Multi-GPU Implementation of the Uniformization Method for Solving Markov Models

Marek Karwacki, Beata Bylina, and Jarosław Bylina
Institute of Mathematics
Marie Curie-Sklodowska University
Pl. M. Curie-Skłodowskiej 5, 20-031 Lublin, Poland
marek.karwacki@gmail.com
beatas@hektor.umcs.lublin.pl
jmbylina@hektor.umcs.lublin.pl

***Abstract*—Markovian models can generate very large sparse matrices, which are difficult to store and solve. A useful method for finding transient probabilities in Markovian models is the uniformization. The aim of this paper is to show that the performance of the uniformization can be improved using multi-GPU architecture. We propose partitioning scheme for HYB sparse matrix storage format and some optimization techniques adjusted so as to minimize communication between GPUs during iterative sparse matrix-vector multiplication, which is the most time consuming step. The results of experiments show that on multi-GPU we can solve larger matrices than on single device and accelerate computations in comparison to a multithreaded CPU. Computational test have been carried out in double precision for a wireless network models. Using multi-GPU we were able to solve model which is described by a matrix of the size 3.6×10^7.**

Keywords: Markovian models, uniformization method, GPU, multi-GPU, heterogeneous computations, parallel computing, wireless network models.

I. Introduction

MODERN graphic cards (GPU—graphics processing units) enable significant speeding up computations in scientific and engineering applications. However, a single GPU has not a big memory so it enables processing data of very limited sizes. To process greater data we need more than one GPU (multi-GPU).

Markov chains are a tool for modelling various complex systems. One of the problems appearing while using Markov chains to model complex systems is a very large size of the model and thereby implied a long computation time and memory consumption.

For a Markov model system we can compute transient probabilities of states at any moment of the model time by solving ordinary differential equation and then we can compute various characteristics of the model from the probabilities.

From a set of various methods we chose the uniformization method [9], [11] (developed also in [10], [6]), because this method usually achieves better results than others; moreover, it is numerically stable and allows us to control truncation errors easily. However, it is quite difficult to use it with huge matrices, because it requires a lot of matrix-vector multiplications (SpMV), where the matrix is sparse but the vector is dense.

In the paper [3] we use one GPU to accelerate the uniformization method for a wireless network model, particularly using an SpMV operation from the CUSP library. We obtain a significant speed-up for large (of more than 2.5 millions rows) matrices—about 8 times.

In this paper, we propose an original solution based on multithreading—in order to implement via CUDA the uniformization method on multi-GPU architectures. This approach utilizes all CPU cores and all GPUs to obtain the highest performance possible (higher than using only a CPU). Our solution is well suited to the case where CPUs are connected to several GPUs.

In the paper [7] authors investigate performance properties of SpMV with matrices of various sparsity patterns on a GPU cluster using special pJDS format (modification of the ELLPACK-R scheme [1], [14]) to represent a sparse matrix to reduce the memory needed. We use another storage scheme, namely the HYB scheme.

In our previous work [2] we studied performance of the sparse matrix-vector multiplication on GPU using a variety of matrix storage formats. This paper develops a partition of the matrix stored in the HYB format between GPUs.

The article outline is following. Section II presents briefly the uniformization method and its algorithm. In Section III we describe an architecture of the multi-GPUs within a single node. In Section IV we propose a multi-GPU algorithm to accelerate the uniformization method for large matrix sizes. In Section V we describe partitioning of the matrix between GPUs and its distribution between them, and the operation of the matrix-vector multiplication. Section VI shows some numerical results—we investigate the capacity of the multi-GPU memory for large sparse matrices, the time, and the speed-up. Section VII concludes the paper.

II. Uniformization

Analyzing transient probabilities we obtain a system of first order ordinary differential equations (ODE) which is to be solved to find the probabilities. The system can be written (as

This work was partially supported within the project N N516 479640 of the Ministry of Science and Higher Education of the Polish Republic (MNiSW) “Modele dynamiki transmisji, sterowania zatłoczeniem i jakością usług w Internecie”

a matrix-vector ODE) as follows:

$$\frac{dx(t)}{dt} = Q^T x(t), \tag{1}$$

where Q is a transition rate matrix of a Markov chain and $x(t)$ is an unknown column vector of the states' probabilities in the moment t.

