PROCEEDINGS OF SCIENCE

GPU-Based Monte Carlo Methods for Solving Linear Algebraic Equations

Siyan Lai[1]
School of Data and Computer Science, Sun Yat-sen University
Guangzhou 510006, China
E-mail: laisy2@mail2.sysu.edu.cn

Jing Jin[2]
School of Data and Computer Science, Sun Yat-sen University
Guangzhou 510006, China
E-mail: jinj5@mail2.sysu.edu.cn

Menghan Guo
School of Data and Computer Science, Sun Yat-sen University
Guangzhou 510006, China
E-mail: guomh5@mail2.sysu.edu.cn

Xiaola Lin[3]
School of Data and Computer Science, Sun Yat-sen University
Guangzhou 510006, China
E-mail: linxl@mail.sysu.edu.cn

Many engineering, physics, chemistry, and computer science problems involve solving systems of linear algebraic equations (SLAE). It is an important issue in scientific computing field. In this paper we study the Monte Carlo methods (MCMs) for solving SLAE. We take the advantage of Graphic Processor Unit (GPU) to accelerate MCMs for solving SLAE. The result of numerical experiments demonstrates that GPU is very suitable for speeding up this application. Moreover, the accelerated ratio can be up 50X with the problem size increasing.

ISCC 2015
18-19, December, 2015
Guangzhou, China

[1]Speaker
[2]This study is supported by the National Natural Science Foundation of China under Grant No. 61472454
[3]Correspongding Author

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

PoS(ISCC2015)047

1.Introduction

Many engineering, physics, chemistry, and computer science problems involve solving systems of linear algebraic equations (SLAE). Solve SLAE

$$Ax = b, \quad (1.1)$$

Where $A \in R^{n \times n}$ is a given matrix, and $b=(b_1,b_2,\ldots,b_n)^t \in R^n$ is a given vector, $x \in R^n$ is the solution. The classical methods include direct methods, iterative methods and Monte Carlo methods(MCMs). MCMs for solving linear algebra problems can be traced back to 1950s [1].

There are some important features of these methods: 1) one of the important features of these methods is that they can be used for calculating only one component of the solutions. Therefore MCMs have advantages compared to other methods when only a few components are needed to be calculated. 2) The complexity of MCMs is independent of the problem size is another important feature. The time complexities of direct method and the iterative methods are $O(n^3)$ and $O(n^2k)$ respectively, where n is the size of solution vector and k is the number of iterations [2]. While in using MCMs, the time complexity is $O(N^{-0.5})$, where N is the number of Markov chains [3]. Using quasi-random sequences instead of pseudo-random sequences can accelerate the speed to $O((log^k N)/N)$ [4]. There are some improvements of sequences techniques presented [5,6]. For large systems, specifically for sparse large systems, MCMs have preferable performance, and only need to calculate none-zero elements [7]. 3) Another feature is that the solving process of each component is naturally independent. So we can conveniently parallel implement these methods. It implements Monte Carlo algorithms for sparse SLAE using MPI [8], authors tried to compress the data and minimize the communications during the computation [9]. Authors tried to use MapReduce computing model to implement MCMs for solving SLAEs [10]. 4) One more feature of MCMs is that the result is relatively rougher than a direct method and the iterative methods, so solutions are suitable for some scenes e.g. obtaining coarse estimation, and will be refined by other methods.

Considering that it is easy to parallel MCMs on many-core environment and GPU is a many-core computing device, we hope to implement this application on GPU to accelerate MCMs for solving SLAE. GPU are getting considerable attentions not only from image but also from computing, after NVIDIA introduces a GPU-based general purpose parallel computing architecture – CUDA™. Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable GPU has evolved into a multicore processor with tremendous computational horsepower and very high memory bandwidth . In CUDA, GPU works with CPU simultaneously. CPU takes charge of serial transactions which involve complex logic, while GPU is responsible for highly parallel tasks. We will implement and optimize both MCMs for solving SLAE on GPU to contrast with CPU. The performance of experiments is inspiring. The accelerate ratio can be up 50X with the problem size increasing.

We have organized the rest of this paper in the following way: in Section 2 some background information is introduced on the ongoing research. Then the implementation of MCMs on GPU for solving SLAE is presented and our work on optimizing implementation is presented in Section3. In Section 4 we use experiments to support our claim. Summarizes the key observations in Section 5.

