(1 - \frac{1}{\mu}\right) = \frac{\sum_{i=1}^{\infty} q^{i-1} C_{i+m-2}^{i-1} (h, A^i f)}{\sum_{i=0}^{\infty} q^i C_{i+m-1}^i (h, A^i f)}.$$

MCM for Matrix-Vector Products

- Consider $(h, A^m f) = h^T A^m f$ Markov chain: $k_0 \to k_1 \to \ldots \to k_m$
- Possible choices for initial and transition densities:

$$p_\alpha = \frac{1}{n}, \;\; p_{\alpha\beta} = \frac{1}{n}, \quad \text{(crude Monte Carlo)}$$

$$p_\alpha = \frac{|h_\alpha|}{\sum_{\alpha=1}^n |h_\alpha|}; \;\; p_{\alpha\beta} = \frac{|a_{\alpha\beta}|}{\sum_{\beta=1}^n |a_{\alpha\beta}|}, \;\; \alpha = 1, \ldots, n, \quad \text{(importance sampling)}$$

Random variable: $\theta = \frac{h_{k_0}}{p_{k_0}} W_m f_{k_m}$ where

$$W_0 = 1, \;\; W_j = W_{j-1} \frac{a_{k_{j-1} k_j}}{p_{k_{j-1} k_j}}, j = 1, \ldots, m$$

$$E[\theta] = h^T A^m f \approx \frac{1}{N} \sum_{s=1}^{N} (\theta)_s$$

with a typical statistical error of size $O(Var(\theta)^{1/2} N^{-1/2})$.

The mathematical expectations

- The matrix-vector products:

$$(h, A^m f) = E[W_m f_{k_m}], \quad i = 1, 2, \ldots.$$

$$(h, [I - qA]^{-m} f) = E[\sum_{i=0}^{\infty} q^i C_{i+m-1}^i W_i f(x_i)].$$

- Scalar product (h, x) for $x = Ax + f$ and given vector h

$$(h, x) = E[\sum_{i=0}^{\infty} W_i f_{k_i}]$$

- Extremal eigenvalues: $\lambda_{max} \approx \frac{E[W_m f_{k_m}]}{E[W_{m-1} f_{k_{m-1}}]}$

$$\lambda_{min} = \frac{1}{q}\left(1 - \frac{1}{\mu}\right) \approx \frac{E[\sum_{i=1}^{\infty} q^{i-1} C_{i+m-2}^{i-1} W_i f(x_i)]}{E[\sum_{i=0}^{\infty} q^i C_{i+m-1}^i W_i f(x_i)]}.$$

The MC and QMC estimations

- Scalar product of the solution $(x = Ax + f)$

$$(h, x) \approx \frac{1}{N} \sum_{s=1}^{N} \sum_{i=0}^{l} (W_i f_{k_i})_s$$

- Elements of the inverse matrix $A^{-1} = C = \{c_{rr'}\}$

$$c_{rr'} \approx \frac{1}{N} \sum_{s=1}^{N} \left[\sum_{(j|k_j=r')} W_j \right]_s$$

Computational complexity: lN

Convergence:

$$O(\frac{||A||^l ||r^{(0)}||}{1 - ||A||} + \sigma N^{-1/2}) \quad \text{and} \quad O(\frac{||A||^l ||r^{(0)}||}{1 - ||A||} + (log^l N) N^{-1})$$

The MC and QMC estimations (cont.)

- Largest eigenvalue (Computational complexity: mN_w):

$$\lambda_{max} \approx \frac{\sum_{s=1}^{N}(W_m f_{k_m})_s}{\sum_{s=1}^{N}(W_{m-1} f_{k_{m-1}})_s}$$

$$O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^m + \sigma N^{-1/2}\right) \quad \text{and} \quad O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^m + (log^m N)N^{-1}\right)$$

- Smallest eigenvalue (Computational complexity: $4lN_w$):

$$\lambda_{min} = \approx \frac{\sum_{s=1}^{N}([\sum_{i=0}^{l} q^i C_{i+m-1}^{i-1} W_{i+1} f(x_{i+1})])_s}{\sum_{s=1}^{N}([\sum_{i=0}^{l} q^i C_{i+m-1}^{i} W_i f(x_i)])_s}.$$

$$O\left(\left|\frac{\mu_2}{\mu_1}\right|^m + \sigma N^{-1/2}\right) \quad \text{and} \quad O\left(\left|\frac{\mu_2}{\mu_1}\right|^m + (log^l N)N^{-1}\right)$$