For the equation (1) there exists an analytical solution of the form:

$$x(t) = x(0)e^{Q^T t}, \tag{2}$$

where $x(0)$ is an initial value of the probabilities vector.

The uniformization method serves solving ODE systems (1) for continuous-time Markov chains (CTMC). The idea of the method is the discretization of the CTMC, that is replacing the CTMC by a DTMC (a discrete-time Markov chain) and a Poisson process. The discretization of the CTMC is based on replacing its transition rate matrix Q by a stochastic matrix P of transitions' probabilities between states in a given time interval.

The probabilities matrix P can be obtained from Q by:

$$P^T = I + \frac{1}{\alpha} Q^T, \qquad \text{where} \qquad \alpha = \max_i |a_{ii}|.$$

Thus, we can write:

$$e^{Q^T t} = e^{\alpha P^T t - \alpha I t} = e^{-t\alpha} e^{(t\alpha)P^T}. \tag{3}$$

Multiplying the equation (3) by $x(0)$ and expanding it, we get a new form of the solution (2):

$$x(t) = \sum_{k=0}^{\infty} x(0) \frac{(\alpha t)^k}{k!} e^{-\alpha t} (P^T)^k. \tag{4}$$

A. The Truncation Error

Among merits of the uniformization method we can find an easy truncation error control facility. For the numerical computations we need to replace the infinite series (4) with a finite one, what means fixing the number of the operations and introducing a truncation error determining the accuracy of the computations.

Let:

$$x^*(t) = \sum_{k=0}^{L} x(0) \frac{(\alpha t)^k}{k!} e^{-\alpha t} (P^T)^k. \tag{5}$$

The truncation error is expressed as $\delta(t) = x(t) - x^*(t)$. Hence, we can obtain a number L defining the length of the finite series (5) needed to achieve a given accuracy ε:

$$||x(t) - x^*(t)||_\infty = 1 - \sum_{k=0}^{L} \frac{(\alpha t)^k}{k!} e^{-\alpha t} \le \varepsilon. \tag{6}$$

Moreover, in practical implementations we will subdivide the integration domain $\langle 0, t \rangle$ to prevent overflow issues. We divide the interval $\langle 0, t \rangle$ into l intervals of equal lengths t/l.

B. Algorithm of the uniformization method on a multicore CPU

Algorithm 1 computes the states' probabilities vector $x(t)$ in the moment t from the equation (5). In this algorithm inputs are: the matrix Q^T, the initial probabilities vector $x(0)$ and the truncation criterion ε. This algorithm is a multithreaded version using a multicore CPU with OpenMP directives and BLAS (Basic Linear Algebra Subprograms) functions, namely the Sparse BLAS from the MKL (Math Kernel Library) [13]—extensively threaded mathematic routines for applications that require the maximum performance.

Algorithm 1 The uniformization method for a multicore CPU

```
Require: $Q^T, \varepsilon, t, pt = x(0)$
Ensure: $pt = \left( \sum_{k=0}^{L} \frac{(\alpha t)^k}{k!} e^{-\alpha t} (P^T)^k \right) \cdot x(0)$
 1: $\alpha \leftarrow \max_i |q_{ii}|$, parallelized by OpenMP
 2: $P^T \leftarrow \frac{1}{\alpha} Q^T$, parallelized by MKL BLAS
 3: $P^T \leftarrow I + P^T$, parallelized by OpenMP
 4: $\Theta \leftarrow 100; l \leftarrow \alpha t/\Theta; t \leftarrow t/l; at \leftarrow \alpha t$, single thread
 5: Compute $L$ from formula (6) using $\varepsilon$, single thread
 6: for $i \leftarrow 1$ to $l$ do
 7:     $m \leftarrow pt$,
 8:     for $k \leftarrow 1$ to $L$ do
 9:         $r \leftarrow P^T * m;$, parallelized by MKL Sparse BLAS
10:         $m \leftarrow \frac{\alpha t}{k} * r$, parallelized by MKL BLAS
11:         $pt \leftarrow pt + m$, parallelized by MKL BLAS
12:     end for
13:     $pt \leftarrow pt * e^{-at}$, parallelized by MKL BLAS
14: end for
```