2.Monte Carlo Methods for Solving Systems of Linear Equations

In this section, we briefly introduce the basic knowledge about MCMs and QMCMs for solving SLAE, which is helpful in understanding our work in following sections.

2.1 MCs for SLAE

Let us consider a system of linear algebraic equations mentioned in Eq. (1.1). It is known that the system of linear algebraic equation given by Eq. (1.1) can be rewritten in the following iterative form [7]:

$$x = Lx + f \quad . \quad (2.1)$$

PoS(ISCC2015)047

When matrix L satisfies $max_{1\le i\le n}\sum_{j=1}^{n}|l_{ij}|<1$, the iteration will converge. We call matrix L and vector f iterative matrix and iterative vector respectively.

The matrix L and vector f norms are determined as follows: $\|L\|_\infty = max_{1\le i\le n}\sum_{j=1}^{n}|l_{ij}|<1$, $\|f\|_\infty = max_{1\le i\le n}|f_i|$.

Then apply the iterations as follows:

$$\begin{gathered} x^{(0)}=f \\ \vdots \\ x^{(k)}=Lx^{(k-1)}f \\ \vdots \end{gathered} , \tag{2.2}$$

Therefore,

$$x=f+Lf+L^2 f+\cdots+L^{(k-1)}f+L^{(k)}f+\cdots . \tag{2.3}$$

Assume x_m is the m-th component of x, then x_m can be presented as follows:

$$x_m=f_m+\sum_{i_1=1}^{n} l_{mi_1}f_{i_1}+\sum_{i_1,l_2=1}^{n} l_{mi_1}l_{mi_2}f_{i_2}+\cdots+\sum_{i_1,l_2,\cdots,i_k=1}^{n} l_{mi_1}l_{mi_2}\cdots l_{mi_k}f_{i_k}+\cdots \tag{2.4}$$

Define the matrixes $B,P\in R^{n\times n}$ and $l_{mj}=b_{mj}p_{mj},b_{mj}$ and p_{mj} statisfy as follows:

$$\begin{cases} p_{mj}>0, if\ a_{mj}\neq 0 \\ p_{mj}\ge 0, if\ a_{mj}=0 \end{cases} \text{ and } \begin{cases} b_{mj}=\dfrac{a_{mj}}{p_{mj}}, if\ a_{mj}\neq 0 \\ b_{mj}=0, if\ a_{mj}=0 \end{cases} . \tag{2.5}$$

Hence Eq. (2.4) is transformed into:

$$x_m=f_m+\sum_{i_1=1}^{n} b_{mi_1}p_{mi_1}f_{i_1}+\sum_{i_1,l_2=1}^{n} b_{mi_1}b_{mi_2}p_{mi_1}p_{mi_2}f_{i_2}+\cdots+\sum_{i_1,l_2,\cdots,i_k=1}^{n} b_{mi_1}b_{mi_2}\cdots b_{mi_k}p_{mi_1}p_{mi_2}\cdots l_{mi_k}f_{i_k}+\cdots . \tag{2.6}$$

Suppose now chain S is an infinite discrete Markov chain with n states, we have

$$S=s_0\rightarrow s_1\rightarrow\cdots\rightarrow s_j\rightarrow\cdots \tag{2.7}$$

where $s_j\in n$ states $\{1,2,\dots,n\}$, for j=1,2,…, $s_0$is start state.

We further define the transition probability to state s_β from state s_α

$$P(s_k=\beta\mid s_{k-1}=\alpha)=p_{\alpha\beta} , \tag{2.8}$$

for α=1,2,..,n and β=1,2,..,n.

Suppose that $p_{\alpha\beta}$ satisfies Eq. (2.5) and $\sum_{=1}^{n\beta} p_{\alpha\beta}=1$. The probabilities $p_{\alpha\beta}$, for α, β=1,2,..,n, thus define the transition matrix P. The matrix B can be computed in terms of the transition matrix P and matrix L.