The purpose of scrambling

- To improve 2-D projections and the quality in general of sequences
- To provide practical method to obtain error estimates for QMC
- To provide a simple and unified way to generate quasirandom numbers for parallel, distributed and grid-computing environments
- To provide more choices of QRN sequences with better (often optimal) quality to be used in QMC applications

Poor 2-d projections: Generic

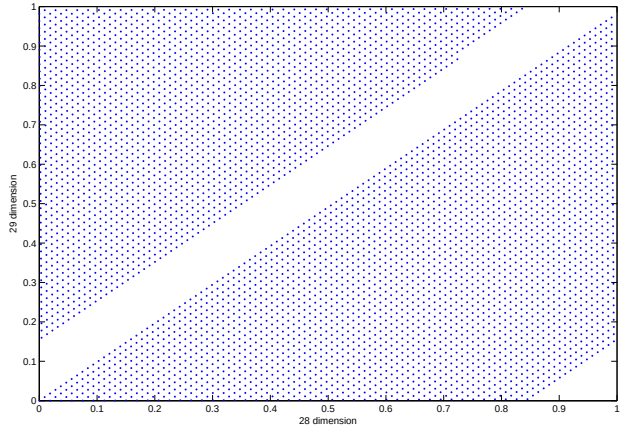

Figure 1: *Halton sequence, 4096 points, dimesions 28 and 29*

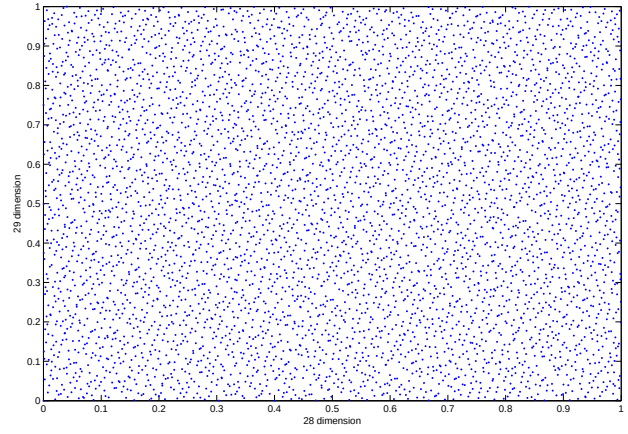

Figure 2: *Scrambled Halton sequence, 4096 points, dimesions 28 and 29*

Scrambling techniques

- Scrambling was first proposed by Cranley and Patterson (1976) who took lattice points and randomized them by adding random shifts to the sequences. Later, Owen and Tezuka independently developed two powerful scrambling methods through permutations
- Although many other methods have been proposed, most of them are modified or simplified Owen or Tezuka schemes.
- Three basic methods of scrambling
  - randomized shifting
  - digital permutations
  - permuting the order of points within the sequence
- The problem with Owen scrambling is its computational complexity.

Owen scrambling

Let $x_n = (x_n^{(1)}, x_n^{(2)}, \ldots, x_n^{(s)})$ be a quasirandom number in $[0,1)^s$, and let $z_n = (z_n^{(1)}, z_n^{(2)}, \ldots, z_n^{(s)})$ be the scrambled version of the point x_n. Suppose each $x_n^{(j)}$ can be represented in base b as $x_n^{(j)} = (0.x_{n1}^{(j)} x_{n2}^{(j)} \ldots x_{nK}^{(j)} \ldots)_b$ with K being the number of digits to be scrambled. Then nested scrambling proposed by Owen can be defined as follows:

$$z_n^{(j)} = \sigma(x_n^{(j)}), j = 1, 2, \ldots, s,$$

where $\sigma = \{\pi_1, \pi_2, \ldots, \pi_K\}$, $z_{ni}^{(j)} = \pi_i(x_{ni}^{(j)})$, for $i = 1, 2, \ldots, K$. Here π_i is a uniformly chosen permutation of the digits, $\{0, \ldots, b-1\}$. Of course, (t, m, s)-net remains (t, m, s)-net under nested scrambling. However, nested scrambling requires b^{i-1} permutations to scramble the ith digit. Owen scrambling (nested scrambling), which can be applied to all (t, s)-sequences, is powerful; however, from the implementation point-of-view, nested scrambling or so-called path dependent permutations requires a considerable amount of bookkeeping, and leads to more problematic implementation.