III. Multi-GPU architecture within a node

Graphics Processing Units (GPUs) have recently been used for many applications beyond graphics, introducing the term *general-purpose computation on graphics processing units* (GPGPU), owing to (among others) the CUDA (*Compute Unified Device Architecture*) [8] prepared by NVIDIA.

Nowadays computers with heterogeneous architecture—a multicore CPU and multi-GPUs—become more and more popular. The host system has multicore CPUs and is connected to two GPUs with a dedicated PCI-Express connection. The host and the GPUs have different memory spaces and an explicit memory copy is required to transfer data between them. Thus, an important problem of multi-GPU programming is accessing to the memory space of the other GPU and peer-to-peer (P2P) copy.

In the multi-GPU programming model, the CPU forms a multiple threads belonging to the same process. Each GPU is associated with a separate p-thread running on the CPU.

There exist a lot of libraries consisting of routines accelerating computations on one GPU—among others the CUSP [12] library for benchmarking, which supports multiple sparse matrices storage formats. In this article we are going to modify the operation of the sparse matrix-vector multiplication from the CUSP library for one GPU to get a good result on multi-GPUs with the HYB storage format.

In the experiments we used 2 $\times$ Tesla M2050 based on Fermi architecture with 448 CUDA Cores and 1.15 GHz clock speed. This model contains 3072MB of the total amount of the memory with the clock rate at 1546 MHz. Unlike the previous CUDA architectures, Fermi offers a much better performance during double precision computations.

IV. Sparse Matrix-Vector Multiplication on 2 GPUs

The matrix-vector multiplication is computationally intensive and wonderfully fits into our GPU architecture described in Section III.

For Markovian matrices we achieved the best performance with the HYB format [2], therefore in our further work we consider only this format.

Parallelization of the SpMV operation requires the decomposition and distribution of the coefficient matrix. Our approach is based on a one-dimensional (1D) scheme for partitioning a sparse matrix, with the goal of efficient parallelizing the SpMV operation on multi-GPUs.

The $n \times n$ matrix is partitioned among 2 GPUs, with each GPU storing complete rows of the matrix (row-wise 1D partitioning)—but every row in only one GPU. The whole $n{\times}1$ vector x is needed on each GPU. Since each GPU performs computations with the use of a $n/2 \times n$ matrix and the $n \times 1$ vector x, therefore after the multiplication each GPU holds a half-vector y of the size $n/2 \times 1$.

The HYB format combines the efficient memory bandwidth of ELLPACK and the flexibility of COO. The most common number of non-zeros per row is stored in the ELLPACK format and the rest of the elements in the COO format. Figure 1 shows an example of storing a primary matrix A in the HYB format, where ell_data, $ell\_indices$ represent the ELLPACK's part and coo_data, coo_row, coo_col belong to COO.

However, to process larger matrices we need to divide matrix between multiple GPUs. In our matrices the ELLPACK part contains over 90% of all the elements, so it is important to split it fairly evenly. We assume that COO vectors are sorted by rows, so we can divide them also by rows. By the fact that we split COO vectors by rows, the result vector after each matrix-vector multiplication has the same size and the communication needed for results exchange is balanced. Figure 2 illustrates concept of the matrix division and Figure 3 shows an example.

To minimize the time needed for the communication between the GPU cards we used peer-to-peer transfers, which allow to copy data from one GPU directly to another without the CPU memory involvement (peer-to-peer multi-GPU DMA allows to copy data in such a manner). Figure 4 presents the matrix-vector multiplication scheme on 2 GPUs. In the uniformization method algorithm the result vector y is additionally further processed.