Define the random variables W_j according to the recursion:

$$W_j=W_{j-1}\times b_{s_{j-1}s_j}=b_{s_0s_1}b_{s_1s_2}\cdots b_{s_{j-1}s_j} , \quad W_0=1 . \tag{2.9}$$

Also define random variable θ^*:

$$\theta^*=\sum_{k=0}^{\infty} W_k f_{s_k} \tag{2.10}$$

$$E\theta^*=f_{s_0}+\sum_{s_1=1}^{n} b_{s_0s_1}p_{s_0s_1}f_{s_1}+\cdots+\sum_{s_1,s_2,\cdots,s_k=1}^{n} b_{s_0s_1}b_{s_1s_2}\cdots b_{s_{k-1}s_k}p_{s_0s_1}p_{s_1s_2}\cdots p_{s_{k-1}s_k}f_{s_k}+\cdots \tag{2.11}$$

Thus when s_0=m, Eq. (2.11) is equal to Eq. (2.6). It is clear that MCMs can solve only one component of the solution vector once and each component is solved independently.

According to Laws of Large Number:

$$\lim_{N\to\infty} P(|\frac{1}{N}\sum_{i=1}^{N}\theta_r^*-E\theta^*|>\varepsilon)=0 , \tag{2.12}$$

where N is the number of chains, θ_r^* is the value of θ^* in the r-th chain and ε is the probable error.

Hence, we have

PoS(ISCC2015)047

$$x_i \approx \frac{1}{N}\sum_{r=1}^{N}\theta_r^* \quad . \tag{2.13}$$

The key results concerning minimization of probable error and the definition of almost optimal transition frequency for MCMs applied to the calculation via iterated functions are presented [11].

The transition matrix P is chosen with elements

$$p_{\alpha\beta} = \frac{|\alpha_{\alpha\beta}|}{\sum_{\beta}|\alpha_{\alpha\beta}|} \quad \text{,for} \quad \alpha, \beta = 1, 2, \ldots, n. \quad . \tag{2.14}$$

It is close to the optimal transition frequency function [11].

2.2 Estimations of Error

Now we will outline the method of estimation of N and T.

(1) The estimation of length of chain, T

Because the length of the Markov chain is infinite in theory, but the length must be finite in simulation, the sum for θ^* must be stopped when $|W_i f_{s_i}| < \delta$, for a given δ. Therefore, for a given δ, the length of chain T can be estimated as follows:

$$|W_i f_{s_i}| = |b_{s_0 s_1} \cdots b_{s_{i-1} s_i}| |f_{s_i}| = \left| \frac{|a_{s_0 s_1} \cdots a_{s_{i-1} s_i}|}{\frac{|a_{s_0 s_1}|}{\sum_{s_1}|a_{s_0 s_1}|} \cdots \frac{|a_{s_0 s_1}|}{\sum_{s_i}|a_{s_0 s_1}|}} \right| |f_{s_i}| \le \|L\|^i \|f\| \tag{2.15}$$

$$T = i \le \frac{\log\left(\delta / \|f\|\right)}{\log(\|L\|)} \quad , \tag{2.16}$$

where δ is the truncation error.

(2) The estimation of number of chains, N

For a given error ε, according to Eq. (2.15), we have $|W_i f_{s_i}| \le \|L\|^i \|f\|$.

Then it follows that

$$|\theta^*| \le |W_0 f_{s_0}| + \cdots + |W_i f_{s_i}| + \cdots \le \|L\|^0 \|f\| + \cdots + \|L\|^0 \|f\| + \cdots = \frac{\|f\|}{(1-\|L\|)} \quad . \tag{2.17}$$

Thus

$$D\theta^* \le E\theta^{*2} \le \frac{\|f\|^2}{(1-\|L\|)^2} \quad . \tag{2.18}$$

According to the Central Limit Theorem:

$$N \ge \frac{0.6745^2}{\varepsilon^2} D\theta^* \quad , \tag{2.19}$$

and

$$N \ge \frac{0.6745^2}{\varepsilon^2} \frac{\|f\|^2}{(1-\|L\|)^2} \quad , \tag{2.20}$$

ε is the probable error.

It is clear that the number of chains, N, and the length of chain, T, are independent of matrix size n, and they only depends on the matrix norm and precision. In simulation, we can use the norm of row $\|L_i\|$ to replace norm $\|L\|_\infty$ for reducing N, and the length of the chain can be dynamically controlled via the truncation error δ.

3.Implementation and Optimizations

In this section, we will present details of how to use MCMs to solve linear equations on GPU along with some optimizations we make.