GPU computing

GPGPU /General-Purpose computation on Graphics Processing Units/ is known as /GPU Computing/. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput.

- For the tests we used NVIDIA GTX 295 graphics card working under Scientific Linux 4.7, which are dual cards with 2×240 cores, offering close to 2 Teraflops in single precision for each Grid Worker Node.
- The GPU is good for data-parallel processing
- The same computation executed on many data elements in parallel - low control flow overhead with high SP floating point arithmetic intensity
- Many calculations per memory access

Owen scrambling for Sobol sequence on GPU

The algorithm for scrambling the nth coordinate of the Nth term of the sequence works as follows:

- Input - coordinate n, term of the sequence without scrambling $\tau = 0.t_1t_2t_3\ldots$, initial seed for the pseudorandom function family s
- Scramble 9 consecutive bits of τ using a sequence of 512 random bits, generated from the initial seed s, the coordinate n and the sequence of bits scrambled so far. The 512 random bits are interpreted as a random binary tree, where the root of the tree is bit number 0 and if the jth random bit corresponds to a node of the binary tree bit number $2 * j + 1$ corresponds to the left node and leaf number $2 * j + 2$ corresponds to the right node. The fi rst bit of the consecutive 9 bits is changed if and only if the root of the tree is 1, and in such case we move to the right leaf, otherwise we move to the left leaf. We continue until we have scrambled all the 9 bits.
- Continue with the next 9 bits until we have scrambled all the bits of τ. We used the SALSA function with 20 rounds. It is amenable to implementation on a GPU since it uses only small number of variables, uses integer operations and does not involve high number of logical operations and conditional statements.

Owen scrambling for Halton sequence on GPU

- Following the Owen definition for scrambling, we should generate independent permutations of the digits $0 \ldots p-1$ for each p.
- We use a pseudorandom function, defined as part of the chacha family of stream cyphers (http://cr.yp.to/chacha.html). Suppose that for 128 bit x this function produces "random" 512-bit output $h(x)$.
- Our idea is to obtain a random permutation by ordering a sequence of p random numbers. We generate these random numbers using the pseudorandom function that we outlined before. We take advantage of the fact that the first few binary digits are usually enough to decide which number is bigger, and we generate the next random digits only if they are necessary for resolving "collisions".
- Suppose $x = 0, a_1 a_2 \ldots$ in p-adic number system and suppose that $p < 64$. Choose random seed s.

Owen scrambling for Halton sequence on GPU (cont.)

- We interpret the output of the pseudorandom function $h(x)$ as 64 8-bit integers and consider the first p of them. If the integer b_1 at position a_1 is not found at other position among the first p, then we count how many of these integers are less than b_1 and we set $\sigma(a_1) = n_1$, where n_1 is that number. If there are repetitions, we again count the number of integers less than b_1, but in order to decide the tie, we call again the pseudorandom function and we consider only as many 8-bit integers in the result as we have collisions (integers equal to b_1 in the initial set of "random" numbers). We repeat this process until no collision occurs.
- The key to the pseudorandom function is always chosen in a way so that the result will always be the same when we approach the same digit. For example, when we scramble different first digit, a_1', we must arrive at the same sequence of initial pseudorandom numbers and at the same sets of pseudorandom numbers during tie-braking.
- This is possible since we have enough freedom in choosing the key.

Owen scrambling for Halton sequence on GPU (cont.)

- The chacha pseudorandom function and stream cypher were defined with efficient implementation in mind and they are GPU-friendly, since they do not use much memory.
- We call the generation kernel as follows:
- gener_kernel«grid,block»(nonse,dim,N,x,areainfodevice);
- where we have dim3 grid(dim,1);
- dim3 block(blockingsize,1,1) with blockingsize=128
- Area info device holds all the information necessary for generating the next set of 128 terms of the sequence from the previous block.
- The most GPU time is spent inside the pseudorandom function, that is why we consider separately the cases of $p < 64$ and $p < 256$ (if $p < 64$ less invocations of the pseudorandom function are needed).