V. Algorithms and Implementation Details

The multi-GPU implementation uses the same algorithm as described previously in Section II-B but the matrix-vector multiplication is performed on multi-GPUs. Some communications have to be added in order to exchange data between the GPUs. The pattern of the required communications is dependent on the way our matrix is scattered across the GPUs memory.

Algorithm 2 The uniformization method for 2 GPUs

```
Require: $Q^T, \varepsilon, t, pt = x(0)$
Ensure: $pt = \left( \sum_{k=0}^{L} \frac{(\alpha t)^k}{k!} e^{-\alpha t} (P^T)^k \right) \cdot x(0)$
 1: $\alpha \leftarrow \max_i |q_{ii}|$
 2: $P^T \leftarrow \frac{1}{\alpha} Q^T$
 3: $P^T \leftarrow I + P^T$
 4: $\Theta \leftarrow 100; l \leftarrow \alpha t/\Theta; t \leftarrow t/l; at \leftarrow \alpha t$
 5: Determine the $L$ needed to achieve a given accuracy $\varepsilon$
 6: Divide $P^T$ matrix into 2 matrices $P_0$, $P_1$ on host
 7: Copy $\{P_0, m\}$ to GPU$_0$ and $\{P_1, m\}$ to GPU$_1$
 8: for $i \leftarrow 1$ to $l$ do
 9:   for $k \leftarrow 1$ to $L$ do
10:     $m_0' \leftarrow P_0^T \cdot m_0 \parallel m_1' \leftarrow P_1^T \cdot m_1$
11:     $m_0' \leftarrow \frac{\alpha t}{k} \cdot m_0' \parallel m_1' \leftarrow \frac{\alpha t}{k} \cdot m_1'$
12:     $m_0 \leftarrow merge(m_0', m_1') \parallel m_1 \leftarrow merge(m_0', m_1')$
        {GPU$_0$ $\leftrightarrow$GPU$_1$}
13:     $pt_0 \leftarrow pt_0 + m_0 \parallel pt_1 \leftarrow pt_1 + m_1$
14:   end for
15:   $pt_0 \leftarrow pt_0 \cdot e^{-\alpha t} \parallel pt_1 \leftarrow pt_1 \cdot e^{-\alpha t}$
16:   $m_0 \leftarrow pt_0 \parallel m_1 \leftarrow pt_1$
17: end for
```

Algorithm 2 performs the uniformization method on 2 GPUs. First, we divide the input matrix into two matrices which are copied to respective GPUs. Then, we compute the matrix-vector product on each GPU and scale the results. Each GPU is assigned its own thread on CPU and operations denoted by $\parallel$ are performed in parallel. In step 7 we send the partial result vector to the other GPU. The remaining steps perform the same operations on the same data, but it is much faster than additional GPU$_0$ $\leftrightarrow$GPU$_1$ transfer.

VI. Numerical Experiments

A. A Wireless Network Model

To verify our multi-GPU implementation of the uniformization method we used a model of two identical wireless devices sharing a common channel—described in details in [5], [4]. Such models (and also this one) are impossible to solve analytically—and that is why it is to be solved numerically.

B. Testing Environment

In this section we compare the performance of the uniformization method algorithm on a multicore CPU, a single GPU and 2 GPUs. The codes were compiled using NVIDIA C

$$A = \begin{bmatrix} 4 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 9 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 1 & 3 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

$$ell\_data = \begin{bmatrix} 4 & 1 \\ 2 & 9 \\ 5 & * \\ 1 & 3 \\ 8 & * \end{bmatrix} \quad ell\_indices = \begin{bmatrix} 0 & 4 \\ 1 & 3 \\ 3 & * \\ 0 & 1 \\ 0 & * \end{bmatrix}$$

$$coo\_data = \begin{bmatrix} 2 & 1 \end{bmatrix} \quad coo\_col = \begin{bmatrix} 3 & 4 \end{bmatrix} \quad coo\_row = \begin{bmatrix} 3 & 3 \end{bmatrix}$$