PoS(ISCC2015)047

3.1 MCMs on GPU for solving SLAE

From the above analysis, we notice that the calculation processes of different variables are irrelevant. By making use of this property, we can simply obtain a parallel version that each thread solve one individual unknown. This approach is rough, but it works if there are more than one unknown. However, in many scientific and engineering problems, though the size of the problem is large, we only need to know values of few unknowns, especially only one. In such cases, this approach acts almost like a single threaded process, which doesn't utilize the parallelism provided by GPU. As a result, a better approach with fine granularity is deserved.

Thinking about the implementation of MCMs for solving SLAE on CPU [3], we find that the job done by the first loop is simple: just iterate through each Sobol point and simulate a Markov chain, and different chains don't affect each other. This nature can be used to find the parallel algorithm with better granularity. We can partition this loop into multiple warps within each of them executed by one thread. Then we get the following parallel algorithm (which is a kernel function in CUDA terminologies) in Figure1.

Algorithm MCMs_GPU
Input: 1. *n*: the order of the linear equations; 2. *P*: the *n***n* probability transition matrix; 3. *B*: the *n***n* associated matrix; 4. *m*: the number of Sobol points; 5. *d*: the dimension of each Sobol point; 6. *Sobol*: *m***d* Sobol points matrix; 7. *f*: the iterative vector of size *n*; 8. *Idx*: the index of the unknown which we want to solve.
Output: 1. *partial_x*: one part of the value of the unknown (corresponding to one block).
1. *tid*←the current thread id 2. *num_thread*←the number of threads within a grid 3. ◇*x_array* is an array whose size is equal to the size of one block. 4. *x_array*[*tid*]←0 5. *i*←1 6. while $i<m$ 7. do *state*←*Idx* 8. *w*←1 9. *e*←*f*[*state*] 10. for *j*←1 to *d* 11. do *p*←*Sobol*[*j*] 12. *next_state*←the smallest k such that $\sum_{i=1}^{k} P\{state\}\{k\} \geq p$ 13. *w*←*w***B*[*state*][*next_state*] 14. e←*e*+*w***f*[*next_state*] 15. *state*←*next_state* 16. *x_array*[*tid*]←*x_array*[*tid*]+*e* 17. *i*←*i*+*num_thread* 18. ◇Synchronize the threads in one block. 19. SynchronizeTheads() 20. *partial_x*←ParallelPrefixSum(*x_array*[*tid*]) 21. return *partial_x*

Figure1: Monte Carlo Algorithm on GPU

PoS(ISCC2015)047

In the description of the algorithm, we use the Sobol sequences to simulate random walks. The process MC_GPU is executed by *num_thread* threads simultaneously, and the threads are organized as multiple blocks, with one block containing multiple threads. Each thread calculates one part of the answer of one unknown and stores the result to *x_array*[*tid*]. Since the threads within one block cannot communicate with the threads in other blocks, the array variable *x_array* is a local variable to each block. Upon completion of the simulation of Markov chain, we need to synchronize the threads in one block (line 19), and then invoke subroutine ParallelPrefixSum to sum up the values in one block and return the result.

Note that this routine only calculates one part of the result of one unknown. After the execution, we obtain multiple "*partial*_x" (one for each block), so we need to invoke ParallelPrefixSum again to sum up all these values, and divide the sum by the number of Sobol points to get the final answer. The ParallelPrefixSum routine can be implemented in O($log(n)$) time, where n is the size of the input array [12].

3.2 Optimizations

Given the above approach, several optimizations can be made in respect to both data structure and hardware environment. First of all, since x_array is a local variable to one block, we can define it as share memory instead of global memory in the CUDA C environment to accelerate the memory access speed. Secondly, in general cases, the size of the matrix n is large, while the matrix itself is sparse. In such situations, we can use sparse representation of matrix to reduce the memory usage by a large factor. Moreover, we notice that all values in the same row of matrix B are the same except the sign. We then can use only one bit to represent the sign, and allocate an array of size n to store the absolute values of the n row.

3.3 Task Partition

The task partition is a very important aspect of the GPU computation performance. Different partition strategies vary the performance greatly. According to former study, when the number of blocks is several times as many as the number of multiprocessors, it can achieve better occupancy and performance [13]. The reason why there should be multiple active blocks per multiprocessor is that blocks aren't waiting for a __syncthreads() which can keep the hardware busy. Additional factors include the register availability and the block size [14]. Because of these reasons, we make the size of each block a multiple of *Max_threads,* which is a factor of both the maximum number of threads per multiprocessor and the maximum number of threads per block. We also make the number of blocks a multiple of the number of multiprocessors, and 3 or 4 is enough according to the procedure of MC_GPU (because only one __syncthreads() is invoked).