Numerical Results

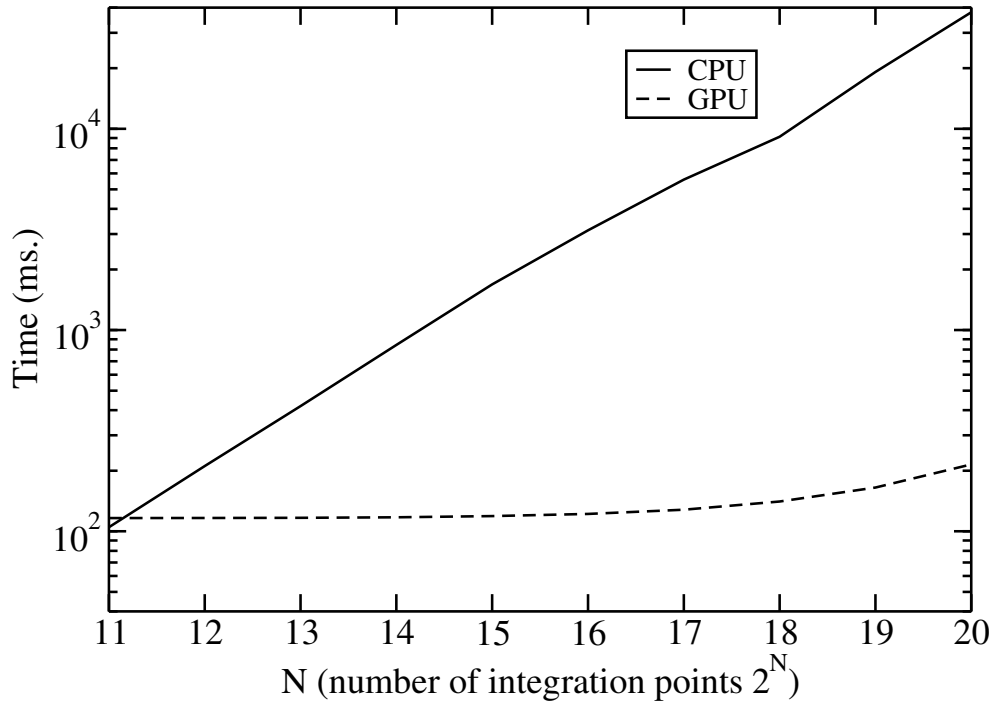

Figure 3: *Producing the scrambled sequence*

Numerical Results (Cont.)

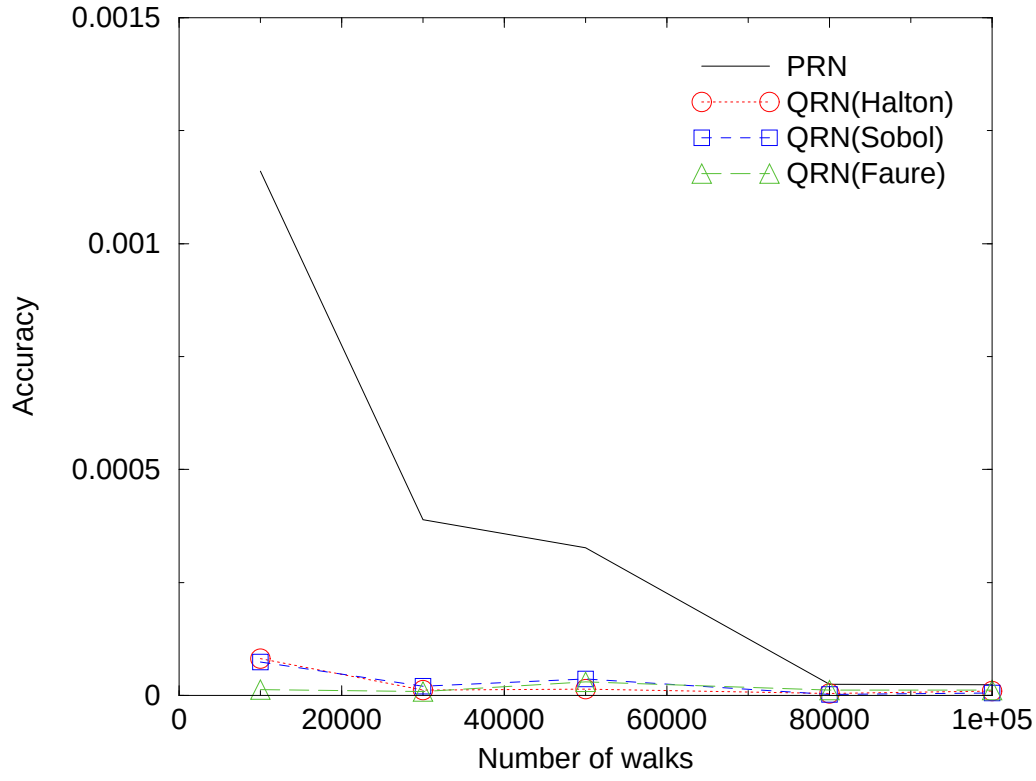

Figure 4: *Error when estimating sum of matrix-vector product, n=1024*

Numerical results (Cont.)