Fig. 1. The HYB storage scheme

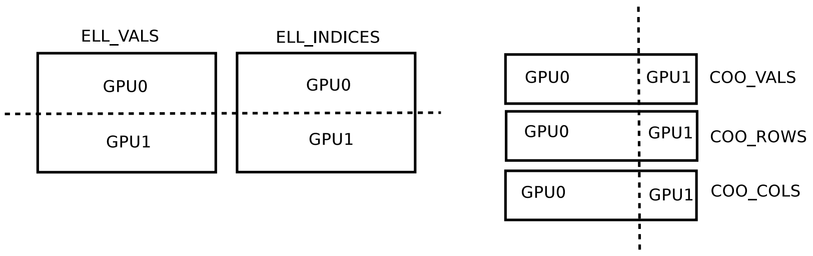

Fig. 2. The HYB format division

$$ell\_data_0 = \begin{bmatrix} 4 & 1 \\ 2 & 9 \end{bmatrix} \quad ell\_indices_0 = \begin{bmatrix} 0 & 4 \\ 1 & 3 \end{bmatrix}$$

$$coo\_data_0 = \begin{bmatrix} \quad \end{bmatrix} \quad coo\_col_0 = \begin{bmatrix} \quad \end{bmatrix} \quad coo\_row_0 = \begin{bmatrix} \quad \end{bmatrix}$$

$$ell\_data_1 = \begin{bmatrix} 5 & * \\ 1 & 3 \\ 8 & * \end{bmatrix} \quad ell\_ind_1 = \begin{bmatrix} 3 & * \\ 0 & 1 \\ 0 & * \end{bmatrix}$$

$$coo\_data_1 = \begin{bmatrix} 2 & 1 \end{bmatrix} \quad coo\_col_1 = \begin{bmatrix} 3 & 4 \end{bmatrix} \quad coo\_row_1 = \begin{bmatrix} 3 & 3 \end{bmatrix}$$

Fig. 3. An example of the HYB format division concept

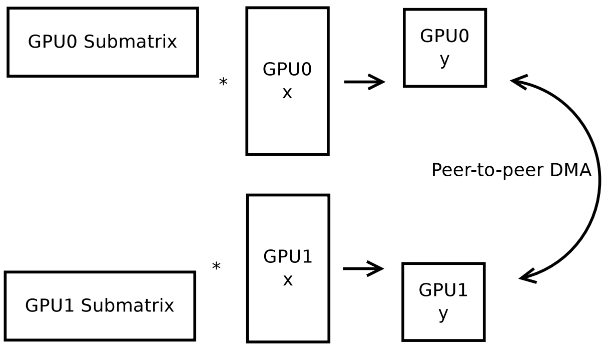

Fig. 4. SpMV for 2 GPUs on 1 host

TABLE I
THE HARDWARE AND SOFTWARE USED IN TESTS

CPU	2 × Intel Xeon X5650 2.67GHz (6 cores with HT)
Host memory	48 GB DDR3 1333 MHz
GPU	2 × Tesla M2050 (515 Gflop/s DP, 3 GB DDR5)
OS	Debian GNU Linux 6.0
Libraries	CUDA Toolkit 4.0, CUSP 0.3, MKL 10.3

TABLE II
THE PERFORMANCE OF THE UNIFORMIZATION METHOD

n [$\times 10^6$]	nz [$\times 10^6$]	$TCPU$ [s]	$TGPU$ [s]	$T2GPU$ [s]	$\frac{TCPU}{TGPU}$	$\frac{TCPU}{T2GPU}$
11	65	370	38	210	9.74	1.76
23	133	472	59	241	8.00	1.96
27	155	511	66	267	7.74	1.91
31	178	654	–	306	–	2.14
36	202	705	–	352	–	2.00

Compiler (nvcc) with the CUSP and MKL libraries. We used the hardware and software configuration shown in Table I.