4.Numerical Experiment

In this section, we present two numerical tests and show the test result. In the first test, we go to figure out the efficiency of random numbers on GPU. Then we make a test based on the algorithm described in the previous section with optimal random numbers. It compares the computing time of MCMs on GPU with CPU, and we expect to see a large acceleration factor. In our experiment environment, the CPU is Intel(R) Xeon E5620 and GPU is Nvidia C2050. E5620 has 4 cores and C2050 has 14 SMs, where each SM has 32 cores, 448 cores totally.

4.1 Efficiency of Random Numbers on GPU

We will use some numerical test to analyze the efficiency of random walks and quasi-random walks solving SLAE on GPU from accuracy, convergence rate and solving rate on GPU. We choose MTGP32 generator to generate pseudo random walks in random walks simulation [13]. Then in quasi-random walks simulation, we use Halton, Faure and Sobol sequences [4].

From Fig.2 we can know that the difference of accuracy of two methods is not obvious, quasi-random walks are a little better than random walks, also learn that the convergence rate of

PoS(ISCC2015)047

quasi-random walks are faster than random walks, consistent with previous theory. However, the gap of solving rate is huge between two methods, Fig.3 and Fig.4 tells us the gap steadies with the increasing computations where the the problem is dense condition and sparse conditon respectively. The reason is that when the random walks requirement grows, more random numbers are needed; in the random walks computing process, pseudo random numbers are newly generated in the new components, but the quasi-random walks just only generate the quasi sequence once for all components computing. Although non-zero numbers increasing with the size of problems, more computations are needed in simulating random walks, the advantages of saving computations of generating random numbers cannot be counteracted. Moreover, the quasi-random walks have higher convergence rate.

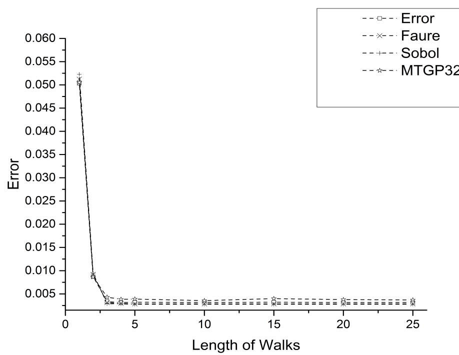

Figure2: Accuracy of Random Numbers

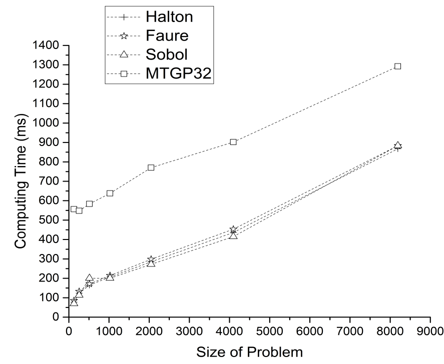

Figure3: Computing Time of Random Numbers

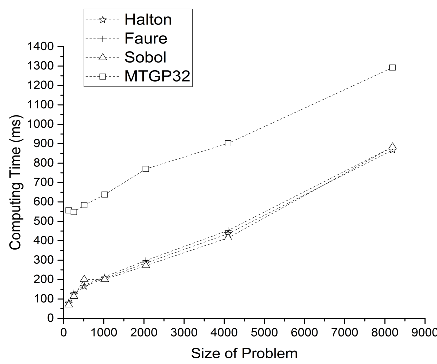

Figure4: Computing Time of Random Numbers

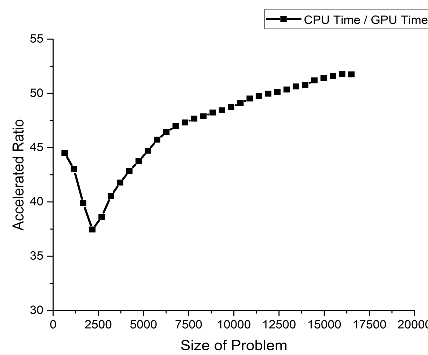

Figure5: Accelerated Ratio of the Two Algorithms

4.2 Running Time of MCMs_CPU and MCMs_GPU

In this test, we do experiments with vary sizes of problems. In all testings, we use the Sobol sequences. When running MCMs on GPU, we set the number of blocks per grid as 84 and the number of threads per block as 512, so that most of the threads run through only one Sobol point. We record the computing time and plot the result in Fig.5.