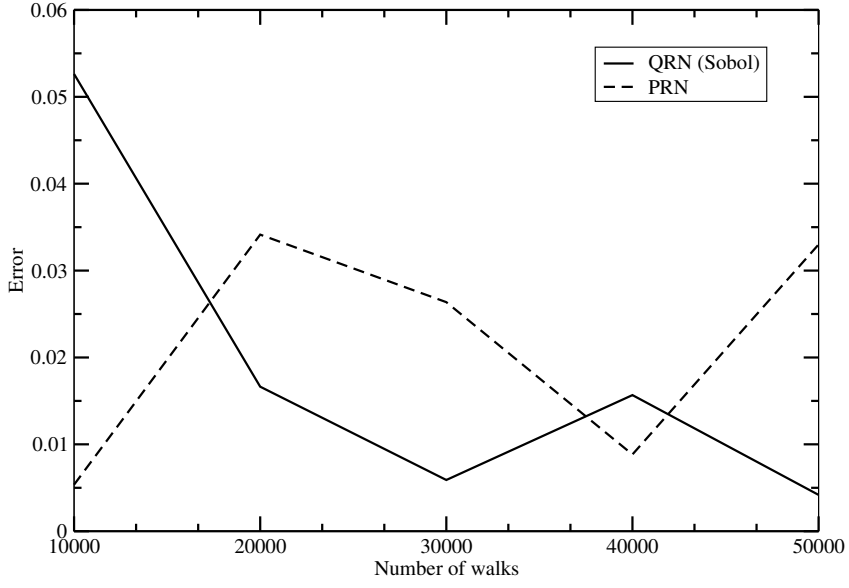

Figure 5: *MC and QMC: Accuracy versus number of walks for computing* $h^T A^k f$, *where* $k = 5, n = 1280$.

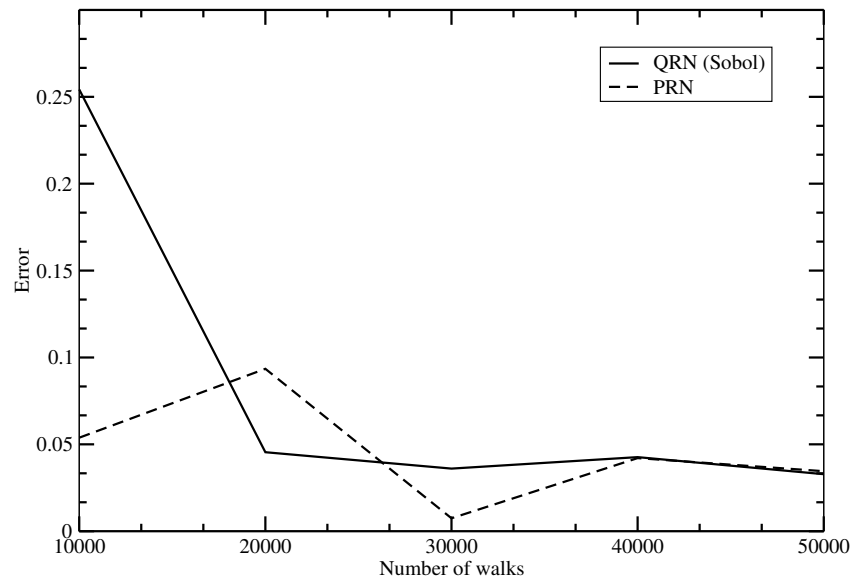

Figure 6: *MC and QMC: Accuracy versus number of walks for computing* $h^T A^k f$, *where* $k = 10, n = 1280$.

Conclusions and Future Directions

- Improvement obtained via quasi-random numbers
- The results demonstrate that the GPU offers a low-cost and high-performance computing solution for Owen scrambling
- GPU is more attractive as a high-performance computing device for quasirandom applications