All the computations were carried out in the double precision. We computed the transient probabilities vector for $t = 10$ with $\pi(0) = (1, 0, \ldots, 0)^T$ [10] of the appropriate length. We run the experiments with the accuracy tolerance parameter value $\varepsilon = 10^{-5}$. However, as shown in [3] t and ε parameters extends processing time linearly, so it is easy to estimate performance on different parameters.

In Table II we present the processing times in seconds of our implementations $TCPU$ (multicore CPU), $TGPU$ (single GPU), $T2GPU$ (2 GPUs).

Each matrix from our models contains about $6n$ non-zero elements in a row, so $TGPU$ algorithm stores $8n$ (the base matrix, the vectors x and y) elements per GPU and T2GPU only $5n$ per GPU, which allows processing of larger matrices. Unfortunately, the communication between GPUs takes 90% of time, therefore the performance on 2 GPUs is much lower than on single GPU, but still higher than on CPU.

VII. Conclusion

In this paper we have proposed a multi-GPU parallel implementation of the uniformization method for solving Markov chains with CUDA in the double precision.

Our approach also permits to solve problems of size 3.7×10^7. We have presented a new approach using multi-GPUs substantially reducing the cost of the uniformization method in comparison to the multicore CPU. We obtain speedup about 2 times.

Unfortunately, when we use GPUs, quite a lot of time is consumed by sending data from the CPU to the GPU and back and between GPUs. However, in the uniformization method we need only one transfer from the CPU to the GPU (the matrix) before computations, then we make $L \times l$ matrix-vector multiplications on GPU, and eventually have to send our result (the vector) from the GPU to the CPU.

But in the uniformization method the performance gap between a GPU and its PCI-Express is a bottleneck. The method is communication-intensive and multiple communications between GPU cards cause the lack of the nice scalability.

The main advantage of our approach is still some speed-up in comparison to a multicore CPU and a substantial benefit in the memory space in comparison to a single GPU, and hence, the bigger problems are possible to solve—what is very important in Markovian models.

References

[1] N. Bell, M. Garland: *Efficient Sparse Matrix-Vector Multiplication on CUDA*, NVIDIA Tech. Report No. NVR-2008-004, 2008.

[2] B. Bylina, J. Bylina, M. Karwacki: *Computational Aspects of GPU-accelerated Sparse Matrix-Vector Multiplication for Solving Markov Models*, Theoretical and Applied Informatics, 23 (2011), no. 2, ISSN 1896-5334, pp. 127–145.

[3] B. Bylina, M. Karwacki, J. Bylina: *A CPU-GPU Hybrid Approach to the Uniformization Method for Solving Markovian Models—A Case Study of a Wireless Network*, CCIS 291, Computer Networks 2012, pp. 401–410.

[4] J. Bylina, B. Bylina: *A Markovian Queuing Model of a WLAN Node*, CCIS 160, Computer Networks 2011, pp. 80–86.

[5] J. Bylina, B. Bylina, M. Karwacki: *A Markovian Model of a Network of Two Wireless Devices*, CCIS 291, Computer Networks 2012, pp. 411–420.

[6] N. J. Dingle, P. G. Harrison, W. J. Knottenbelt: *Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models*, Journal of parallel and distributed computing, 64 (2004), 908-920.

[7] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, A. R. Bishop: *Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation* CoRR abs/1112.5588: (2011).

[8] NVIDIA Corporation. CUDA Programming Guide. NVIDIA Corporation, 2009. http://www.nvidia.com/

[9] R. B. Sidje: *Expokit: A software package for computing matrix exponentials*, ACM Trans. Math. Software, 24 (1998), pp. 130–156.

[10] R. B. Sidje, K. Burrage, S. MacNamara: *Inexact Uniformization Method for Computing Transient Distributions of Markov Chains.* SIAM J. Scientific Computing 29(6): 2562–2580 (2007).

[11] W. J. Stewart: *Introduction to the numerical solution of Markov chains*, Princeton University Press, Princeton, NJ, 1994.

[12] http://code.google.com/p/cusp-library/

[13] http://software.intel.com/en-us/articles /intel-mkl/

[14] http://www.cs.purdue.edu/ellpack/