As we can see in Fig.5, the computing time on CPU is increasing logarithmically along with the problem size roughly because the complexity of MCMs_CPU is O(m*d*log(X)), where m(the number of random walks) and d(the length of random walks) are fixed, and X is sparse ratio. There are some outliers on the curve before average each point with the 10X running, but they don't affect the shape of the curve a lot. These outliers may be generated due to the shape of the random matrix or CPU scheduling.

For the runtime on GPU, the curve is much smoother, although it is still a logarithmical growth theoretically. The reason may be that the GPU is only scheduled for at most one job at any time and is idle for most of the time. Learn from results that the average speed-up ratio can be above 50X.

5. Conclusion

PoS(ISCC2015)047

Solving SLAE is an important issue in the scientific computing field. In this paper, we study on MCMs for solving SLAE. We propose an GPU version MCMs to accelerate the computation of solving SLAE problem. We make some optimizations for the MCMs to fit GPU architecture and CUDA programming model. In the implementation, we use compression technique to store probability transition matrix and associated matrix to reduce the memory demand. Furthermore, we design variables storage strategy according to GPU specific memory architecture to speed up computing. Moreover, we set a suitable number of threads in a grid and grids to hide the delay sufficiently. Then through simulating some kind of random numbers on GPU to solve SLAE, we find that quasi-random walks have a better performance than random walks. The performance of numerical experiments demonstrates that GPU can speed up the computing remarkable. The accelerated ratio increases with the scale of problem increasing.

References

[1] J H Curtiss. *"Monte Carlo" methods for the iteration of linear operators*[J]. Journal of Mathematics and Physics, 1953, 32(1): 209-232.

[2] D P Bertsekas, J N Tsitsiklis. *Parallel and distributed computation: numerical methods*[M]. Prentice-Hall, 1989: 110-130.

[3] V N Alexandrov. *Efficient parallel Monte Carlo methods for matrix computations*[J]. Mathematics and computers in Simulation, 1998, 47(2): 113-122.

[4] M Mascagni, A Karaivanova. *A parallel Quasi-Monte Carlo method for solving systems of linear equations*[M]//Computational Science—ICCS 2002. Springer Berlin Heidelberg, 2002: 598-608.

[5] H Ji, Y Li. *Reusing Random Walks in Monte Carlo Methods for Linear Systems*[J]. Procedia Computer Science, 2012, 9: 383-392.

[6] I Dimov, S Maire, J M Sellier. *A new Walk on Equations Monte Carlo method for solving systems of linear algebraic equations*[J]. Applied Mathematical Modelling, 2015,39(15):4494–4510

[7] J Straßburg, V N Alexandrov. *A Monte Carlo approach to sparse approximate inverse matrix computations*[J]. Procedia Computer Science, 2013, 18: 2307-2316.

[8] V Alexandrov, A Karaivanova. *Parallel Monte Carlo algorithms for sparse SLAE using MPI*[M]//Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer Berlin Heidelberg, 1999: 283-290.

[9] B F,Vajargah, K F Vajargah. *Parallel Monte Carlo computations for solving SLAE with minimum communications*[J]. Applied mathematics and computation, 2006, 183(1): 1-9

[10] P Jakovits, I Kromonov, S N Srirama. *Monte carlo linear system solver using mapreduce*[C]//Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on. IEEE, 2011: 293-299.

[11] I Dimov. *Minimization of the probable error for some Monte Carlo methods*[C]//Proc. Int. Conf. on Mathematical Modeling and Scientific Computation, Albena, Bulgaria, Sofia, Publ. House of the Bulgarian Academy of Sciences. 1991: 159-170.

[12] S Sengupta, M Harris, M Garland. Efficient parallel scan algorithms for GPUs[J]. NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003, 2008 (1): 1-17.

[13] M Harris, S Sengupta, J D Owens. *Parallel prefix sum (scan) with CUDA*[J]. GPU gems, 2007, 3(39): 851-876.

[14] T Bradley, J du Toit, et al. *Parallelisation Techniques for Random Number Generators*[M]. Hwu editor. GPU Gems: Emerald Edition. Morgan Kaufman, Amsterdam 2011:231-246.

PoS(ISCC2015)047